User Documentation for CVODE v2.5.0

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

November 6, 2006

UCRL-SM-208108

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

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.

Approved for public release; further dissemination unlimited

Contents

List of Tables vii

List of Figures ix

1 Introduction 1
- 1.1 Historical Background . . . 1
- 1.2 Changes from previous versions . . . 2
- 1.3 Reading this User Guide . . . 3

2 CVODE Installation Procedure 5
- 2.1 Configuration options . . . 6
- 2.2 Configuration examples . . . 10
- 2.3 Installed libraries and exported header files . . . 11
- 2.4 Building SUNDIALS without the configure script . . . 11

3 Mathematical Considerations 15
- 3.1 IVP solution . . . 15
- 3.2 Preconditioning . . . 18
- 3.3 BDF stability limit detection . . . 19
- 3.4 Rootfinding . . . 20

4 Code Organization 23
- 4.1 SUNDIALS organization . . . 23
- 4.2 CVODE organization . . . 23

5 Using CVODE for C Applications 27
- 5.1 Access to library and header files . . . 27
- 5.2 Data types . . . 28
- 5.3 Header files . . . 28
- 5.4 A skeleton of the user's main program . . . 29
- 5.5 User-callable functions . . . 31
  - 5.5.1 CVODE initialization and deallocation functions . . . 31
  - 5.5.2 Advice on choice and use of tolerances . . . 32
  - 5.5.3 Linear solver specification functions . . . 33
  - 5.5.4 CVODE solver function . . . 36
  - 5.5.5 Optional input functions . . . 37
    - 5.5.5.1 Main solver optional input functions . . . 38
    - 5.5.5.2 Dense linear solver . . . 44
    - 5.5.5.3 Band linear solver . . . 45
    - 5.5.5.4 SPILS linear solvers . . . 45
  - 5.5.6 Interpolated output function . . . 48
  - 5.5.7 Optional output functions . . . 48
    - 5.5.7.1 Main solver optional output functions . . . 50

5.5.7.2 Dense linear solver . . . 55
5.5.7.3 Band linear solver . . . 57
5.5.7.4 Diagonal linear solver . . . 58
5.5.7.5 SPILS linear solvers . . . 59
5.5.8 CVODE reinitialization function . . . 62
5.6 User-supplied functions . . . 63
5.6.1 ODE right-hand side . . . 63
5.6.2 Error message handler function . . . 64
5.6.3 Error weight function . . . 64
5.6.4 Jacobian information (direct method with dense Jacobian) . . . 65
5.6.5 Jacobian information (direct method with banded Jacobian) . . . 66
5.6.6 Jacobian information (matrix-vector product) . . . 67
5.6.7 Preconditioning (linear system solution) . . . 68
5.6.8 Preconditioning (Jacobian data) . . . 68
5.7 Rootfinding . . . 69
5.7.1 User-callable functions for rootfinding . . . 69
5.7.2 User-supplied function for rootfinding . . . 70
5.8 Preconditioner modules . . . 71
5.8.1 A serial banded preconditioner module . . . 71
5.8.2 A parallel band-block-diagonal preconditioner module . . . 75

6 FCVODE, an Interface Module for FORTRAN Applications 83
6.1 FCVODE routines . . . 83
6.1.1 Important note on portability . . . 84
6.2 Usage of the FCVODE interface module . . . 84
6.3 FCVODE optional input and output . . . 92
6.4 Usage of the FCVROOT interface to rootfinding . . . 94
6.5 Usage of the FCVBP interface to CVBANDPRE . . . 95
6.6 Usage of the FCVBBD interface to CVBBDPRE . . . 96

7 Description of the NVECTOR module 101
7.1 The NVECTOR_SERIAL implementation . . . 105
7.2 The NVECTOR_PARALLEL implementation . . . 107
7.3 NVECTOR functions used by CVODE . . . 109

8 Providing Alternate Linear Solver Modules 111
8.1 Initialization function . . . 111
8.2 Setup function . . . 112
8.3 Solve function . . . 113
8.4 Memory deallocation function . . . 113

9 Generic Linear Solvers in SUNDIALS 115
9.1 The DENSE module . . . 115
9.1.1 Type DenseMat . . . 116
9.1.2 Accessor Macros . . . 116
9.1.3 Functions . . . 116
9.1.4 Small Dense Matrix Functions . . . 117
9.2 The BAND module . . . 118
9.2.1 Type BandMat . . . 119
9.2.2 Accessor Macros . . . 119
9.2.3 Functions . . . 121
9.3 The SPGMR module . . . 121
9.3.1 Functions . . . 122
9.4 The SPBCG module . . . 122
9.4.1 Functions . . . 122

9.5 The SPTFQMR module . . . 123
9.5.1 Functions . . . 123

10 CVODE Constants **125**
10.1 CVODE input constants . . . 125
10.2 CVODE output constants . . . 125

Bibliography **129**

Index **131**

List of Tables

2.1 SUNDIALS libraries and header files (names are relative to *libdir* for libraries and to *includedir* for header files) . . . 12

5.1 Optional inputs for CVODE, CVDENSE, CVBAND, and CVSPILS . . . 39
5.2 Optional outputs from CVODE, CVDENSE, CVBAND, CVDIAG, and CVSPILS . . . 49

6.1 Keys for setting FCVODE optional inputs . . . 92
6.2 Description of the FCVODE optional output arrays `IOUT` and `ROUT` . . . 93

7.1 Description of the NVECTOR operations . . . 103
7.2 List of vector functions usage by CVODE code modules . . . 110

List of Figures

4.1 Organization of the SUNDIALS suite 24
4.2 Overall structure diagram of the CVODE package 25

9.1 Diagram of the storage for a matrix of type `BandMat` 120

Chapter 1

Introduction

CVODE is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [14]. This suite consists of CVODE, KINSOL, and IDA, and variants of these with sensitivity analysis capabilities.

1.1 Historical Background

FORTRAN solvers for ODE initial value problems are widespread and heavily used. Two solvers that have been written at LLNL in the past are VODE [1] and VODPK [3]. VODE is a general purpose solver that includes methods for stiff and nonstiff systems, and in the stiff case uses direct methods (full or banded) for the solution of the linear systems that arise at each implicit step. Externally, VODE is very similar to the well known solver LSODE [18]. VODPK is a variant of VODE that uses a preconditioned Krylov (iterative) method, namely GMRES, for the solution of the linear systems. VODPK is a powerful tool for large stiff systems because it combines established methods for stiff integration, nonlinear iteration, and Krylov (linear) iteration with a problem-specific treatment of the dominant source of stiffness, in the form of the user-supplied preconditioner matrix [2]. The capabilities of both VODE and VODPK have been combined in the C-language package CVODE [8].

At present, CVODE contains three Krylov methods that can be used in conjuction with Newton iteration: the GMRES (Generalized Minimal RESidual) [19], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [20], and TFQMR (Transpose-Free Quasi-Minimal Residual) linear iterative methods [9]. As Krylov methods, these require almost no matrix storage for solving the Newton equations as compared to direct methods. However, the algorithms allow for a user-supplied preconditioner matrix, and for most problems preconditioning is essential for an efficient solution. For very large stiff ODE systems, the Krylov methods are preferable over direct linear solver methods, and are often the only feasible choice. Among the three Krylov methods in CVODE, we recommend GMRES as the best overall choice. However, users are encouraged to compare all three, especially if encountering convergence failures with GMRES. Bi-CGFStab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for GMRES depends on the desired Krylov subspace size.

In the process of translating the VODE and VODPK algorithms into C, the overall CVODE organization has been changed considerably. One key feature of the CVODE organization is that the linear system solvers comprise a layer of code modules that is separated from the integration algorithm, allowing for easy modification and expansion of the linear solver array. A second key feature is a separate module devoted to vector operations; this facilitated the extension to multiprosessor environments with minimal impacts on the rest of the solver, resulting in PVODE [6], the parallel variant of CVODE.

Recently, the functionality of CVODE and PVODE has been combined into one single code, simply called CVODE. Development of the new version of CVODE was concurrent with a redesign of the vector operations module across the SUNDIALS suite. The key feature of the new NVECTOR module is that it is written in terms of abstract vector operations with the actual vector kernels attached by a particular implementation (such as serial or parallel) of NVECTOR. This allows writing the SUNDIALS solvers in a manner independent of the actual NVECTOR implementation (which can be user-supplied), as well as allowing more than one NVECTOR module linked into an executable file.

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

1.2 Changes from previous versions

Changes in v2.5.0

The main changes in this release involve a rearrangement of the entire SUNDIALS source tree (see §4.1). At the user interface level, the main impact is in the mechanism of including SUNDIALS header files which must now include the relative path (e.g. `#include <cvode/cvode.h>`). Additional changes were made to the build system: all exported header files are now installed in separate subdirectories of the instaltion *include* directory.

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

Changes in v2.4.0

CVSPBCG and CVSPTFQMR modules have been added to interface with the Scaled Preconditioned Bi-CGstab (SPBCG) and Scaled Preconditioned Transpose-Free Quasi-Minimal Residual (SPTFQMR) linear solver modules, respectively (for details see Chapter 5). Corresponding additions were made to the FORTRAN interface module FCVODE. At the same time, function type names for Scaled Preconditioned Iterative Linear Solvers were added for the user-supplied Jacobian-times-vector and preconditioner setup and solve functions.

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

To reduce the possibility of conflicts, the names of all header files have been changed by adding unique prefixes (`cvode_` and `sundials_`). When using the default installation procedure, the header files are exported under various subdirectories of the target `include` directory. For more details see Chapter 2.

Changes in v2.3.0

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

Changes in v2.2.1

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

Changes in v2.2.0

The major changes from the previous version involve a redesign of the user interface across the entire SUNDIALS suite. We have eliminated the mechanism of providing optional inputs and extracting optional statistics from the solver through the `iopt` and `ropt` arrays. Instead, CVODE now provides a set of routines (with prefix `CVodeSet`) to change the default values for various quantities controlling the solver and a set of extraction routines (with prefix `CVodeGet`) to extract statistics after return from the main solver routine. Similarly, each linear solver module provides its own set of `Set-` and `Get`-type routines. For more details see §5.5.5 and §5.5.7.

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

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

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

1.3 Reading this User Guide

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

There are different possible levels of usage of CVODE. The most casual user, with a small IVP problem only, can get by with reading §3.1, then Chapter 5 through §5.5.4 only, and looking at examples in [15]. In a different direction, a more expert user with an IVP problem may want to (a) use a package preconditioner (§5.8), (b) supply his/her own Jacobian or preconditioner routines (§5.6), (c) do multiple runs of problems of the same size (§5.5.8), (d) supply a new NVECTOR module (Chapter 7), or even (e) supply a different linear solver module (§4.2 and Chapter 9).

The structure of this document is as follows:

- In Chapter 2 we begin with instructions for the installation of CVODE, within the structure of SUNDIALS.
- In Chapter 3, we give short descriptions of the numerical methods implemented by CVODE for the solution of initial value problems for systems of ODEs.
- The following chapter describes the structure of the SUNDIALS suite of solvers (§4.1) and the software organization of the CVODE solver (§4.2).
- Chapter 5 is the main usage document for CVODE for C applications. It includes a complete description of the user interface for the integration of ODE initial value problems.
- In Chapter 6, we describe FCVODE, an interface module for the use of CVODE with FORTRAN applications.
- Chapter 7 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, and details on the two NVECTOR implementations provided with SUNDIALS: a serial implementation (§7.1) and a parallel implementation based on MPI (§7.2).
- Chapter 8 describes the interfaces to the linear solver modules, so that a user can provide his/her own such module.
- Chapter 9 describes in detail the generic linear solvers shared by all SUNDIALS solvers.
- Finally, Chapter 10 lists the constants used for input to and output from CVODE.

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.

Acknowledgments. We wish to acknowledge the contributions to previous versions of the CVODE and PVODE codes and user guides of Scott D. Cohen [7] and George D. Byrne [5].

Chapter 2

CVODE Installation Procedure

The installation of CVODE is accomplished by installing the SUNDIALS suite as a whole, according to the instructions that follow. The same procedure applies whether or not the downloaded file contains solvers other than CVODE.[1]

The SUNDIALS suite (or individual solvers) are distributed as compressed archives (`.tgz`). The name of the distribution archive is of the form *solver*-`x.y.z.tgz`, where *solver* is one of: `sundials`, `cvode`, `cvodes`, `ida`, or `kinsol`, and `x.y.z` represents the version number (of the SUNDIALS suite or of the individual solver).

To begin the installation, first uncompress and expand the sources, by issuing

```
% tar xzf solver-x.y.z.tgz
```

This will extract source files under a directory *solver*-`x.y.z`.

The installation procedure outlined below will work on commodity LINUX/UNIX systems without modification. However, users are still encouraged 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 directory created above.

In the remainder of this chapter, we make the following distinctions:

- *srcdir*

  is the directory *solver*-`x.y.z` created above; i.e., the directory containing the SUNDIALS sources.

- *builddir*

  is the directory under which SUNDIALS is built; i.e., the directory from within which the `configure` command is issued. Usually, this is the same as *srcdir*.

- *instdir*

  is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory *instdir*`/include` while libraries are installed under *instdir*`/lib`, with *instdir* specified with the `--prefix` flag to `configure`. See §2.1 for more details on the installation directories, including the special cases of the SUNDIALS examples and the SUNDIALSTB toolbox.

Note: The installation directory *instdir* should *not* be the same as the source directory *srcdir*.

[1] Files for both the serial and parallel versions of CVODE are included in the distribution. For users in a serial computing environment, the files specific to parallel environments (which may be deleted) are as follows: all files in `src/nvec_par/`; `cvode_bbdpre.c, cvode_bbdpre_impl.h` (in `src/cvode/`); `cvode_bbdpre.h` (in `include/cvode/`); `fcvbbd.c, fcvbbd.h` (in `src/cvode/fcmix/`); all files in `examples/cvode/parallel/`; all files in `examples/cvode/fcmix_parallel/`. (By "serial version" of CVODE we mean the CVODE solver with the serial NVECTOR module attached, and similarly for "parallel version".)

The installation steps for SUNDIALS can be as simple as

```
% tar xzf solver-x.y.z.tgz
% cd solver-x.y.z
% ./configure
% make
% make install
```

in which case the SUNDIALS header files and libraries are installed under `/usr/local/include` and `/usr/local/lib`, respectively. Note that, by default, neither the example programs nor the SUNDIALSTB toolbox are built and installed.

If disk space is a priority, then to delete all temporary files created by building SUNDIALS, issue

```
% make clean
```

To prepare the SUNDIALS distribution for a new install (using, for example, different options and/or installation destinations), issue

```
% make distclean
```

2.1 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=`/usr/local

`--exec-prefix=EPREFIX`

Location for architecture-dependent files.

Default: `EPREFIX=`/usr/local

`--includedir=DIR`

Alternate location for installation of header files.

Default: `DIR=PREFIX/include`

`--libdir=DIR`

Alternate location for installation of libraries.

Default: `DIR=EPREFIX/lib`

`--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`.

`--enable-examples`

Available example programs are *not* built by default. Use this option to enable compilation of all pertinent example programs. Upon completion of the `make` command, the example executables will be created under solver-specific subdirectories of *builddir*`/examples`:

builddir/examples/*solver*/serial : serial C examples

builddir/examples/*solver*/parallel : parallel C examples

builddir/examples/*solver*/fcmix_serial : serial FORTRAN examples

builddir/examples/*solver*/fcmix_parallel : parallel FORTRAN examples

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

`--with-examples-instdir=EXINSTDIR`

Alternate location for example executables and sample output files (valid only if examples are enabled). Note that installtion of example files can be completely disabled by issuing `EXINSTDIR=no` (in case building the examples is desired only as a test of the SUNDIALS libraries).

Default: `DIR=EPREFIX/examples`

`--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`

Users should *not* build SUNDIALS with support for single-precision floating-point arithmetic on 32- or 64-bit systems. This will almost certainly result in unreliable numerical solutions. The configuration option `--with-precision=single` is intended for systems on which single-precision arithmetic involves at least 14 decimal digits.

Options for Fortran support

`--disable-fcmix`

Using this option will disable all FORTRAN support. The FCVODE, FKINSOL, FIDA, 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 FCVODE, FKINSOL, FIDA, 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 FCVODE, FKINSOL, FIDA, 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` and `LIBDIR=MPIDIR/lib`

`--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, FKINSOL, and FIDA 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 Matlab support

The following options are relevant only for configuring and building the SUNDIALSTB Matlab toolbox:

`--enable-sundialsTB`

The SUNDIALSTB Matlab toolbox is *not* built by default. Use this option to enable configuration and compilation of the `mex` files. Upon completion of the `make` command, the following `mex` files will be created:

builddir`/sundialsTB/cvodes/cvm/cvm.`*mexext*

builddir`/sundialsTB/idas/idm/idm.`*mexext*

builddir`/sundialsTB/kinsol/kim/kim.`*mexext*

where *mexext* is the platform-specific extension of `mex` files.

`--with-sundialsTB-instdir=STBINSTDIR`

Alternate location for the installed SUNDIALSTB toolbox (valid only if SUNDIALSTB is enabled). As for the example programs, installation of SUNDIALSTB can be completely disabled by issuing `STBINSTDIR=no` (in case building the toolbox is desired but its installtion will be done manually afterwards). Otherwise, all required SUNDIALSTB files will be installed under the directory `STBINSTDIR/sundialsTB`.

Default: `DIR=MATLAB/toolbox` (see below for the definition of `MATLAB`).

`--with-matlab=MATLAB`

This option can be used to specify the location of the Matlab executable. The default is to search the path.

`--with-mexopts=ARG`

Specify the `mex` options file to be used.

Default: Standard Matlab `mex` options file.

`--with-mexflags=ARG`

Specify the `mex` compiler flags to be used.

Default: `ARG=-O`

`--with-mexldadd=ARG`

Specify additional `mex` linker flags.

Default: none

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.

2.2 Configuration examples

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

To build SUNDIALS using the default C and Fortran compilers, and default `mpicc` and `mpif77` parallel compilers, enable compilation of examples, build the Matlab `mex` files for SUNDIALSTB, and install it under `/home/myname/matlab/sundialsTB`, use

```
% configure --prefix=/home/myname/sundials --enable-examples \
            --enable-sundialsTB --with-sundialsTB-instdir=/home/myname/matlab
```

To disable installation of the examples, use:

```
% configure --prefix=/home/myname/sundials \
            --enable-examples --with-examples-instdir=no \
            --enable-sundialsTB --with-sundialsTB-instdir=/home/myname/matlab
```

The following 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 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 next 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 explicitly disables the examples (which means a FORTRAN compiler is not required). The `--with-mpi-libs` option is required so that the configure script can check if `gcc` can link with the appropriate MPI library.

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

2.3 Installed libraries and exported header files

Using the standard SUNDIALS build system, the command

```
% make install
```

will install the libraries under *libdir* and the public header files under *includedir*. The default values for these directories are *instdir*/`lib` and *instdir*/`include`, respectively, but can be changed using the configure script options `--prefix`, `--exec-prefix`, `--includedir` and `--libdir` (see §2.1). For example, a global installation of SUNDIALS on a *NIX system could be accomplished using

```
% configure --prefix=/opt/sundials-2.1.1
```

Although all installed libraries reside under *libdir*, the public header files are further organized into subdirectories under *includedir*.

The installed libraries and exported header files are listed for reference in Table 2.1. The file extension *.lib* is typically `.so` for shared libraries and `.a` for static libraries (see *Options for library support* for additional details).

A typical user program need not explicitly include any of the shared SUNDIALS header files from under the *includedir*/`sundials` directory since they are explicitly included by the appropriate solver header files (*e.g.*, `cvode_dense.h` includes `sundials_dense.h`). However, it is both legal and safe to do so (*e.g.*, the functions declared in `sundials_smalldense.h` could be used in building a preconditioner).

2.4 Building SUNDIALS without the configure script

If the `configure` script cannot be used (*e.g.*, when building SUNDIALS under Microsoft Windows without using Cygwin), or if the user prefers to own the build process (*e.g.*, when SUNDIALS is incorporated into a larger project with its own build system), then the header and source files for a given module can be copied from the *srcdir* to some other location and compiled separately.

The following files are required to compile a SUNDIALS solver module:

- public header files located under *srcdir*/`include/`*solver*
- implementation header files and source files located under *srcdir*/`src/`*solver*
- (optional) FORTRAN/C interface files located under *srcdir*/`src/`*solver*/`fcmix`
- shared public header files located under *srcdir*/`include/sundials`
- shared source files located under *srcdir*/`src/sundials`
- (optional) NVECTOR_SERIAL header and source files located under *srcdir*/`include/nvector` and *srcdir*/`src/nvec_ser`
- (optional) NVECTOR_PARALLEL header and source files located under *srcdir*/`include/nvector` and *srcdir*/`src/nvec_par`
- configuration header file `sundials_config.h` (see below)

A sample header file that, appropriately modified, can be used as `sundials_config.h` (otherwise created automatically by the `configure` script) is provided below. The various preprocessor macros defined within `sundials_config.h` have the following uses:

- Precision of the SUNDIALS `realtype` type

  Only one of the macros `SUNDIALS_SINGLE_PRECISION`, `SUNDIALS_DOUBLE_PRECISION` and `SUNDIALS_EXTENDED_PRECISION` should be defined to indicate if the SUNDIALS `realtype` type is an alias for `float`, `double`, or `long double`, respectively.

Table 2.1: SUNDIALS libraries and header files (names are relative to *libdir* for libraries and to *includedir* for header files)

SHARED	Libraries	n/a	
	Header files	sundials/sundials_types.h	sundials/sundials_math.h
		sundials/sundials_config.h	sundias/sundials_nvector.h
		sundials/sunials_smalldense.h	sundials/sundials_dense.h
		sundials/sundials_iterative.h	sundials/sundials_band.h
		sundials/sundials_spbcgs.h	sundials/sundials_sptfqmr.h
		sundials/sundials_spgmr.h	
NVECTOR_SERIAL	Libraries	libsundials_nvecserial.*lib*	libsundials_fnvecserial.a
	Header files	nvector/nvector_serial.h	
NVECTOR_PARALLEL	Libraries	libsundials_nvecparallel.*lib*	libsundials_fnvecparallel.a
	Header files	nvector/nvector_parallel.h	
CVODE	Libraries	libsundials_cvode.*lib*	libsundials_fcvode.a
	Header files	cvode/cvode.h	
		cvode/cvode_dense.h	cvode/cvode_band.h
		cvode/cvode_diag.h	cvode/cvode_spils.h
		cvode/cvode_bandpre.h	cvode/cvode_bbdpre.h
		cvode/cvode_spgmr.h	cvode/cvode_spbcgs.h
		cvode/cvode_sptfqmr.h	cvode/cvode_impl.h
CVODES	Libraries	libsundials_cvodes.*lib*	
	Header files	cvodes/cvodes.h	
		cvodes/cvodes_dense.h	cvodes/cvodes_band.h
		cvodes/cvodes_diag.h	cvodes/cvodes_spils.h
		cvodes/cvodes_bandpre.h	cvodes/cvodes_bbdpre.h
		cvodes/cvodes_spgmr.h	cvodes/cvodes_spbcgs.h
		cvodes/cvodes_sptfqmr.h	cvodes/cvodes_impl.h
		cvodes/cvodea_impl.h	
IDA	Libraries	libsundials_ida.*lib*	libsundials_fida.a
	Header files	ida/ida.h	
		ida/ida_dense.h	ida/ida_band.h
		ida/ida_spils.h	ida/ida_spgmr.h
		ida/ida_spbcgs.h	ida/ida_sptfqmr.h
		ida/ida_bbdpre.h	ida/ida_impl.h
KINSOL	Libraries	libsundials_kinsol.*lib*	libsundials_fkinsol.a
	Header files	kinsol/kinsol.h	
		kinsol/kinsol_dense.h	kinsol/kinsol_band.h
		kinsol/kinsol_spils.h	kinsol/kinsol_spgmr.h
		kinsol/kinsol_spbcgs.h	kinsol/kinsol_sptfqmr.h
		kinsol/kinsol_bbdpre.h	kinsol/kinsol_impl.h

- Use of generic math functions

  If `SUNDIALS_USE_GENERIC_MATH` is defined, then the functions in `sundials_math.(h,c)` will use the `pow`, `sqrt`, `fabs`, and `exp` functions from the standard math library (see `math.h`), regardless of the definition of `realtype`. Otherwise, if `realtype` is defined to be an alias for the `float` C-type, then SUNDIALS will use `powf`, `sqrtf`, `fabsf`, and `expf`. If `realtype` is instead defined to be a synonym for the `long double` C-type, then `powl`, `sqrtl`, `fabsl`, and `expl` will be used.

  Note: Although the `powf/powl`, `sqrtf/sqrtl`, `fabsf/fabsl`, and `expf/expl` routines are not specified in the ANSI C standard, they are ISO C99 requirements. Consequently, these routines will only be used if available.

- Fortran name-mangling scheme

  The macros given below are used to transform the C-language function names defined in the Fortran-C inteface modules in a manner consistent with the preferred Fortran compiler, thus allowing native C functions to be called from within a Fortran subroutine. The name-mangling scheme can be specified either by appropriately defining the parameterized macros (using the stringization operator, `##`, if necessary)

  - `F77_FUNC(name,NAME)`
  - `F77_FUNC_(name,NAME)`

  or by defining *one* macro from each of the following lists:

  - `SUNDIALS_CASE_LOWER` or `SUNDIALS_CASE_UPPER`
  - `SUNDIALS_UNDERSCORE_NONE`, `SUNDIALS_UNDERSCORE_ONE`, or `SUNDIALS_UNDERSCORE_TWO`

  For example, to specify that mangled C-language function names should be lowercase with one underscore appended include either

```
#define F77_FUNC(name,NAME) name ## _
#define F77_FUNC_(name,NAME) name ## _
```

  or

```
#define SUNDIALS_CASE_LOWER 1
#define SUNDIALS_UNDERSCORE_ONE 1
```

  in the `sundials_config.h` header file.

- Use of an MPI communicator other than `MPI_COMM_WORLD` in Fortran

  If the macro `SUNDIALS_MPI_COMM_F2C` is defined, then the MPI implementation used to build SUNDIALS defines the type `MPI_Fint` and the function `MPI_Comm_f2c`, and it is possible to use MPI communicators other than `MPI_COMM_WORLD` with the Fortran-C interface modules.

```
/*
 * -----------------------------------------------------------------
 * Copyright (c) 2005, The Regents of the University of California.
 * Produced at the Lawrence Livermore National Laboratory.
 * All rights reserved.
 * For details, see sundials/shared/LICENSE.
 * -----------------------------------------------------------------
 * SUNDIALS configuration header file
 * -----------------------------------------------------------------
 */


/* Define SUNDIALS version number
 * ------------------------------ */

#define SUNDIALS_PACKAGE_VERSION "2.2.1"

/* Define precision of SUNDIALS data type 'realtype'
 * ------------------------------------------------- */

/* Define SUNDIALS data type 'realtype' as 'double' */
#define SUNDIALS_DOUBLE_PRECISION 1

/* Define SUNDIALS data type 'realtype' as 'float' */
/* #define SUNDIALS_SINGLE_PRECISION 1 */

/* Define SUNDIALS data type 'realtype' as 'long double' */
/* #define SUNDIALS_EXTENDED_PRECISION 1 */

/* Use generic math functions
 * -------------------------- */

#define SUNDIALS_USE_GENERIC_MATH 1

/* FCMIX: Define Fortran name-mangling macro
 * ----------------------------------------- */

#define F77_FUNC(name,NAME) name ## _
#define F77_FUNC_(name,NAME) name ## _

/* FCMIX: Define case of function names
 * ------------------------------------ */

/* FCMIX: Make function names lowercase */
/* #define SUNDIALS_CASE_LOWER 1 */

/* FCMIX: Make function names uppercase */
/* #define SUNDIALS_CASE_UPPER 1 */

/* FCMIX: Define number of underscores to append to function names
 * --------------------------------------------------------------- */

/* FCMIX: Do NOT append any underscores to functions names */
/* #define SUNDIALS_UNDERSCORE_NONE 1 */

/* FCMIX: Append ONE underscore to function names */
/* #define SUNDIALS_UNDERSCORE_ONE 1 */

/* FCMIX: Append TWO underscores to function names */
/* #define SUNDIALS_UNDERSCORE_TWO 1 */

/* FNVECTOR: Allow user to specify different MPI communicator
 * ---------------------------------------------------------- */

#define SUNDIALS_MPI_COMM_F2C 1
```

Chapter 3

Mathematical Considerations

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

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

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

3.1 IVP solution

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

$$\sum_{i=0}^{K_1} \alpha_{n,i} y^{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} \dot{y}^{n-i} = 0\,. \tag{3.2}$$

Here the y^n are computed approximations to $y(t_n)$, and $h_n = t_n - t_{n-1}$ is the step size. The user of CVODE must choose appropriately one of two multistep methods. For nonstiff problems, CVODE includes the Adams-Moulton formulas , characterized by $K_1 = 1$ and $K_2 = q$ above, where the order q varies between 1 and 12. For stiff problems, CVODE includes the Backward Differentiation Formulas (BDFs) in so-called fixed-leading coefficient form, given by $K_1 = q$ and $K_2 = 0$, with order q varying between 1 and 5. The coefficients are uniquely determined by the method type, its order, the recent history of the step sizes, and the normalization $\alpha_{n,0} = -1$. See [4] and [17].

For either choice of formula, the nonlinear system

$$G(y^n) \equiv y^n - h_n \beta_{n,0} f(t_n, y^n) - a_n = 0\,, \tag{3.3}$$

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

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

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

$$M[y^{n(m+1)} - y^{n(m)}] = -G(y^{n(m)})\,, \tag{3.4}$$

in which

$$M \approx I - \gamma J\,, \quad J = \partial f / \partial y\,, \quad \text{and} \quad \gamma = h_n \beta_{n,0}\,. \tag{3.5}$$

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

- a dense direct solver (serial version only),
- a band direct solver (serial version only),
- a diagonal approximate Jacobian solver,
- SPGMR = scaled preconditioned GMRES (Generalized Minimal Residual method) without restarts,
- SPBCG = scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method), or
- SPTFQMR = scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method).

For large stiff systems, where direct methods are not feasible, the combination of a BDF integrator and any of the preconditioned Krylov methods (SPGMR, SPBCG, or SPTFQMR) yields a powerful tool because it combines established methods for stiff integration, nonlinear iteration, and Krylov (linear) iteration with a problem-specific treatment of the dominant source of stiffness, in the form of the user-supplied preconditioner matrix [2]. Note that the direct linear solvers (dense and band) can only be used with serial vector representations.

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

$$W_i = 1/[\mathrm{RTOL} \cdot |y_i| + \mathrm{ATOL}_i] \,. \tag{3.6}$$

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

In the cases of a direct solver (dense, band, or diagonal), the iteration is a Modified Newton iteration, in that the iteration matrix M is fixed throughout the nonlinear iterations. However, for any of the Krylov methods, it is an Inexact Newton iteration, in which M is applied in a matrix-free manner, with matrix-vector products Jv obtained by either difference quotients or a user-supplied routine. The matrix M (direct cases) or preconditioner matrix P (Krylov cases) is updated as infrequently as possible to balance the high costs of matrix operations against other costs. Specifically, this matrix update occurs when:

- starting the problem,
- more than 20 steps have been taken since the last update,
- the value $\bar{\gamma}$ of γ at the last update satisfies $|\gamma/\bar{\gamma} - 1| > 0.3$,
- a non-fatal convergence failure just occurred, or
- an error test failure just occurred.

When forced by a convergence failure, an update of M or P may or may not involve a reevaluation of J (in M) or of Jacobian data (in P), depending on whether Jacobian error was the likely cause of the failure. More generally, the decision is made to reevaluate J (or instruct the user to reevaluate Jacobian data in P) when:

- starting the problem,
- more than 50 steps have been taken since the last evaluation,
- a convergence failure occurred with an outdated matrix, and the value $\bar{\gamma}$ of γ at the last update satisfies $|\gamma/\bar{\gamma} - 1| < 0.2$, or
- a convergence failure occurred that forced a step size reduction.

The stopping test for the Newton iteration is related to the subsequent local error test, with the goal of keeping the nonlinear iteration errors from interfering with local error control. As described below, the final computed value $y^{n(m)}$ will have to satisfy a local error test $\|y^{n(m)} - y^{n(0)}\| \leq \epsilon$. Letting y^n denote the exact solution of (3.3), we want to ensure that the iteration error $y^n - y^{n(m)}$ is small relative to ϵ, specifically that it is less than 0.1ϵ. (The safety factor 0.1 can be changed by the user.) For this, we also estimate the linear convergence rate constant R as follows. We initialize R to 1, and reset $R = 1$ when M or P is updated. After computing a correction $\delta_m = y^{n(m)} - y^{n(m-1)}$, we update R if $m > 1$ as

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

Now we use the estimate

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

Therefore the convergence (stopping) test is

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

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

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

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

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

The increments σ_j are given by

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

where U is the unit roundoff, σ_0 is a dimensionless value, and W_j is the error weight defined in (3.6). In the dense case, this scheme requires N evaluations of f, one for each column of J. In the band case, the columns of J are computed in groups, by the Curtis-Powell-Reid algorithm, with the number of f evaluations equal to the bandwidth.

In the case of a Krylov method, preconditioning may be used on the left, on the right, or both, with user-supplied routines for the preconditioning setup and solve operations, and optionally also for the required matrix-vector products Jv. If a routine for Jv is not supplied, these products are computed as

$$Jv = [f(t, y + \sigma v) - f(t, y)]/\sigma\,. \tag{3.7}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and

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

The new order and step size are then set according to

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

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

The various algorithmic features of CVODE described above, as inherited from the solvers VODE and VODPK, are documented in [1, 3, 13]. They are also summarized in [14].

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

3.2 Preconditioning

When using a Newton method to solve the nonlinear system (3.3), CVODE makes repeated use of a linear solver to solve linear systems of the form $Mx = -r$, where x is a correction vector and r is a residual vector. If this linear system solve is done with one of the scaled preconditioned iterative linear solvers, these solvers are rarely successful if used without preconditioning; it is generally necessary to precondition the system in order to obtain acceptable efficiency. A system $Ax = b$ can be preconditioned on the left, as $(P^{-1}A)x = P^{-1}b$; on the right, as $(AP^{-1})Px = b$; or on both sides, as $(P_L^{-1}AP_R^{-1})P_Rx = P_L^{-1}b$. The Krylov method is then applied to a system with the matrix $P^{-1}A$, or

AP^{-1}, or $P_L^{-1}AP_R^{-1}$, instead of A. In order to improve the convergence of the Krylov iteration, the preconditioner matrix P, or the product $P_L P_R$ in the last case, should in some sense approximate the system matrix A. Yet at the same time, in order to be cost-effective, the matrix P, or matrices P_L and P_R, should be reasonably efficient to evaluate and solve. Finding a good point in this tradeoff between rapid convergence and low cost can be very difficult. Good choices are often problem-dependent (for example, see [2] for an extensive study of preconditioners for reaction-transport systems).

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

Typical preconditioners used with CVODE are based on approximations to the Newton iteration matrix of the systems involved; in other words, $P \approx I - \gamma J$, where γ is a scalar proportional to the integration step size h and J is the system Jacobian, i.e. $J = \partial f / \partial y$. Because the Krylov iteration occurs within a Newton iteration and further also within a time integration, and since each of these iterations has its own test for convergence, the preconditioner may use a very crude approximation, as long as it captures the dominant numerical feature(s) of the system. We have found that the combination of a preconditioner with the Newton-Krylov iteration, using even a fairly poor approximation to the Jacobian, can be surprisingly superior to using the same matrix without Krylov acceleration (i.e., a modified Newton iteration), as well as to using the Newton-Krylov method with no preconditioning.

3.3 BDF stability limit detection

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

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

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

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

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

The STALD algorithm attempts to detect, in a direct manner, the presence of a stability region boundary that is limiting the step sizes in the presence of a weakly damped oscillation [11]. The algorithm supplements (but differs greatly from) the existing algorithms in CVODE for choosing step size and order based on estimated local truncation errors. It works directly with history data that is readily available in CVODE. If it concludes that the step size is in fact stability-limited, it dictates a reduction in the method order, regardless of the outcome of the error-based algorithm. The STALD algorithm has been tested in combination with the VODE solver on linear advection-dominated advection-diffusion problems [12], where it works well. The implementation in CVODE has been successfully tested on linear and nonlinear advection-diffusion problems, among others.

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

3.4 Rootfinding

The CVODE solver has been augmented to include a rootfinding feature. This means that, while integrating the Initial Value Problem (3.1), CVODE can also find the roots of a set of user-defined functions $g_i(t, y)$ that depend on t and the solution vector $y = y(t)$. The number of these root functions is arbitrary, and if more than one g_i is found to have a root in any given interval, the various root locations are found and reported in the order that they occur on the t axis, in the direction of integration.

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

The basic scheme used is to check for sign changes of any $g_i(t)$ over each time step taken, and then (when a sign change is found) to home in on the root (or roots) with a modified secant method [10]. In addition, each time g is computed, CVODE checks to see if $g_i(t) = 0$ exactly, and if so it reports this as a root. However, if an exact zero of any g_i is found at a point t, CVODE computes g at $t + \delta$ for a small increment δ, slightly further in the direction of integration, and if any $g_i(t + \delta) = 0$ also, CVODE stops and reports an error. This way, each time CVODE takes a time step, it is guaranteed that the values of all g_i are nonzero at some past value of t, beyond which a search for roots is to be done.

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

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

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

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

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

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

Chapter 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 $F(t, y, y') = 0$.

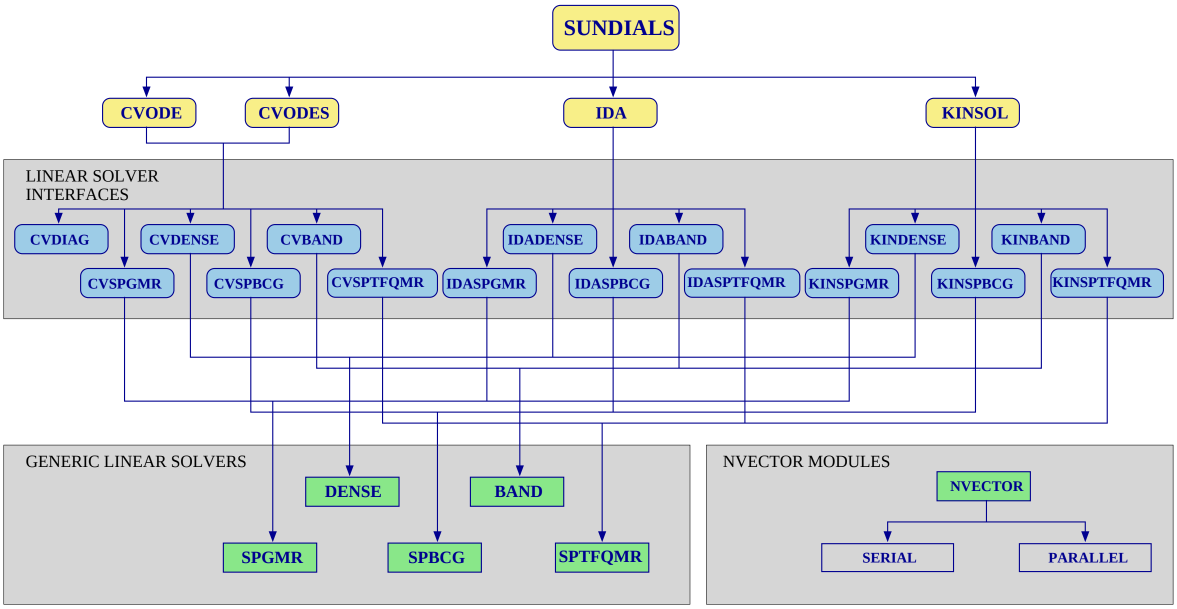
4.2 CVODE organization

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

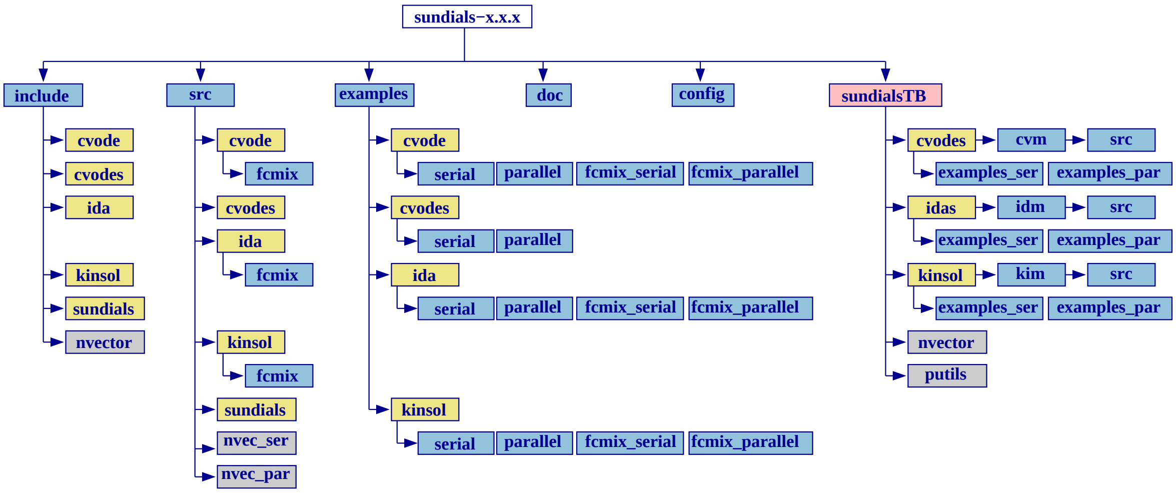
The overall organization of the CVODE package is shown in Figure 4.2. The central integration module, implemented in the files `cvode.h`, `cvode_impl.h`, and `cvode.c`, deals with the evaluation of integration coefficients, the functional or Newton iteration process, estimation of local error, selection of stepsize and order, and interpolation to user output points, among other issues. Although this module contains logic for the basic Newton iteration algorithm, it has no knowledge of the method being used to solve the linear systems that arise. For any given user problem, one of the linear system modules is specified, and is then invoked as needed during the integration.

At present, the package includes the following six CVODE linear system 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;
- CVSPBCG: scaled preconditioned Bi-CGStab method;



(a) High-level diagram



(b) Directory structure of the source tree

Figure 4.1: Organization of the SUNDIALS suite

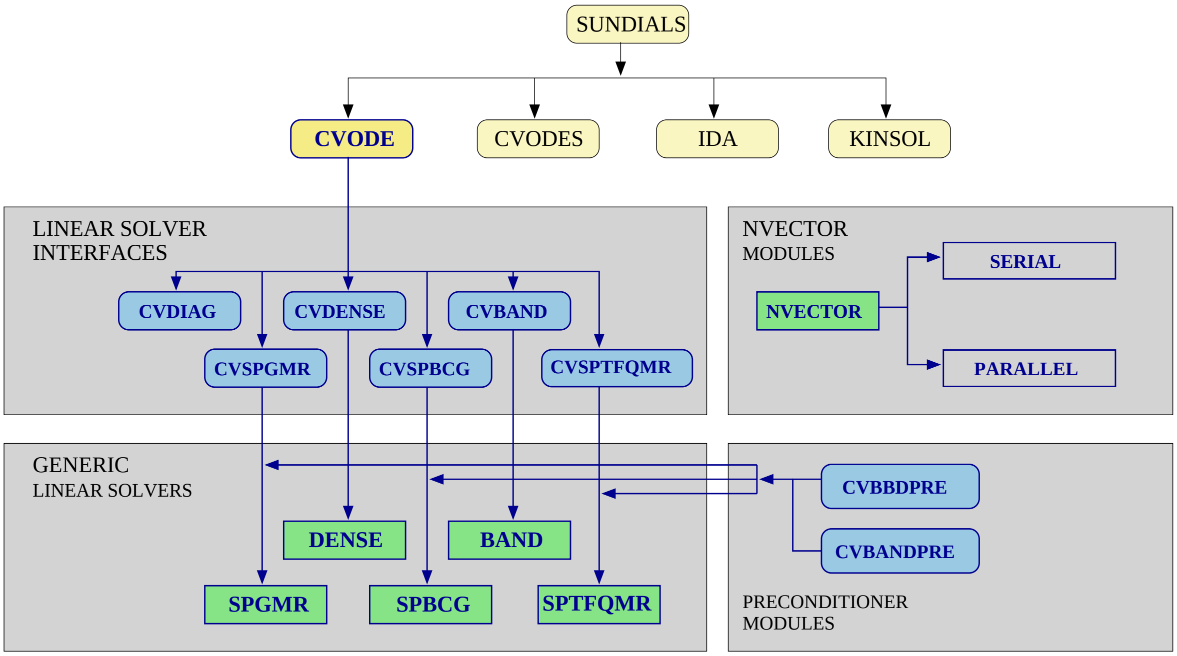


Figure 4.2: Overall structure diagram of the CVODE package. Modules specific to CVODE are distinguished by rounded boxes, while generic solver and auxiliary modules are in rectangular boxes.

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

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

These modules are also decomposed in another way. Each of the modules CVDENSE, CVBAND, CVSPGMR, CVSPBCG, and CVSPTFQMR is a set of interface routines built on top of a generic solver module, named DENSE, BAND, SPGMR, SPBCG, and SPTFQMR, respectively. The interfaces deal with the use of these methods in the CVODE 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 CVODE package elsewhere.

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

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

Chapter 5

Using CVODE for C Applications

This chapter is concerned with the use of CVODE for the integration of IVPs. The following sections treat the header files, the layout of the user's main program, description of the CVODE user-callable functions, and user-supplied functions. The final section describes the FORTRAN/C interface module, which supports users with applications written in FORTRAN77. The listings of the example programs in the companion document [15] may also be helpful. Those codes may be used as templates (with the removal of some lines involved in testing), and are included in the CVODE 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 CVODE modules can only be used with NVECTOR_SERIAL: CVDENSE, CVBAND, and CVBANDPRE. The preconditioner module CVBBDPRE can only be used with NVECTOR_PARALLEL.

CVODE uses various constants for both input and output. These are defined as needed in this chapter, but for convenience are also listed separately in Chapter 10.

5.1 Access to library and header files

At this point, it is assumed that the installation of CVODE, 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 appropriate locations for the library and header files required by CVODE. The relevant library files are

- *libdir*/`libsundials_cvode.`*lib*,
- *libdir*/`libsundials_nvec*.`*lib* (one or two files),

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

- *incdir*/`include`
- *incdir*/`include/cvode`
- *incdir*/`include/sundials`

The directories *libdir* and *incdir* are the install libray and include directories. For a default installation, these are *builddir*/`lib` and *builddir*/`include`, respectively, where *builddir* was defined in Chapter 2.

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

5.2 Data types

The sundials_types.h file contains the definition of the type realtype, which is used by the SUNDIALS solvers for all floating-point data. The type realtype can be float, double, or long double, with the default being double. The user can change the precision of the SUNDIALS solvers arithmetic at the configuration stage (see §2.1).

Additionally, based on the current precision, sundials_types.h defines BIG_REAL to be the largest value representable as a realtype, SMALL_REAL to be the smallest value representable as a realtype, and UNIT_ROUNDOFF to be the difference between 1.0 and the minimum realtype greater than 1.0.

Within SUNDIALS, real constants are set by way of a macro called RCONST. It is this macro that needs the ability to branch on the definition realtype. In ANSI C, a floating-point constant with no suffix is stored as a double. Placing the suffix "F" at the end of a floating point constant makes it a float, whereas using the suffix "L" makes it a long double. For example,

```
#define A 1.0
#define B 1.0F
#define C 1.0L
```

defines A to be a double constant equal to 1.0, B to be a float constant equal to 1.0, and C to be a long double constant equal to 1.0. The macro call RCONST(1.0) automatically expands to 1.0 if realtype is double, to 1.0F if realtype is float, or to 1.0L if realtype is long double. SUNDIALS uses the RCONST macro internally to declare all of its floating-point constants.

A user program which uses the type realtype and the RCONST macro to handle floating-point constants is precision-independent except for any calls to precision-specific standard math library functions. (Our example programs use both realtype and RCONST.) Users can, however, use the type double, float, or long double in their code (assuming 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 §2.1).

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:

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

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

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

- nvector_serial.h, which defines the serial implementation NVECTOR_SERIAL;
- nvector_parallel.h, which defines the parallel MPI implementation, NVECTOR_PARALLEL.

Note that both these files include in turn the header file sundials_nvector.h which defines the abstract N_Vector 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 solver options in CVODE are:

- cvode_dense.h, which is used with the dense direct linear solver in the context of CVODE. This in turn includes a header file (sundials_dense.h) which defines the DenseMat type and corresponding accessor macros;

- `cvode_band.h`, which is used with the band direct linear solver in the context of CVODE. This in turn includes a header file (`sundials_band.h`) which defines the `BandMat` type and corresponding accessor macros;
- `cvode_diag.h`, which is used with a diagonal linear solver in the context of CVODE;
- `cvode_spgmr.h`, which is used with the Krylov solver SPGMR in the context of CVODE;
- `cvode_spbcgs.h`, which is used with the Krylov solver SPBCG in the context of CVODE;
- `cvode_sptfqmr.h`, which is used with the Krylov solver SPTFQMR in the context of CVODE;

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

Other headers may be needed, according as to the choice of preconditioner, etc. In one of the examples in [15], preconditioning is done with a block-diagonal matrix. For this, the header `sundials_smalldense.h` is included.

5.4 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) for the integration of an ODE IVP. Some steps are independent of the NVECTOR implementation used; where this is not the case, usage specifications are given for the two implementations provided with CVODE: 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, aside from the internal use in NVECTOR_PARALLEL. Here `argc` and `argv` are the command line argument counter and array received by `main`.

2. **Set problem dimensions**

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

   [**P**] Set `Nlocal`, the local vector length (the sub-vector length for this processor); `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 functions defined by a particular NVECTOR implementation. If a `realtype` array `ydata` already exists, containing the initial values of y, make the call:

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

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

   Otherwise, make the call:

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

   [**P**] `y0 = N_VNew_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 CVODE object**

   Call `cvode_mem = CVodeCreate(lmm,iter);` to create the CVODE memory block and specify the solution method (linear multistep method and nonlinear solver iteration type). `CVodeCreate` returns a pointer to the CVODE memory structure. See §5.5.1 for details.

5. **Allocate internal memory**

   Call `CVodeMalloc(...);` to provide required problem specifications, allocate internal memory for CVODE, and initialize CVODE. `CVodeMalloc` returns an error flag to indicate success or an illegal argument value. See §5.5.1 for details.

6. **Set optional inputs**

   Call `CVodeSet*` functions to change from their default values any optional inputs that control the behavior of CVODE. See §5.5.5 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 §5.5.3):

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

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

   `ier = CVDiag(...);`

   `ier = CVSpgmr(...);`

   `ier = CVSpbcg(...);`

   `ier = CVSptfqmr(...);`

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 §5.5.5 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 §5.7.1 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 §5.5.4 for details.

11. **Get optional outputs**

    Call `CVodeGet*` and `CV*Get*` functions to obtain optional output. See §5.5.7 and §5.7.1 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**] `N_VDestroy_Serial(y);`

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

13. **Free solver memory**

    `CVodeFree(&cvode_mem);` to free the memory allocated for CVODE.

14. **[P] Finalize MPI**

    Call `MPI_Finalize();` to terminate MPI.

5.5 User-callable functions

This section describes the CVODE functions that are called by the user to set up and solve an IVP. Some of these are required. However, starting with §5.5.5, the functions listed involve optional inputs/outputs or restarting, and those paragraphs can be skipped for a casual use of CVODE. In any case, refer to §5.4 for the correct order of these calls. Calls related to rootfinding are described in §5.7.

5.5.1 CVODE initialization and deallocation functions

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

`CVodeCreate`

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

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

Arguments

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

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

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

Arguments

- `cvode_mem` (`void *`) pointer to the CVODE 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`. Here `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, replacing Eq.(3.6). See `CVodeSetEwtFn` in §5.5.5.

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 CVodeMalloc was successful.
CV_MEM_NULL	The CVODE memory block was not initialized through a previous call to CVodeCreate.
CV_MEM_FAIL	A memory allocation request has failed.
CV_ILL_INPUT	An input argument to CVodeMalloc has an illegal value.

Notes See also §5.5.2 for advice on tolerances.

The tolerance values in reltol and abstol may be changed between calls to CVode (see CVodeSetTolerances in §5.5.5).

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

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

CVodeFree

Call `CVodeFree(&cvode_mem);`

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

Arguments The argument is the address of the pointer to the CVODE memory block returned by CVodeCreate (of type void *).

Return value The function CVodeFree has no return value.

5.5.2 Advice on choice and use of tolerances

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

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

(2) The absolute tolerances abstol (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector y may be so small that pure relative error control is meaningless. For example, if y[i] starts at some nonzero value, but in time decays to zero, then pure relative error control on y[i] makes no sense (and is overly costly) after y[i] is below some noise level. Then abstol (if scalar) or abstol[i] (if a vector) needs to be set to that noise level. If the different components have different noise levels, then abstol should be a vector. See the example cvdenx in the CVODE package, and the discussion of it in the CVODE Examples document [15]. In that problem, the three components vary betwen 0 and 1, and have different noise levels; hence the abstol vector. It is impossible to give any general advice on abstol values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.

(3) Finally, it is important to pick all the tolerance values conservately, because they control the error committed on each individual time step. The final (global) errors are some sort of accumulation of those per-step errors. A good rule of thumb is to reduce the tolerances by a factor of .01 from the actual desired limits on errors. So if you want .01% accuracy (globally), a good choice is reltol = 1.0E-6. But in any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

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

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

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

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

5.5.3 Linear solver specification functions

As previously explained, Newton iteration requires the solution of linear systems of the form (3.4). There are six CVODE linear solvers currently available for this task: CVDENSE, CVBAND, CVDIAG, CVSPGMR, CVSPBCG, and CVSPTFQMR. The first three are direct solvers and their names indicate 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 last three CVODE linear solvers — CVSPGMR, CVSPBCG, and CVSPTFQMR — are Krylov iterative solvers, which use scaled preconditioned GMRES, scaled preconditioned Bi-CGStab, and scaled preconditioned TFQMR, respectively. Together, they are refered to as CVSPILS (from scaled preconditioned iterative linear solvers).

With any of the Krylov methods, preconditioning can be done on the left only, on the right only, on both the left and the right, or not at all. For the specification of a preconditioner, see the iterative linear solver sections in §5.5.5 and §5.6.

If preconditioning is done, user-supplied functions define left and right preconditioner matrices P_1 and P_2 (either of which could be the identity matrix), such that the product $P_1 P_2$ approximates the Newton matrix $M = I - \gamma J$ of (3.5).

To specify a CVODE linear solver, after the call to `CVodeCreate` but before any calls to `CVode`, the user's program must call one of the functions `CVDense`, `CVBand`, `CVDiag`, `CVSpgmr`, `CVSpbcg`, or `CVSptfqmr`, as documented below. The first argument passed to these functions is the CVODE memory pointer returned by `CVodeCreate`. A call to one of these functions links the main CVODE integrator to a linear solver and allows the user to specify parameters which are specific to a particular solver, such as the half-bandwidths in the CVBAND case. The use of each of the linear solvers involves certain constants and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the linear solver, as specified below.

In each case except the diagonal approximation case CVDIAG, the linear solver module used by CVODE is actually built on top of a generic linear system solver, which may be of interest in itself. These generic solvers, denoted DENSE, BAND, SPGMR, SPBCG, and SPTFQMR, are described separately in Chapter 9.

`CVDense`

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

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

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

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

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

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

`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 by SUNDIALS, only NVECTOR_SERIAL is compatible, while NVECTOR_PARALLEL is not.

`CVBand`

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

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

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

Arguments:

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

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

`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 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 by SUNDIALS, only NVECTOR_SERIAL is compatible, while NVECTOR_PARALLEL is not. The half-bandwidths are to be set so that the nonzero locations (i, j) in the banded (approximate) Jacobian satisfy $-\texttt{mlower} \leq j - i \leq \texttt{mupper}$.

`CVDiag`

Call: `flag = CVDiag(cvode_mem);`

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

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

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

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

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

Notes — The CVDIAG solver is the simplest of all the current CVODE linear solvers. The CVDIAG solver uses an approximate diagonal Jacobian formed by way of a difference quotient. The user does *not* have the option to supply 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 `cvode_spgmr.h` header file.

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

Return value — The return value `flag` (of type `int`) is one of
- `CVSPILS_SUCCESS` The CVSPGMR initialization was successful.
- `CVSPILS_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPILS_ILL_INPUT` The preconditioner type `pretype` is not valid.
- `CVSPILS_MEM_FAIL` A memory allocation request failed.

Notes — The CVSPGMR solver uses a scaled preconditioned GMRES iterative method to solve the linear system (3.4).

`CVSpbcg`

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

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

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

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

Return value — The return value `flag` (of type `int`) is one of
- `CVSPILS_SUCCESS` The CVSPBCG initialization was successful.
- `CVSPILS_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPILS_ILL_INPUT` The preconditioner type `pretype` is not valid.
- `CVSPILS_MEM_FAIL` A memory allocation request failed.

Notes — The CVSPBCG solver uses a scaled preconditioned Bi-CGStab iterative method to solve the linear system (3.4).

`CVSptfqmr`

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

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

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

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

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

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

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

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

Notes The CVSPTFQMR solver uses a scaled preconditioned TFQMR iterative method to solve the linear system (3.4).

5.5.4 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 CVODE memory block.

tout (realtype) the next time at which a computed solution is desired.

yout (N_Vector) the computed solution vector.

tret (realtype *) the time reached by the solver.

itask (int) a flag indicating the job of the solver for the next user 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, 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 root was found.
CV_TSTOP_RETURN	CVode succeeded by reaching the stopping point specified via the optional input function CVodeSetStopTime (see §5.5.5).
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 §5.7 for more information.
CV_MEM_NULL	The cvode_mem argument was NULL.
CV_NO_MALLOC	The CVODE memory 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 function initialization (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 error message for details.

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

`CV_TOO_MUCH_WORK` The solver took `mxstep` internal steps but 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` Error test failures occurred too many times (`MXNEF = 7`) during one internal time step or occurred with $|h| = h_{min}$.

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

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

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

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

`CV_RHSFUNC_FAIL` The right-hand side function failed in an unrecoverable manner.

`CV_FIRST_RHSFUNC_FAIL` The right-hand side function had a recoverable error at the first call.

`CV_REPTD_RHSFUNC_ERR` Convergence tests occurred too many times due to repeated recoverable errors in the right-hand side function. The value `CV_REPTD_RHSFUNC_ERR` will also be returned if the right-hand side function had repeated recoverable errors during the estimation of an initial step size.

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

`CV_RTFUNC_FAIL` The rootfinding function failed.

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

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

All failure return values are negative and therefore a test `flag`< 0 will trap all `CVode` failures.

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

5.5.5 Optional input functions

CVODE provides an extensive list of functions that can be used to change from their default values various optional input parameters that control the behavior of the CVODE solver. Table 5.1 lists all optional input functions in CVODE which are then described in detail in the remainder of this section, begining with those for the main CVODE solver and continuing with those for the linear solver modules.

Note that the diagonal linear solver module has no optional inputs. For the most casual use of CVODE, the reader can skip to §5.6.

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

5.5.5.1 Main solver optional input functions

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

`CVodeSetErrHandlerFn`

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

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block.
- `ehfun` (`CVErrHandlerFn`) is the C error handler function (see §5.6.2).
- `eh_data` (`void *`) pointer to user data passed to `ehfun` every time it is called.

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

Notes The default internal error handler function directs error messages to the file specified by the file pointer `errfp` (see `CVodeSetErrFile` below).

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

`CVodeSetErrFile`

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

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

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

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

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

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

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.

`CVodeSetFdata`

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

Description The function `CVodeSetFdata` specifies the user data block `f_data`, for use by the user right-hand side function f, and attaches it to the main CVODE memory block.

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

Table 5.1: Optional inputs for CVODE, CVDENSE, CVBAND, and CVSPILS

Optional input	Function name	Default
CVODE main solver		
Error handler function	`CVodeSetErrHandlerFn`	internal fn.
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`	undefined
Maximum no. of error test failures	`CVodeSetMaxErrTestFails`	7
Maximum no. of nonlinear iterations	`CVodeSetMaxNonlinIters`	3
Maximum no. of convergence failures	`CVodeSetMaxConvFails`	10
Coefficient in the nonlinear convergence test	`CVodeSetNonlinConvCoef`	0.1
Nonlinear iteration type	`CVodeSetIterType`	none
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
CVSPILS linear solvers		
Preconditioner functions and data	`CVSpilsSetPreconditioner`	all NULL
Jacobian-times-vector function and data	`CVSpilsSetJacTimesVecFn`	internal DQ, NULL
Preconditioning type	`CVSpilsSetPrecType`	none
Ratio between linear and nonlinear tolerances	`CVSpilsSetDelt`	0.05
Type of Gram-Schmidt orthogonalization[a]	`CVSpilsSetGSType`	classical GS
Maximum Krylov subspace size[b]	`CVSpilsSetMaxl`	5

[a] Only for CVSPGMR

[b] Only for CVSPBCG and CVSPTFQMR

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 CVODE 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 CVODE memory block, its value can not 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 CVODE memory block.
- `mxsteps` (`long int`) maximum allowed number of steps.

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

- `CV_SUCCESS` The optional value has been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CV_ILL_INPUT` `mxsteps` is non-positive.

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

`CVodeSetMaxHnilWarns`

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

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

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

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 `mxhnil` value indicates that no warning messages should be issued.

`CVodeSetStabLimDet`

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

Description The function `CVodeSetStabLimDet` indicates to turn on/off the BDF stability limit detection algorithm. See §3.3.

Arguments
`cvode_mem` (`void *`) pointer to the CVODE memory block.
`stldet` (`booleantype`) flag to control 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 3 or greater, an internal function, `CVsldet`, is called to detect stability limit. If limit is detected, the order is reduced.

`CVodeSetInitStep`

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

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

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

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

Notes By default, CVODE estimates the initial stepsize as 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 the minimum absolute value of the step size.

Arguments
`cvode_mem` (`void *`) pointer to the CVODE 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 not positive or it is larger than the maximum allowable step.

Notes The default value is 0.0.

`CVodeSetMaxStep`

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

Description The function `CVodeSetMaxStep` specifies the maximum absolute value of the step size.

Arguments
`cvode_mem` (`void *`) pointer to the CVODE 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 not positive or it is smaller than the minimum allowable step.

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 CVODE memory block.
- tstop (realtype) value of the independent variable past which the solution should not proceed.

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

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

Notes: The default value is ∞.

CVodeSetMaxErrTestFails

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

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

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

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 at one step.

Arguments:
- cvode_mem (void *) pointer to the CVODE memory block.
- maxcor (int) maximum number of nonlinear solver iterations 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 3.

`CVodeSetMaxConvFails`

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

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

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`maxncf` (`int`) maximum number of allowable nonlinear solver convergence failures 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 10.

`CVodeSetNonlinConvCoef`

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

Description The function `CVodeSetNonlinConvCoef` specifies the safety factor in the nonlinear convergence test (see §3.1).

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

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`iter` (`int`) specifies the type of nonlinear solver iteration and may be either `CV_NEWTON` or `CV_FUNCTIONAL`.

Return value The return value `flag` (of type `int`) is one of
`CV_SUCCESS` The optional value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_ILL_INPUT` The `iter` value passed is neither `CV_NEWTON` nor `CV_FUNCTIONAL`.

Notes The nonlinear solver iteration type is initially specified in the call to `CVodeCreate` (see §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 CVODE 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) 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) 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 between successive 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, which is normally defined by Eq.(3.6).

Arguments

cvode_mem	(void *) pointer to the CVODE memory block.
efun	(CVEwtFn) is the C function which defines the ewt vector (see §5.6.3).
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.

5.5.5.2 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 §5.6.4.

5.5.5.3 Band linear solver

The CVDENSE 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 §5.6.5.

5.5.5.4 SPILS linear solvers

If any preconditioning is to be done with one of the CVSPILS linear solvers, then the user must supply a preconditioner solve function `psolve` and specify its name in a call to `CVSpilsSetPreconditioner`.

The evaluation and preprocessing of any Jacobian-related data needed by the user's preconditioner solve function is done in the optional user-supplied function `psetup`. Both of these functions are fully specified in §5.6. If used, the `psetup` function should also be specified in the call to `CVSpilsSetPreconditioner`. Optionally, a CVSPILS solver passes the pointer `p_data` received through `CVSpilsSetPreconditioner` to the preconditioner `psetup` and `psolve` functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program. The pointer `p_data` may be identical to `f_data`, if the latter was specified through `CVodeSetFdata`.

Ther CVSPILS solvers require a function to compute an approximation to the product between the Jacobian matrix $J(t, y)$ and a vector v. The user can supply his/her own Jacobian-times-vector approximation function, or use the difference quotient function `CVSpilsDQJtimes` that comes with the CVSPILS solvers. A user-defined Jacobian-vector function must be of type `CVSpilsJacTimesVecFn` and can be specified through a call to `CVSpilsSetJacTimesVecFn` (see §5.6.6 for specification details). As with the preconditioner user data structure `p_data`, the user can also specify, in the call to `CVSpilsSetJacTimesVecFn`, a pointer to a user-defined data structure, `jac_data`, which the CVSPILS 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`.

`CVSpilsSetPreconditioner`

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

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

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

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

Notes The function type `CVSpilsPrecSolveFn` is described in §5.6.7. The function type `CVSpilsPrecSetupFn` is described in §5.6.8.

`CVSpilsSetJacTimesVecFn`

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

Description The function `CVSpilsSetJacTimesFn` 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` (`CVSpilsJacTimesVecFn`) 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
- `CVSPILS_SUCCESS` The optional value has been successfully set.
- `CVSPILS_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPILS_LMEM_NULL` The CVSPILS linear solver has not been initialized.

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

The function type `CVSpilsJacTimesVecFn` is described in §5.6.6.

`CVSpilsSetPrecType`

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

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

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

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

`CVSPILS_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVSPILS_LMEM_NULL`	The CVSPILS linear solver has not been initialized.
`CVSPILS_ILL_INPUT`	The preconditioner type `pretype` is not valid.

Notes: The preconditioning type is initially set in the call to the linear solver's specification function (see §5.5.3). This function call is needed only if `pretype` is being changed from its original value.

`CVSpilsSetGSType`

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

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

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

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

`CVSPILS_SUCCESS`	The optional value has been successfully set.
`CVSPILS_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVSPILS_LMEM_NULL`	The CVSPILS linear solver has not been initialized.
`CVSPILS_ILL_INPUT`	The Gram-Schmidt orthogonalization type `gstype` is not valid.

Notes: The default value is `MODIFIED_GS`.

This option is available only for the CVSPGMR linear solver.

`CVSpilsSetDelt`

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

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

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

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

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

Notes: The default value is 0.05.

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

`CVSpilsSetMaxl`

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

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

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

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

CVSPILS_SUCCESS	The optional value has been successfuly set.
CVSPILS_MEM_NULL	The cv_mem pointer is NULL.
CVSPILS_LMEM_NULL	The CVSPBCG linear solver has not been initialized.

Notes The maximum subspace dimension is initially specified in the call to the linear solver specification function(see §5.5.3). This function call is needed only if maxl is being changed from its previous value.

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

5.5.6 Interpolated output function

An optional function CVodeGetDky is available to obtain additional output values. This function must be called after a successful return from CVode and provides interpolated values of y or its derivatives, up to the current order of the integration method, interpolated to any value of t in the last internal step taken by CVODE.

The call to the CVodeGetDky function has the following form:

CVodeGetDky

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

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

Arguments

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

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.

5.5.7 Optional output functions

CVODE provides an extensive list of functions that can be used to obtain solver performance information. Table 5.2 lists all optional output functions in CVODE, which are then described in detail in the remainder of this section, begining with those for the main CVODE solver and continuing with those for the linear solver modules. Where the name of an output from a linear solver module would otherwise conflict with the name of an optional output from the main solver, a suffix LS (for Linear Solver) has been added here (e.g. lenrwLS).

Table 5.2: Optional outputs from CVODE, CVDENSE, CVBAND, CVDIAG, and CVSPILS

Optional output	Function name
CVODE main solver	
Size of CVODE real and integer workspaces	`CVodeGetWorkSpace`
Cumulative number of internal steps	`CVodeGetNumSteps`
No. of calls to r.h.s. function	`CVodeGetNumRhsEvals`
No. of calls to linear solver setup function	`CVodeGetNumLinSolvSetups`
No. of local error test failures that have occurred	`CVodeGetNumErrTestFails`
Order used during the last step	`CVodeGetLastOrder`
Order to be attempted on the next step	`CVodeGetCurrentOrder`
Order reductions due to stability limit detection	`CVodeGetNumStabLimOrderReds`
Actual initial step size used	`CVodeGetActualInitStep`
Step size used for the last step	`CVodeGetLastStep`
Step size to be attempted on the next step	`CVodeGetCurrentStep`
Current internal time reached by the solver	`CVodeGetCurrentTime`
Suggested factor for tolerance scaling	`CVodeGetTolScaleFactor`
Error weight vector for state variables	`CVodeGetErrWeights`
Estimated local error vector	`CVodeGetEstLocalErrors`
No. of nonlinear solver iterations	`CVodeGetNumNonlinSolvIters`
No. of nonlinear convergence failures	`CVodeGetNumNonlinSolvConvFails`
All CVODE integrator statistics	`CVodeGetIntegratorStats`
CVODE nonlinear solver statistics	`CVodeGetNonlinSolvStats`
Array showing roots found	`CvodeGetRootInfo`
No. of calls to user root function	`CVodeGetNumGEvals`
Name of constant associated with a return flag	`CVodeGetReturnFlagName`
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`
Name of constant associated with a return flag	`CVDenseGetReturnFlagName`
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`
Name of constant associated with a return flag	`CVBandGetReturnFlagName`
CVDIAG linear solver	
Size of CVDIAG real and integer workspaces	`CVDiagGetWorkSpace`
No. of r.h.s. calls for finite diff. Jacobian evals.	`CVDiagGetNumRhsEvals`
Last return from a CVDIAG function	`CVDiagGetLastFlag`
Name of constant associated with a return flag	`CVDiagGetReturnFlagName`
CVSPILS linear solvers	
Size of real and integer workspaces	`CVSpilsGetWorkSpace`
No. of linear iterations	`CVSpilsGetNumLinIters`
No. of linear convergence failures	`CVSpilsGetNumConvFails`
No. of preconditioner evaluations	`CVSpilsGetNumPrecEvals`
No. of preconditioner solves	`CVSpilsGetNumPrecSolves`
No. of Jacobian-vector product evaluations	`CVSpilsGetNumJtimesEvals`
No. of r.h.s. calls for finite diff. Jacobian-vector evals.	`CVSpilsGetNumRhsEvals`
Last return from a linear solver function	`CVSpilsGetLastFlag`
Name of constant associated with a return flag	`CVSpilsGetReturnFlagName`

5.5.7.1 Main solver optional output functions

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

`CVodeGetWorkSpace`

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

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

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

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 In terms of the problem size N, the maximum method order `maxord`, and the number `nrtfn` of root functions (see §5.7), the actual size of the real workspace, in `realtype` words, is given by the following:

- base value: `lenrw` $= 89 + ($`maxord+5`$) * N_r + 3*$`nrtfn`;
- if `itol = CV_SV`: `lenrw` = `lenrw` $+N_r$;

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

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

- base value: `leniw` $= 40 + ($`maxord+5`$) * N_i\ +$ `nrtfn`;
- if `itol = CV_SV`: `leniw` = `leniw` $+N_i$;

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

For the default value of `maxord`, with no rootfinding, and with `itol` $\neq$ `CV_SV`, these lengths are given roughly by:

- For the Adams method: `lenrw` $= 89 + 17N$ and `leniw` $= 57$
- For the BDF method: `lenrw` $= 89 + 10N$ and `leniw` $= 50$

`CVodeGetNumSteps`

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

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

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

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

`CVodeGetNumRhsEvals`

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

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

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

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

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

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

`CVodeGetNumLinSolvSetups`

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

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

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

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

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

`CVodeGetNumErrTestFails`

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

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

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

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

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

`CVodeGetLastOrder`

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

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

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

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

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

CVodeGetCurrentOrder

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

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

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

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

CVodeGetLastStep

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

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

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

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

CVodeGetCurrentStep

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

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

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

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

CVodeGetActualInitStep

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

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

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

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

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

`CVodeGetCurrentTime`

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

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

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

`tcur` (`realtype`) current internal time reached.

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

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

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

`CVodeGetNumStabLimOrderReds`

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

Description The function `CVodeGetNumStabLimOrderReds` returns the number of order reductions dictated by the BDF stability limit detection algorithm (see §3.3).

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

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

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

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

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

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

`CVodeGetTolScaleFactor`

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

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

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

`tolsfac` (`realtype`) suggested scaling factor for user 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 normally the W_i of (3.6).

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

`eweight` (`N_Vector`) solution error weights at the current time.

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

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

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

Notes The user must allocate memory for `eweight`.

`CVodeGetEstLocalErrors`

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

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

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

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

Notes The user must allocate memory for `ele`.

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

`CVodeGetIntegratorStats`

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

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

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

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

`CVodeGetNumNonlinSolvIters`

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

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

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

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

`CVodeGetNumNonlinSolvConvFails`

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

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

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`nncfails` (`long int`) number of nonlinear convergence failures.

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

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

`CVodeGetNonlinSolvStats`

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

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

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

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

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

`CVodeGetReturnFlagName`

Call `name = CVodeGetReturnFlagName(flag);`

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

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

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

5.5.7.2 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, &lenrwLS, &leniwLS);`

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

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`lenrwLS` (`long int`) the number of `realtype` values in the CVDENSE workspace.
`leniwLS` (`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 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: 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, &njevals);`

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

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

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

- `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, &nfevalsLS);`

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

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

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

- `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 `nfevalsLS` 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 CVODE 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 `lsflag` is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the dense Jacobian matrix.

`CVDenseGetReturnFlagName`

Call `name = CVDenseGetReturnFlagName(flag);`

Description The function `CVDenseGetReturnFlagName` returns the name of the CVDENSE constant corresponding to `flag`.

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

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

5.5.7.3 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, &lenrwLS, &leniwLS);`

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

Arguments
`cvode_mem` (`void *`) pointer to the CVODE memory block.
`lenrwLS` (`long int`) the number of `realtype` values in the CVBAND workspace.
`leniwLS` (`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 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 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, &njevals);`

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

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

Return value The return value `flag` (of type `int`) is one of
`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, &nfevalsLS);`

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

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

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

`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 `nfevalsLS` 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 last return value from a CVBAND routine.

Arguments `cvode_mem` (`void *`) pointer to the CVODE 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 sunction failed (`CVode` returned `CV_LSETUP_FAIL`), the value `lsflag` is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the banded Jacobian matrix.

`CVBandGetReturnFlagName`

Call `name = CVBandGetReturnFlagName(flag);`

Description The function `CVBandGetReturnFlagName` returns the name of the CVBAND constant corresponding to `flag`.

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

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

5.5.7.4 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, &lenrwLS, &leniwLS);`

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

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

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

`CVDIAG_SUCCESS`	The optional output 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 In terms of the problem size N, the actual size of the real workspace is roughly $3N$ `realtype` words.

`CVDiagGetNumRhsEvals`

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

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

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

`nfevalsLS` (`long int`) the number of calls to the user 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 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 CVODE 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 `lsflag` is equal to `CVDIAG_INV_FAIL`, indicating that a zero diagonal element was encountered. The same value for `lsflag` is set if the CVDIAG solve function failed (`CVode` returned `CV_LSOLVE_FAIL`).

`CVDiagGetReturnFlagName`

Call `name = CVDiagGetReturnFlagName(flag);`

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

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

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

5.5.7.5 SPILS linear solvers

The following optional outputs are available from the CVSPILS modules: workspace requirements, number of linear iterations, number of linear convergence failures, number of calls to the preconditioner setup and solve routines, number of calls to the Jacobian-vector product routine, number of calls to the right-hand side routine for finite-difference Jacobian-vector product approximation, and last return value from a linear solver function.

`CVSpilsGetWorkSpace`

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

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

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

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

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

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

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

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

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

Notes In terms of the problem size N and maximum subspace size `maxl`, the actual size of the real workspace is roughly:
$(\texttt{maxl}+5) * N + \texttt{maxl} * (\texttt{maxl}+4) + 1$ `realtype` words for CVSPGMR,
$9 * N$ `realtype` words for CVSPBCG,
and $11 * N$ `realtype` words for IDASPTFQMR.

In a parallel setting, the above values are global — summed over all processors.

`CVSpilsGetNumLinIters`

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

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

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

`nliters` (`long int`) the current number of linear iterations.

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

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

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

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

`CVSpilsGetNumConvFails`

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

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

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

`nlcfails` (`long int`) the current number of linear convergence failures.

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

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

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

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

`CVSpilsGetNumPrecEvals`

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

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

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

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

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

`CVSpilsGetNumPrecSolves`

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

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

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

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

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

`CVSpilsGetNumJtimesEvals`

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

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

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

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

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

`CVSpilsGetNumRhsEvals`

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

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

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

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

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

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

`CVSpilsGetLastFlag`

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

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

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

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

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

Notes If the CVSPILS setup function failed (`CVode` returned `CV_LSETUP_FAIL`), `lsflag` will be `SPGMR_PSET_FAIL_UNREC`, `SPBCG_PSET_FAIL_UNREC`, or `SPTFQMR_PSET_FAIL_UNREC`.

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

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

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

`CVSpilsGetReturnFlagName`

Call `name = CVSpilsGetReturnFlagName(flag);`

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

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

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

5.5.8 CVODE reinitialization function

The function `CVodeReInit` reinitializes the main CVODE 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, assuming that the existing internal memory is sufficient for the new problem.

The use of `CVodeReInit` requires that the maximum method order, `maxord`, is no larger for the new problem than for the problem specified in the last call to `CVodeMalloc`. 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 `Set` calls, as described in §5.5.3

CVodeReInit

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

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE 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=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 (3.6). See `CVodeSetEwtFn` in §5.5.5.
- `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 CVODE memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_NO_MALLOC` Memory space for the CVODE memory block was not allocated through a previous call to `CVodeMalloc`.
- `CV_ILL_INPUT` An input argument to `CVodeReInit` has an illegal value.

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

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 handles error and warning messages, (optionally) a function that provides the error weight vector, (optionally) a function that provides Jacobian-related information for the linear solver (if Newton iteration is chosen), and (optionally) one or two functions that define the preconditioner for use in the any of the Krylov iterative algorithms.

5.6.1 ODE right-hand side

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

CVRhsFn

Definition
```
typedef int (*CVRhsFn)(realtype t, N_Vector y, N_Vector ydot,
                       void *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` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CV_RHSFUNC_FAIL` is returned).

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

!

For efficiency considerations, the right-hand side function is not evaluated at the converged solution of the nonlinear solver. Therefore, a recoverable error in `CVRhsFn` at that point cannot be corrected (as it will occur when the right-hand side function is called the first time during the following integration step and a successful step cannot be undone).

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

5.6.2 Error message handler function

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

`CVErrHandlerFn`

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

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

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

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

Notes `error_code` is negative for errors and positive (`CV_WARNING`) for warnings. If a function returning a pointer to memory (e.g. `CVBBDPrecAlloc`) encounters an error, it sets `error_code` to 0 before returning `NULL`.

5.6.3 Error weight function

As an alternative to providing the relative and absolute tolerances, the user may provide a function of type `CVEwtFn` to compute a vector `ewt` containing the weights in the WRMS norm $\| v\|_{\mathrm{WRMS}} = \sqrt{(1/N)\sum_1^N (W_i \cdot v_i)^2}$. 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 CVODE.

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

5.6.4 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 int (*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 current value of the vector $f(t, y)$.
- `jac_data` is a pointer to user data — the same as the `jac_data` parameter passed to `CVDenseSetJacFn`.
- `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` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct, while CVDENSE sets `last_flag` on `CVDENSE_JACFUNC_RECVR`), or a negative value if it failed unrecoverably (in which case the integration is halted, `CVode` returns `CV_LSETUP_FAIL` and CVDENSE sets `last_flag` on `CVDENSE_JACFUNC_UNRECVR`).

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 for use in small problems in which efficiency of access is not a major concern. Thus, in terms of indices m and n running from 1 to

N, the Jacobian element $J_{m,n}$ can be loaded with the statement `DENSE_ELEM(J, m-1, n-1) =` $J_{m,n}$. Alternatively, `DENSE_COL(J, j)` returns a pointer to the storage for the `j`th column of `J` (`j`$= 0 \ldots N-1$), and the elements of the `j`th column are then accessed via ordinary array indexing. Thus $J_{m,n}$ can be loaded with 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, not 1.

The `DenseMat` type and the accessor macros `DENSE_ELEM` and `DENSE_COL` are documented in §9.1.

If the user's `CVDenseJacFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, use the `CVodeGet*` functions described in §5.5.7.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

5.6.5 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 int (*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 current value of the vector $f(t, y)$.
- `jac_data` is a pointer to user data — the same as the `jac_data` parameter passed to `CVBandSetJacFn`.
- `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` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct, while CVBAND sets `last_flag` on `CVBAND_JACFUNC_RECVR`), or a negative value if it failed unrecoverably (in which case the integration is halted, `CVode` returns `CV_LSETUP_FAIL` and CVBAND sets `last_flag` on `CVBAND_JACFUNC_UNRECVR`).

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 preset to zero 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 for use in small problems in which efficiency of access is not a major concern. Thus, in terms of indices m and n running from 1 to N with (m, n) within the band defined by mupper and mlower, the Jacobian element $J_{m,n}$ can be loaded with 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 jth column of J, and if we assign this address to realtype *col_j, then the ith element of the jth 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 jth 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 the combination of BAND_COL and BAND_COL_ELEM than to use the BAND_ELEM. As in the dense case, these macros all number rows and columns starting from 0, not 1.

The BandMat type and the accessor macros BAND_ELEM, BAND_COL, and BAND_COL_ELEM are documented in §9.2.

If the user's CVBandJacFn function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, use the CVodeGet* functions described in §5.5.7.1. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

5.6.6 Jacobian information (matrix-vector product)

If one of the Krylov iterative linear solvers SPGMR, SPBCG, or SPTFQMR is selected (CVSp* is called in step 7 of §5.4), the user may provide a function of type CVSpilsJacTimesVecFn in the following form:

CVSpilsJacTimesVecFn

Definition

```
typedef int (*CVSpilsJacTimesVecFn)(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 to the right.
- Jv is the output vector computed.
- t is the current value of the independent variable.
- y is the current value of the dependent variable vector.
- fy is the current value of the vector $f(t, y)$.
- jac_data is a pointer to user data — the same as the jac_data parameter passed to CVSp*SetJacTimesVecFn.
- 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-times-vector function should be 0 if successful. Any other return value will result in an unrecoverable error of the generic Krylov solver, in which case the integration is halted.

Notes If the user's CVSpilsJacTimesVecFn function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, use the CVodeGet* functions described in §5.5.7.1. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

5.6.7 Preconditioning (linear system solution)

If preconditioning is used, then the user must provide a C function to solve the linear system $Pz = r$ where P may be either a left or a right preconditioner matrix. This function must be of type CVSpilsPrecSolveFn, defined as follows:

CVSpilsPrecSolveFn

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

Purpose: This function solves the preconditioning system $Pz = r$.

Arguments:
- t is the current value of the independent variable.
- y is the current value of the dependent variable vector.
- fy is the current value of the vector $f(t, y)$.
- r is the right-hand side vector of the linear system.
- z is the output vector computed.
- gamma is the scalar γ appearing in the Newton matrix $M = I - \gamma J$.
- 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 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 §5.5.7.1).
- lr is an input flag indicating whether the preconditioner solve function is to use the left preconditioner (lr = 1) or the right preconditioner (lr = 2);
- p_data is a pointer to user data — the same as the p_data parameter passed to the function CVSp*SetPreconditioner.
- 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), negative for an unrecoverable error (in which case the integration is halted).

5.6.8 Preconditioning (Jacobian data)

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

CVSpilsPrecSetupFn

Definition
```
typedef int (*CVSpilsPrecSetupFn)(realtype t, N_Vector y, N_Vector fy,
                                booleantype jok, booleantype *jcurPtr,
                                realtype gamma, void *p_data,
                                N_Vector tmp1, N_Vector tmp2,
                                N_Vector tmp3);
```

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

Arguments: The arguments of a CVSpilsPrecSetupFn are as follows:

`t`	is the current value of the independent variable.
`y`	is the current value of the dependent variable vector, namely the predicted value of $y(t)$.
`fy`	is the current value of the vector $f(t, y)$.
`jok`	is an input flag indicating whether Jacobian-related data needs to be recomputed. The `jok` argument provides for the re-use of Jacobian data in the preconditioner solve function. `jok == FALSE` means that Jacobian-related data must be recomputed from scratch. `jok == TRUE` means that 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 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 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 that passed to `CVSp*SetPreconditioner`.
`tmp1`	
`tmp2`	
`tmp3`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `CVSpilsPrecSetupFn` as temporary storage or work space.

Return value The value to be returned by the preconditioner setup function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), 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 on the resulting approximation to $M = I - \gamma J$.

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

This function is not called in advance of every call to the preconditioner solve function, but rather is called only as often as needed to achieve convergence in the Newton iteration.

If the user's `CVSpilsPrecSetupFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, use the `CVodeGet*` functions described in §5.5.7.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

5.7 Rootfinding

While integrating the IVP, CVODE has the capability of finding 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 obtain solution information, and it also describes the required additional user-supplied function.

5.7.1 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 §5.7.2 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_ILL_INPUT` The function `g` is `NULL`, but `nrtfn`> 0.

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

There are two optional output functions associated with rootfinding.

`CVodeGetRootInfo`

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

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block.
- `rootsfound` (`int *`) array of length `nrtfn` with the indices of the user functions g_i found to have a root. For $i = 0, \ldots,$`nrtfn` -1, `rootsfound`$[i]= 1$ 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 to the user root function g.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block.
- `ngevals` (`long int`) number of calls to the user's function `g` so 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`.

5.7.2 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 int (*CVRootFn)(realtype t, N_Vector y, realtype *gout, void *g_data);`

Purpose This function computes a vector-valued function $g(t, y)$ such that the roots of the `nrtfn` components $g_i(t, y)$ are to be found during the integration.

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 `CVodeRootInit`.

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

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

5.8 Preconditioner modules

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

5.8.1 A serial banded preconditioner module

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

In order to use the CVBANDPRE module, the user need not define any additional functions. Besides 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 `cvode_bandpre.h` which declares the needed function prototypes. The following is a summary of the usage of this module and describes the sequence of calls in the user main program. Steps that are unchanged from the user main program presented in §5.4 are grayed-out.

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

   Specify the upper and lower half-bandwidths `mu` and `ml` and call

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

   to allocate memory for and initialize a data structure `bp_data` (of type `void *`) to be passed to the appropriate CVSPILS linear solver.

7. **Attach the Krylov linear solver, one of:**

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

   Each function `CVBPSp*` is a wrapper around the corresponding specification function `CVSp*` and performs the following actions:

   - Attaches the CVSPILS linear solver to the main CVODE 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 of `CVBPSp*` is the pointer to the CVBANDPRE data 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 CVSPILS 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 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 CVODE memory block.
- `N` (`long int`) problem dimension.
- `mu` (`long int`) upper half-bandwidth of the problem Jacobian approximation.
- `ml` (`long int`) lower half-bandwidth of the problem 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 its nonzeros only in locations (i, j) with $-\texttt{ml} \le j - i \le \texttt{mu}$.

`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 CVODE memory block.

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

pretype (int) preconditioning type. Must be one of PREC_LEFT or PREC_RIGHT.

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

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

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

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

CVBPSpbcg

Call `flag = CVBPSpbcg(cvode_mem, pretype, maxl, bp_data);`

Description The function CVBPSpbcg links the CVBANDPRE data to the CVSPBCG linear solver and attaches the latter to the CVODE memory block.

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

pretype (int) preconditioning type. Must be one of PREC_LEFT or PREC_RIGHT.

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

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

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

CVSPILS_SUCCESS	The CVSPBCG initialization was successful.
CVSPILS_MEM_NULL	The cvode_mem pointer is NULL.
CVSPILS_ILL_INPUT	The preconditioner type pretype is not valid.
CVSPILS_MEM_FAIL	A memory allocation request failed.
CVBANDPRE_PDATA_NULL	The CVBANDPRE preconditioner has not been initialized.

CVBPSptfqmr

Call `flag = CVBPSptfqmr(cvode_mem, pretype, maxl, bp_data);`

Description The function CVBPSptfqmr links the CVBANDPRE data to the CVSPTFQMR linear solver and attaches the latter to the CVODE memory block.

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

pretype (int) preconditioning type. Must be one of PREC_LEFT or PREC_RIGHT.

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

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

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

CVSPILS_SUCCESS	The CVSPTFQMR initialization was successful.
CVSPILS_MEM_NULL	The cvode_mem pointer is NULL.
CVSPILS_ILL_INPUT	The preconditioner type pretype is not valid.
CVSPILS_MEM_FAIL	A memory allocation request failed.
CVBANDPRE_PDATA_NULL	The CVBANDPRE preconditioner has not been initialized.

`CVBandPrecFree`

Call `CVBandPrecFree(&bp_data);`

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

Arguments The only argument of `CVBandPrecFree` is the address of 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 sizes of the CVBANDPRE real and integer workspaces.

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
- `CVBANDPRE_SUCCESS` The optional output value has been successfully set.
- `CVBANDPRE_PDATA_NULL` The CVBANDPRE preconditioner has not been initialized.

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

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

`CVBandPrecGetNumRhsEvals`

Call `flag = CVBandPrecGetNumRhsEvals(bp_data, &nfevalsBP);`

Description The function `CVBandPrecGetNumRhsEvals` returns the number of calls to the user right-hand side function for finite difference banded Jacobian approximation used within CVBANDPRE's 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.
- `CVBANDPRE_PDATA_NULL` The CVBANDPRE preconditioner has not been initialized.

Notes The counter `nfevalsBP` is distinct from the counter `nfevalsLS` returned by the corresponding `CVSp***GetNumRhsEvals` function, and also from `nfevals`, returned by `CVodeGetNumRhsEvals`. The total number of right-hand side function evaluations is the sum of all three of these counters.

`CVBandPrecGetReturnFlagName`

Call `name = CVBandPrecGetReturnFlagName(flag);`

Description The function `CVBandPrecGetReturnFlagName` returns the name of the CVBANDPRE constant corresponding to `flag`.

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

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

5.8.2 A parallel band-block-diagonal preconditioner module

A principal reason for using a parallel ODE solver such as CVODE lies in the solution of partial differential equations (PDEs). Moreover, the use of a Krylov iterative method for the solution of many such problems is motivated by the nature of the underlying linear system of equations (3.4) that must be solved at each time step. The linear algebraic system is large, sparse, and structured. However, if a Krylov iterative method is to be effective in this setting, then a nontrivial preconditioner needs to be used. Otherwise, the rate of convergence of the Krylov iterative method is usually unacceptably slow. Unfortunately, an effective preconditioner tends to be problem-specific.

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used for several realistic, large-scale problems [16] and is included in a software module within the CVODE package. This module works with the parallel vector module NVECTOR_PARALLEL and is usable with any of the Krylov iterative linear solvers. It generates a preconditioner that is a block-diagonal matrix with each block being a band matrix. The blocks need not have the same number of super- and sub-diagonals and these numbers may vary from block to block. This Band-Block-Diagonal Preconditioner module is called CVBBDPRE.

One way to envision these preconditioners is to think of the domain of the computational PDE problem as being subdivided into M non-overlapping subdomains. Each of these subdomains is then assigned to one of the M processors to be used to solve the ODE system. The basic idea is to isolate the preconditioning so that it is local to each processor, 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 on y_m and also 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. Such an efficiency gain may occur if the couplings in the ODE system outside a certain bandwidth are considerably weaker than those within the band. Reducing `mu` and `ml` while keeping `mudq` and `mldq` at their true values, discards the elements outside the narrower band. Reducing both pairs has the additional effect of lumping the outer Jacobian elements into the computed elements within the band, and requires more caution and experimentation.

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.

Similar block-diagonal preconditioners could be considered with different treatment of the blocks P_m. For example, incomplete LU factorization or an iterative method could be used instead of banded LU factorization.

The CVBBDPRE module calls two user-provided functions to construct P: a required function `gloc` (of type `CVLocalFn`) which approximates the right-hand side function $g(t, y) \approx f(t, y)$ and which is computed locally, and an optional function `cfn` (of type `CVCommFn`) which performs all inter-process 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` as that passed by the user to `CVodeSetFdata` and 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 by `cfn` from the other processors, and that are then used by `gloc`, which is not expected to do any communication.

CVLocalFn

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

Notes This function assumes that all inter-processor communication of data needed to calculate `glocal` has already been done, and 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 inter-processor communications 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` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CVode` returns `CV_LSETUP_FAIL`).

Notes The `cfn` function is expected to save communicated data in space defined within the 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 communications 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 `cvode_bbdpre.h` which declares the needed function prototypes.

The following is a summary of the usage of this module and describes the sequence of calls in the user main program. Steps that are unchanged from the user main program presented in §5.4 are grayed-out.

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

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

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

   to allocate memory for and initialize a data structure `bbd_data` (of type `void *`) to be passed to the Krylov linear solver selected (in the next step). The last two arguments of `CVBBDPrecAlloc` are the two user-supplied functions described above.

8. **Attach the Krylov linear solver, one of:**

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

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

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

   The function `CVBPSp*` is a wrapper around the corresponding specification function `CVSp*` and performs the following actions:

   - Attaches the CVSPILS linear solver to the main CVODE 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 of `CVBBDSp*` is the pointer to the CVBBDPRE data 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 CVSPILS 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 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 CVODE memory block.
- `local_N` (`long int`) local vector length.
- `mudq` (`long int`) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mldq` (`long int`) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mukeep` (`long int`) upper half-bandwidth of the retained banded approximate Jacobian block.
- `mlkeep` (`long int`) lower half-bandwidth of the retained banded approximate Jacobian block.
- `dqrely` (`realtype`) the relative increment in components of `y` used in the difference quotient approximations. The default is `dqrely`= $\sqrt{\text{unit roundoff}}$, which can be specified by passing `dqrely`= 0.0.
- `gloc` (`CVLocalFn`) the C function which computes the approximation $g(t, y) \approx f(t, y)$.
- `cfn` (`CVCommFn`) the optional C function which performs all inter-process 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 computation costs further.

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

`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 CVODE memory block.

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

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

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

CVBBDSpbcg

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

Description The function CVBBDSpbcg links the CVBBDPRE data to the CVSPBCG linear solver and attaches the latter to the CVODE memory block.

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

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

CVSPILS_SUCCESS	The CVSPBCG initialization was successful.
CVSPILS_MEM_NULL	The cvode_mem pointer is NULL.
CVSPILS_ILL_INPUT	The preconditioner type pretype is not valid.
CVSPILS_MEM_FAIL	A memory allocation request failed.
CVBBDPRE_PDATA_NULL	The CVBBDPRE preconditioner has not been initialized.

CVBBDSptfqmr

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

Description The function CVBBDSptfqmr links the CVBBDPRE data to the CVSPTFQMR linear solver and attaches the latter to the CVODE memory block.

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

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

CVSPILS_SUCCESS	The CVSPTFQMR initialization was successful.
CVSPILS_MEM_NULL	The cvode_mem pointer is NULL.
CVSPILS_ILL_INPUT	The preconditioner type pretype is not valid.
CVSPILS_MEM_FAIL	A memory allocation request failed.
CVBBDPRE_PDATA_NULL	The CVBBDPRE preconditioner has not been initialized.

CVBBDPrecFree

Call `CVBBDPrecFree(&bbd_data);`

Description The function CVBBDPrecFree frees the memory allocated by CVBBDPrecAlloc.

Arguments The only argument of CVBBDPrecFree is the address of 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 the same linear solver choice, provided there is no change in `local_N`, `mukeep`, or `mlkeep`. After solving one problem, and after calling `CVodeReInit` to re-initialize CVODE for a subsequent problem, a call to `CVBBDPrecReInit` can be made to change any of the following: the half-bandwidths `mudq` and `mldq` used in the difference-quotient Jacobian approximations, the relative increment `dqrely`, or one of the user-supplied functions `gloc` and `cfn`. If there is a change in any of the linear solver inputs, an additional call to `CVSpgmr`, `CVSpbcg`, or `CVSptfqmr`, and/or one or more of the corresponding `CVSp***Set***` functions, must also be made.

`CVBBDPrecReInit`

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

Description The function `CVBBDPrecReInit` reinitializes 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 inter-process communication required for the computation of $g(t, y)$.

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

- `CVBBDPRE_SUCCESS` The reinitialization was successful.
- `CVBBDPRE_PDATA_NULL` The `bbd_data` argument is `NULL`.

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

- `CVBBDPRE_SUCCESS` The optional output value has been successfully set.
- `CVBBDPRE_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 the current processor.

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

`CVBBDPrecGetNumGfnEvals`

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

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

Arguments `bbd_data` (`void *`) pointer to the CVBBDPRE data structure.

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

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

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

`CVBBDPRE_PDATA_NULL` The CVBBDPRE preconditioner has not been initialized.

`CVBBDPrecGetReturnFlagName`

Call `name = CVBBDPrecGetReturnFlagName(flag);`

Description The function `CVBBDPrecGetReturnFlagName` returns the name of the CVBBDPRE constant corresponding to `flag`.

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

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

In addition to the `ngevalsBBDP` `gloc` evaluations, the costs associated with CVBBDPRE also include `nlinsetups` LU factorizations, `nlinsetups` calls to `cfn`, `npsolves` banded backsolve calls, and `nfevalsLS` right-hand side function evaluations, where `nlinsetups` is an optional CVODE output and `npsolves` and `nfevalsLS` are linear solver optional outputs (see §5.5.7).

Chapter 6

FCVODE, an Interface Module for FORTRAN Applications

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

6.1 FCVODE routines

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

- Interface to the NVECTOR modules
  - FNVINITS (defined by NVECTOR_SERIAL) interfaces to N_VNewEmpty_Serial.
  - FNVINITP (defined by NVECTOR_PARALLEL) interfaces to N_VNewEmpty_Parallel.
- Interface to the main CVODE module
  - FCVMALLOC interfaces to CVodeCreate, CVodeSetFdata, and CVodeMalloc.
  - FCVREINIT interfaces to CVodeReInit.
  - FCVSETIIN and FCVSETRIN interface to CVodeSet* functions.
  - FCVEWTSET interfaces to CVodeSetEwtFn.
  - FCVODE interfaces to CVode, CVodeGet* functions, and to the optional output functions for the selected linear solver module.
  - FCVDKY interfaces to the interpolated output function CVodeGetDky.
  - FCVGETERRWEIGHTS interfaces to CVodeGetErrWeights.
  - FCVGETESTLOCALERR interfaces to CVodeGetEstLocalErrors.
  - FCVFREE interfaces to CVodeFree.
- Interface to the linear solver modules
  - FCVDIAG interfaces to CVDiag
  - FCVDENSE interfaces to CVDense.
  - FCVDENSESETJAC interfaces to CVDenseSetJacFn.
  - FCVBAND interfaces to CVBand.

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

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

FCVODE routine (FORTRAN)	CVODE function (C)	CVODE function type
FCVFUN	FCVf	CVRhsFn
FCVEWT	FCVEwtSet	CVEwtFn
FCVDJAC	FCVDenseJac	CVDenseJacFn
FCVBJAC	FCVBandJac	CVBandJacFn
FCVPSOL	FCVPSol	CVSpilsPrecSolveFn
FCVPSET	FCVPSet	CVSpilsPrecSetupFn
FCVJTIMES	FCVJtimes	CVSpilsJacTimesVecFn

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

6.1.1 Important note on portability

In this package, the names of the interface functions, and the names of the FORTRAN user routines called by them, appear as dummy names which are mapped to actual values by a series of definitions in the header files `fcvode.h`, `fcvroot.h`, `fcvbp.h`, and `fcvbbd.h`. By default, those mapping definitions depend in turn on the C macro `F77_FUNC` defined in the header file `sundials_config.h` by `configure`. However, the set of flags `SUNDIALS_CASE_UPPER`, `SUNDIALS_CASE_LOWER`, `SUNDIALS_UNDERSCORE_NONE`, `SUNDIALS_UNDERSCORE_ONE`, and `SUNDIALS_UNDERSCORE_TWO` can be explicitly defined in the header file `sundials_config.h` when configuring SUNDIALS via the options `--with-f77underscore` and `--with-f77case` to override the default behavior if necessary (see Chapter 2). Either way, the names into which the dummy names are mapped are in upper or lower case and have up to two underscores appended.

The user must also ensure that variables in the user FORTRAN code are declared in a manner consistent with their counterparts in CVODE. All real variables must be declared as `REAL`, `DOUBLE PRECISION`, or perhaps as `REAL*`n, where n denotes the number of bytes, depending on whether CVODE was built in single, double, or extended precision (see Chapter 2). Moreover, some of the FORTRAN integer variables must be declared as `INTEGER*4` or `INTEGER*8` according to the C type `long int`. These integer variables include: the array of integer optional outputs (`IOUT`), problem dimensions (`NEQ`, `NLOCAL`, `NGLOBAL`), Jacobian half-bandwidths (`MU`, `ML`, etc.), as well as the array of user integer data, `IPAR`. This is particularly important when using CVODE and the FCVODE package on 64-bit architectures.

6.2 Usage of the FCVODE interface module

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

Steps marked with [**S**] in the instructions below apply to the serial NVECTOR implementation (NVECTOR_SERIAL) only, while those marked with [**P**] apply to NVECTOR_PARALLEL.

1. **Right-hand side specification**

   The user must in all cases supply the following FORTRAN routine

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

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

2. NVECTOR **module initialization**

   [**S**] To initialize the serial NVECTOR module, the user must make the following call:

   ```
   CALL FNVINITS(KEY, NEQ, IER)
   ```

   where `KEY` is the solver id (`KEY` $= 1$ for CVODE), `NEQ` is the size of vectors, and `IER` is a return completion flag which is 0 on success and -1 if a failure occurred.

   [**P**] To initialize the parallel vector module, the user must make the following call:

   ```
   CALL FNVINITP(COMM, KEY, NLOCAL, NGLOBAL, IER)
   ```

   in which the arguments are: `COMM` = MPI communicator, `KEY` = 1, `NLOCAL` = the local size of vectors on this processor, and `NGLOBAL` = the system size (and the global size of all vectors, equal to the sum of all values of `NLOCAL`). The return flag `IER` is set to 0 on a successful return and to -1 otherwise.

   If the header file `sundials_config.h` defines `SUNDIALS_MPI_COMM_F2C` to be 1 (meaning the MPI implementation used to build SUNDIALS includes the `MPI_Comm_f2c` function), then `COMM` can be any valid MPI communicator. Otherwise, `MPI_COMM_WORLD` will be used, so just pass an integer value as a placeholder.

3. **Problem specification**

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

   `FCVMALLOC`

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

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

   Arguments `T0` is the initial value of t.
   `Y0` is an array of initial conditions.

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

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

Notes The user intger data array `IPAR` must be declared as `INTEGER*4` or `INTEGER*8` according to the C type `long int`.

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

The optional outputs associated with the main CVODE integrator are listed in Table 6.2.

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

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

It must set the positive components of the error weight vector `EWT` for the calculation of the WRMS norm of `Y`. On return, set `IER = 0` if `FCVEWT` was successful, and nonzero otherwise. The arrays `IPAR` (of integers) and `RPAR` (of reals) contain user data and are the same as those passed to `FCVMALLOC`.

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

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

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

4. **Linear solver specification**

In the case of a stiff system, the implicit `BDF` method involves the solution of linear systems related to the Jacobian $J = \partial f / \partial y$ of the ODE system. CVODE presently includes six choices for the treatment of these systems, and the user of FCVODE must call a routine with a specific name to make the desired choice.

[S] Diagonal approximate Jacobian

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

```
CALL FCVDIAG(IER)
```

IER is an error return flag set on 0 on success or −1 if a memory failure occurred. There is no additional user-supplied routine. Optional outputs specific to the DIAG case listed in Table 6.2.

[S] Dense treatment of the linear system

The user must make the call:

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

where NEQ is the size of the ODE system. The argument IER is an error return flag which is 0 for success , −1 if a memory allocation failure occurred, or −2 for illegal input. As an option when using the DENSE linear solver, the user may supply a routine that computes a dense approximation of the system Jacobian $J = \partial f/\partial y$. If supplied, it must have the following form:

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

Typically this routine will use only NEQ, T, Y, and DJAC. It must compute the Jacobian and store it columnwise in DJAC. The input arguments T, Y, and FY contain the current values of t, y, and $f(t, y)$, respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FCVDJAC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVDJAC failed unrecoverably (in which case the integration is halted).

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

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

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

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

Optional outputs specific to the DENSE case are listed in Table 6.2.

[S] Band treatment of the linear system

The user must make the call:

```
CALL FCVBAND (NEQ, MU, ML, IER)
```

The arguments are: MU, the upper half-bandwidth; ML, the lower half-bandwidth; and IER an error return flag which is 0 for success , −1 if a memory allocation failure occurred, or −2 in case an input has an illegal value.

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

```
 SUBROUTINE FCVBJAC(NEQ, MU, ML, MDIM, T, Y, FY, BJAC, H, IPAR, RPAR,
&                   WK1, WK2, WK3, IER)
```

```
 DIMENSION Y(*), FY(*), BJAC(MDIM,*), IPAR(*), RPAR(*),
&                WK1(*), WK2(*), WK3(*)
```

Typically this routine will use only NEQ, MU, ML, T, Y, and BJAC. It must load the MDIM by N array BJAC with the Jacobian matrix at the current (t,y) in band form. Store in BJAC(k,j) the Jacobian element $J_{i,j}$ with $k = i - j+$ MU $+1$ ($k = 1\cdots$ ML + MU + 1) and $j = 1\cdots N$. The input arguments T, Y, and FY contain the current values of t, y, and $f(t,y)$, respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FCVBJAC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVBJAC failed unrecoverably (in which case the integration is halted).

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

If the FCVBJAC routine is provided, then, following the call to FCVBAND, the user must make the call:

```
CALL FCVBANDSETJAC(FLAG, IER)
```

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

Optional outputs specific to the BAND case are listed in Table 6.2.

[S][P] SPGMR treatment of the linear systems

For the Scaled Preconditioned GMRES solution of the linear systems, the user must make the call

```
CALL FCVSPGMR(IPRETYPE, IGSTYPE, MAXL, DELT, IER)
```

The arguments are as follows. IPRETYPE specifies the preconditioner type: 0 for no preconditioning, 1 for left only, 2 for right only, or 3 for both sides. IGSTYPE indicates the Gram-Schmidt process type: 1 for modified G-S or 2 for classical G-S. MAXL is the maximum Krylov subspace dimension. DELT is the linear convergence tolerance factor. For all of the input arguments, a value of 0 or 0.0 indicates the default. IER is an error return flag which is 0 to indicate success, -1 if a memory allocation failure occurred, or -2 to indicate an illegal input.

Optional outputs specific to the SPGMR case are listed in Table 6.2.

For descriptions of the relevant optional user-supplied routines, see **User-supplied routines for SPGMR/SPBCG/SPTFQMR** below.

[S][P] SPBCG treatment of the linear systems

For the Scaled Preconditioned Bi-CGStab solution of the linear systems, the user must make the call

```
CALL FCVSPBCG(IPRETYPE, MAXL, DELT, IER)
```

Its arguments are the same as those with the same names for FCVSPGMR.

Optional outputs specific to the SPBCG case are listed in Table 6.2.

For descriptions of the relevant optional user-supplied routines, see **User-supplied routines for SPGMR/SPBCG/SPTFQMR** below.

[S][P] SPTFQMR treatment of the linear systems

For the Scaled Preconditioned Transpose-Reee Quasi-Minimal Residual solution of the linear systems, the user must make the call

```
CALL FCVSPTFQMR(IPRETYPE, MAXL, DELT, IER)
```

Its arguments are the same as those with the same names for `FCVSPGMR`.

Optional outputs specific to the SPTFQMR case are listed in Table 6.2.

For descriptions of the relevant optional user-supplied routines, see below.

[S][P] Functions used by SPGMR/SPBCG/SPTFQMR

An optional user-supplied routine, `FCVJTIMES` (see below), can be provided for Jacobian-vector products. If it is, then, following the call to `FCVSPGMR`, `FCVSPBCG`, or `FCVSPTFQMR`, the user must make the call:

```
CALL FCVSPILSSETJAC(FLAG, IER)
```

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

If preconditioning is to be done (`IPRETYPE` $\neq 0$), then the user must call

```
CALL FCVSPILSSETPREC(FLAG, IER)
```

with `FLAG` $\neq 0$. The return flag `IER` is 0 if successful, or negative if a memory error occurred. In addition, the user program must include preconditioner routines `FCVPSOL` and `FCVPSET` (see below).

[S][P] User-supplied routines for SPGMR/SPBCG/SPTFQMR

With treatment of the linear systems by any of the Krylov iterative solvers, there are three optional user-supplied routines — `FCVJTIMES`, `FCVPSOL`, and `FCVPSET`. The specifications for these routines are given below.

As an option when using the SPGMR, SPBCG, or SPTFQMR linear solvers, the user may supply a routine that computes the product of the system Jacobian $J = \partial f/\partial y$ and a given vector v. If supplied, it must have the following form:

```
SUBROUTINE FCVJTIMES (V, FJV, T, Y, FY, H, IPAR, RPAR, WORK, IER)
DIMENSION V(*), FJV(*), Y(*), FY(*), IPAR(*), RPAR(*), WORK(*)
```

Typically this routine will use only `NEQ`, `T`, `Y`, `V`, and `FJV`. It must compute the product vector Jv, where the vector v is stored in `V`, and store the product in `FJV`. The input arguments `T`, `Y`, and `FY` contain the current values of t, y, and $f(t, y)$, respectively. On return, set `IER = 0` if `FCVJTIMES` was successful, and nonzero otherwise. The arrays `IPAR` (of integers) and `RPAR` (of reals) contain user data and are the same as those passed to `FCVMALLOC`. The vector `WORK`, of length `NEQ`, is provided as work space for use in `FCVJTIMES`.

If preconditioning is to be included, the following routine must be supplied, for solution of the preconditioner linear system:

```
 SUBROUTINE FCVPSOL(T, Y, FY, R, Z, GAMMA, DELTA, LR, IPAR, RPAR,
&                   WORK, IER)
 DIMENSION Y(*), FY(*), R(*), Z(*), IPAR(*), RPAR(*), WORK(*)
```

It must solve the preconditioner linear system $Pz = r$, where $r =$ `R` is input, and store the solution z in `Z`. Here P is the left preconditioner if `LR=1` and the right preconditioner if `LR=2`.

The preconditioner (or the product of the left and right preconditioners if both are nontrivial) should be an approximation to the matrix $I - \gamma J$, where I is the identity matrix, J is the system Jacobian, and $\gamma =$ `GAMMA`. The input arguments `T`, `Y`, and `FY` contain the current values of t, y, and $f(t, y)$, respectively. On return, set `IER = 0` if `FCVPSOL` was successful, set `IER` positive if a recoverable error occurred, and set `IER` negative if a non-recoverable error occurred.

The arrays `IPAR` (of integers) and `RPAR` (of reals) contain user data and are the same as those passed to `FCVMALLOC`. The argument `WORK` is a work array of length `NEQ` for use by this routine.

If the user's preconditioner requires that any Jacobian related data be evaluated or preprocessed, then the following routine can be used for the evaluation and preprocessing of the preconditioner:

```
 SUBROUTINE FCVPSET(T, Y, FY, JOK, JCUR, GAMMA, H, IPAR, RPAR,
&                    WORK1, WORK2, WORK3, IER)
 DIMENSION Y(*), FY(*), EWT(*), IPAR(*), RPAR(*),
&          WORK1(*), WORK2(*), WORK3(*)
```

It must perform any evaluation of Jacobian-related data and preprocessing needed for the solution of the preconditioner linear systems by `FCVPSOL`. The input argument `JOK` allows for Jacobian data to be saved and reused: If `JOK = 0`, this data should be recomputed from scratch. If `JOK = 1`, a saved copy of it may be reused, and the preconditioner constructed from it. The input arguments `T`, `Y`, and `FY` contain the current values of t, y, and $f(t, y)$, respectively. On return, set `JCUR = 1` if Jacobian data was computed, and set `JCUR = 0` otherwise. Also on return, set `IER = 0` if `FCVPSET` was successful, set `IER` positive if a recoverable error occurred, and set `IER` negative if a non-recoverable error occurred.

The arrays `IPAR` (of integers) and `RPAR` (of reals) contain user data and are the same as those passed to `FCVMALLOC`. The arguments `WORK1`, `WORK2`, `WORK3` are work arrays of length `NEQ` for use by this routine.

! If the user calls `FCVSPILSSETPREC` the routine `FCVPSET` must be provided, even if it is not needed and must return `IER=0`.

Notes

(a) If the user's `FCVJTIMES` or `FCVPSET` routine uses difference quotient approximations, it may need to use the error weight array `EWT`, the current stepsize `H`, and/or the unit roundoff, in the calculation of suitable increments. Also, If `FCVPSOL` uses an iterative method in its solution, the residual vector $\rho = r - Pz$ of the system should be made less than `DELTA` in weighted ℓ_2 norm, i.e. $\sqrt{\sum(\rho_i * \mathtt{EWT}[i])^2} < \mathtt{DELTA}$.

(b) If needed in `FCVJTIMES`, `FCVPSOL`, or `FCVPSET`, the error weight array `EWT` can be obtained by calling `FCVGETERRWEIGHTS` using one of the work arrays as temporary storage for `EWT`.

(c) If needed in `FCVJTIMES`, `FCVPSOL`, or `FCVPSET`, the unit roundoff can be obtained as the optional output `ROUT(6)` (available after the call to `FCVMALLOC`) and can be passed using either the `RPAR` user data array or a common block.

5. **Problem solution**

Carrying out the integration is accomplished by making calls as follows:

```
CALL FCVODE(TOUT, T, Y, ITASK, IER)
```

The arguments are as follows. `TOUT` specifies the next value of t at which a solution is desired (input). `T` is the value of t reached by the solver on output. `Y` is an array containing the computed solution on output. `ITASK` is a task indicator and should be set to 1 for normal mode (overshoot `TOUT` and interpolate), to 2 for one-step mode (return after each internal step taken), to 3 for normal mode with the additional `tstop` constraint, or to 4 for one-step mode with the additional constraint `tstop`. `IER` is a completion flag and will be set to a positive value upon successful return or to a negative value if an error occurred. These values correspond to the `CVode` returns (see §5.5.4 and §10.2). The current values of the optional outputs are available in `IOUT` and `ROUT` (see Table 6.2).

6. **Additional solution output**

   To obtain a derivative of the solution, of order up to the current method order, make the following call:

   ```
   CALL FCVDKY(T, K, DKY, IER)
   ```

   where `T` is the value of t at which solution derivative is desired, and `K` is the derivative order ($0 \leq$ `K` $\leq$ `QU`). On return, `DKY` is an array containing the computed `K`-th derivative of y. The value `T` must lie between `TCUR - HU` and `TCUR`. The return flag `IER` is set to 0 upon successful return or to a negative value to indicate an illegal input.

7. **Problem reinitialization**

   To re-initialize the CVODE solver for the solution of a new problem of the same size as one already solved, make the following call:

   ```
   CALL FCVREINIT(T0, Y0, IATOL, RTOL, ATOL, IER)
   ```

   The arguments have the same names and meanings as those of `FCVMALLOC`. `FCVREINIT` performs the same initializations as `FCVMALLOC`, but does no memory allocation, using instead the existing internal memory created by the previous `FCVMALLOC` call. The call to specify the linear system solution method may or may not be needed.

   Following this call, a call to specify the linear system solver must be made if the choice of linear solver is being changed. Otherwise, a call to reinitialize the linear solver last used may or may not be needed, depending on changes in the inputs to it.

   In the case of the BAND solver, for any change in the half-bandwidths, call `FCVBAND` as described above.

   In the case of SPGMR, for a change of inputs other than `MAXL`, make the call

   ```
   CALL FCVSPGMRREINIT (IPRETYPE, IGSTYPE, DELT, IER)
   ```

   which reinitializes SPGMR without reallocating its memory. The arguments have the same names and meanings as those of `FCVSPGMR`. If `MAXL` is being changed, then call `FCVSPGMR` instead.

   In the case of SPBCG, for a change in any inputs, make the call

   ```
   CALL FCVSPBCGREINIT (IPRETYPE, MAXL, DELT, IER)
   ```

   which reinitializes SPBCG without reallocating its memory. The arguments have the same names and meanings as those of `FCVSPBCG`.

   In the case of SPTFQMR, for a change in any inputs, make the call

   ```
   CALL FCVSPTFQMRREINIT (IPRETYPE, MAXL, DELT, IER)
   ```

   which reinitializes SPTFQMR without reallocating its memory. The arguments have the same names and meanings as those of `FCVSPTFQMR`.

Table 6.1: Keys for setting FCVODE optional inputs

Integer optional inputs (`FCVSETIIN`)

Key	Optional input	Default value
`MAX_ORD`	Maximum LMM method order	5 (BDF), 12 (Adams)
`MAX_NSTEPS`	Maximum no. of internal steps before $t_{\rm out}$	500
`MAX_ERRFAIL`	Maximum no. of error test failures	7
`MAX_NITERS`	Maximum no. of nonlinear iterations	3
`MAX_CONVFAIL`	Maximum no. of convergence failures	10
`HNIL_WARNS`	Maximum no. of warnings for $t_n + h = t_n$	10
`STAB_LIM`	Flag to activate stability limit detection	0

Real optional inputs (`FCVSETRIN`)

Key	Optional input	Default value
`INIT_STEP`	Initial step size	estimated
`MAX_STEP`	Maximum absolute step size	∞
`MIN_STEP`	Minimum absolute step size	0.0
`STOP_TIME`	Value of t_{stop}	undefined
`NLCONV_COEF`	Coefficient in the nonlinear convergence test	0.1

8. **Memory deallocation**

   To free the internal memory created by the call to `FCVMALLOC`, make the call

   ```
   CALL FCVFREE
   ```

6.3 FCVODE optional input and output

In order to keep the number of user-callable FCVODE interface routines to a minimum, optional inputs to the CVODE solver are passed through only two routines: `FCVSETIIN` for integer optional inputs and `FCVSETRIN` for real optional inputs. These functions should be called as follows:

```
CALL FCVSETIIN(KEY, IVAL, IER)
CALL FCVSETRIN(KEY, RVAL, IER)
```

where `KEY` is a quoted string indicating which optoinal input is set (see Table 6.1), `IVAL` is the integer input value to be used, `RVAL` is the real input value to be used, and `IER` is an integer return flag which is set to 0 on success and a negative value if a failure occurred.

The optional outputs from the CVODE solver are accessed not through individual functions, but rather through a pair of arrays, `IOUT` (integer type) of dimension at least 21, and `ROUT` (real type) of dimension at least 6. These arrays are owned (and allocated) by the user and are passed as arguments to `FCVMALLOC`. Table 6.2 lists the entries in these two arrays and specifies the optional variable as well as the CVODE function which is actually called to extract the optional output.

For more details on the optional inputs and outputs, see §5.5.5 and §5.5.7.

In addition to the optional inputs communicated through `FCVSET*` calls and the optional outputs extracted from `IOUT` and `ROUT`, the following user-callable routines are available:

To obtain the error weight array `EWT`, containing the multiplicative error weights used the WRMS norms, make the following call:

```
CALL FCVGETERRWEIGHTS (EWT, IER)
```

This computes the `EWT` array normally defined by Eq. (3.6). The array `EWT`, of length `NEQ` or `NLOCAL`, must already have been declared by the user. The error return flag `IER` is zero if successful, and negative if there was a memory error.

Table 6.2: Description of the FCVODE optional output arrays IOUT and ROUT

Integer output array IOUT

Index	Optional output	CVODE function
CVODE main solver		
1	LENRW	CVodeGetWorkSpace
2	LENIW	CVodeGetWorkSpace
3	NST	CVodeGetNumSteps
4	NFE	CVodeGetNumRhsEvals
5	NETF	CVodeGetNumErrTestFails
6	NCFN	CVodeGetNumNonlinSolvConvFails
7	NNI	CVodeGetNumNonlinSolvIters
8	NSETUPS	CVodeGetNumLinSolvSetups
9	QU	CVodeGetLastOrder
10	QCUR	CVodeGetCurrentOrder
11	NOR	CVodeGetNumStabLimOrderReds
12	NGE	CVodeGetNumGEvals
CVDENSE linear solver		
13	LENRWLS	CVDenseGetWorkSpace
14	LENIWLS	CVDenseGetWorkSpace
15	LS_FLAG	CVDenseGetLastFlag
16	NFELS	CVDenseGetNumRhsEvals
17	NJE	CVDenseGetNumJacEvals
CVBAND linear solver		
13	LENRWLS	CVBandGetWorkSpace
14	LENIWLS	CVBandGetWorkSpace
15	LS_FLAG	CVBandGetLastFlag
16	NFELS	CVBandGetNumRhsEvals
17	NJE	CVBandGetNumJacEvals
CVDIAG linear solver		
13	LENRWLS	CVDiagGetWorkSpace
14	LENIWLS	CVDiagGetWorkSpace
15	LS_FLAG	CVDiagGetLastFlag
16	NFELS	CVDiagGetNumRhsEvals
CVSPGMR, CVSPBCG, CVSPTFQMR linear solvers		
13	LENRWLS	CVSpilsGetWorkSpace
14	LENIWLS	CVSpilsGetWorkSpace
15	LS_FLAG	CVSpilsGetLastFlag
16	NFELS	CVSpilsGetNumRhsEvals
17	NJTV	CVSpilsGetNumJacEvals
18	NPE	CVSpilsGetNumPrecEvals
19	NPS	CVSpilsGetNumPrecSolves
20	NLI	CVSpilsGetNumLinIters
21	NCFL	CVSpilsGetNumConvFails

Real output array ROUT

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

To obtain the estimated local errors, following a successful call to FCVSOLVE, make the following call:

```
CALL FCVGETESTLOCALERR (ELE, IER)
```

This computes the ELE array of estimated local errors as of the last step taken. The array ELE must already have been declared by the user. The error return flag IER is zero if successful, and negative if there was a memory error.

6.4 Usage of the FCVROOT interface to rootfinding

The FCVROOT interface package allows programs written in FORTRAN to use the rootfinding feature of the CVODE solver module. The user-callable functions in FCVROOT, with the corresponding CVODE functions, are as follows:

- FCVROOTINIT interfaces to CVodeRootInit.
- FCVROOTINFO interfaces to CVodeGetRootInfo.
- FCVROOTFREE interfaces to CVodeRootFree.

In order to use the rootfinding feature of CVODE, the following call must be made, after calling FCVMALLOC but prior to calling FCVODE, to allocate and initialize memory for the FCVROOT module:

```
CALL FCVROOTINIT (NRTFN, IER)
```

The arguments are as follows: NRTFN is the number of root functions. IER is a return completion flag; its values are 0 for success, -1 if the CVODE memory was NULL, and -11 if a memory allocation failed.

To specifiy the functions whose roots are to be found, the user must define the following routine:

```
SUBROUTINE FCVROOTFN (T, Y, G, IPAR, RPAR, IER)
DIMENSION Y(*), G(*), IPAR(*), RPAR(*)
```

It must set the G array, of length NRTFN, with components $g_i(t, y)$, as a function of T $= t$ and the array Y $= y$. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. Set IER on 0 if successful, or on a non-zero value if an error occurred.

When making calls to FCVODE to solve the ODE system, the occurrence of a root is flagged by the return value IER $= 2$. In that case, if NRTFN > 1, the functions g_i which were found to have a root can be identified by making the following call:

```
CALL FCVROOTINFO (NRTFN, INFO, IER)
```

The arguments are as follows: NRTFN is the number of root functions. INFO is an integer array of length NRTFN with root information. IER is a return completion flag; its values are 0 for success, negative if there was a memory failure. The returned values of INFO(i) (i$= 1, \ldots,$ NRTFN) are 0 or 1, such that INFO(i) $= 1$ if $g_{\mathtt{i}}$ was found to have a root, and INFO(i) $= 0$ otherwise.

The total number of calls made to the root function FCVROOTFN, denoted NGE, can be obtained from IOUT(12). If the FCVODE/CVODE memory block is reinitialized to solve a different problem via a call to FCVREINIT, then the counter NGE is reset to zero.

To free the memory resources allocated by a prior call to FCVROOTINIT make the following call:

```
CALL FCVROOTFREE
```

See §5.7 for additional information on the rootfinding feature.

6.5 Usage of the FCVBP interface to CVBANDPRE

The FCVBP interface sub-module is a package of C functions which, as part of the FCVODE interface module, support the use of the CVODE solver with the serial NVECTOR_SERIAL module, and the combination of the CVBANDPRE preconditioner module (see §5.8.1) with any of the Krylov iterative linear solvers.

The user-callable functions in this package, with the corresponding CVODE and CVBANDPRE functions, are as follows:

- `FCVBPINIT` interfaces to `CVBandPrecAlloc`.
- `FCVBPSPGMR` interfaces to `CVBPSpgmr` and SPGMR optional input functions.
- `FCVBPREINIT` interfaces to `CVBandPrecReInit`.
- `FCVBPOPT` interfaces to CVBANDPRE optional output functions.
- `FCVBPFREE` interfaces to `CVBandPrecFree`.

As with the rest of the FCVODE routines, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `fcvbp.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §6.2 are grayed-out.

1. **Right-hand side specification**
2. NVECTOR **module initialization**
3. **Problem specification**
4. **Linear solver specification**

   To initialize the CVBANDPRE preconditioner, make the following call:

   ```
   CALL FCVBPINIT(NEQ, MU, ML, IER)
   ```

   The arguments are as follows. `NEQ` is the problem size. `MU` and `ML` are the upper and lower half-bandwidths of the band matrix that is retained as an approximation of the Jacobian. `IER` is a return completion flag. A value of 0 indicates success, while a value of -1 indicates that a memory failure occurred.

   To specify the SPGMR linear system solver and use the CVBANDPRE preconditioner, make the following call:

   ```
   CALL FCVBPSPGMR(IPRETYPE, IGSTYPE, MAXL, DELT, IER)
   ```

   Its arguments are the same as those of `FCVSPGMR` (see step 4 in §6.2).

   To specify the SPBCG linear system solver and use the CVBANDPRE preconditioner, make the following call:

   ```
   CALL FCVBPSPBCG(IPRETYPE, MAXL, DELT, IER)
   ```

   Its arguments are the same as those of `FCVSPBCG` (see step 4 in §6.2).

   To specify the SPBCG linear system solver and use the CVBANDPRE preconditioner, make the following call:

   ```
   CALL FCVBPSPTFQMR(IPRETYPE, MAXL, DELT, IER)
   ```

Its arguments are the same as those of FCVSPTFQMR (see step 4 in §6.2).

Optionally, to specify that SPGMR, SPBCG, or SPTFQMR should use the supplied FCVJTIMES, make the call

```
CALL FCVSPILSSETJAC(FLAG, IER)
```

with FLAG $\neq 0$ (see step 4 in §6.2 for details).

5. **Problem solution**

6. CVBANDPRE **Optional outputs**

   Optional outputs specific to the SPGMR, SPBCG, or SPTFQMR solver are listed in Table 6.2. To obtain the optional outputs associated with the CVBANDPRE module, make the following call:

   ```
   CALL FCVBPOPT(LENRWBP, LENIWBP, NFEBP)
   ```

   The arguments returned are as follows. LENRWBP is the length of real preconditioner work space, in realtype words. LENIWBP is the length of integer preconditioner work space, in integer words. NFEBP is the number of $f(t, y)$ evaluations (calls to FCVFUN) for difference-quotient banded Jacobian approximations.

7. **Memory deallocation**

   To free the internal memory created by the call to FCVBPINIT, before calling FCVFREE, the user must make the call

   ```
   CALL FCVBPFREE
   ```

6.6 Usage of the FCVBBD interface to CVBBDPRE

The FCVBBD interface sub-module is a package of C functions which, as part of the FCVODE interface module, support the use of the CVODE solver with the parallel NVECTOR_PARALLEL module, and the combination of the CVBBDPRE preconditioner module (see §5.8.2) with any of the Krylov iterative linear solvers.

The user-callable functions in this package, with the corresponding CVODE and CVBBDPRE functions, are as follows:

- FCVBBDINIT interfaces to CVBBDPrecAlloc.
- FCVBBDSPGMR interfaces to CVBBDSpgmr and SPGMR optional input functions.
- FCVBBDSPBCG interfaces to CVBBDSpbcg and SPBCG optional input functions.
- FCVBBDSPTFQMR interfaces to CVBBDSptfqmr and SPTFQMR optional input functions.
- FCVBBDREINIT interfaces to CVBBDPrecReInit.
- FCVBBDOPT interfaces to CVBBDPRE optional output functions.
- FCVBBDFREE interfaces to CVBBDPrecFree.

In addition to the FORTRAN right-hand side function FCVFUN, the user-supplied functions used by this package, are listed below, each with the corresponding interface function which calls it (and its type within CVBBDPRE or CVODE):

FCVBBD routine (FORTRAN)	CVODE function (C)	CVODE function type
FCVLOCFN	FCVgloc	CVLocalFn
FCVCOMMF	FCVcfn	CVCommFn
FCVJTIMES	FCVJtimes	CVSpilsJacTimesVecFn

As with the rest of the FCVODE routines, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program. Additionally, based on flags discussed above in §6.1, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `fcvbbd.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §6.2 are grayed-out.

1. **Right-hand side specification**
2. NVECTOR **module initialization**
3. **Problem specification**
4. **Linear solver specification**

   To initialize the CVBBDPRE preconditioner, make the following call:

   ```
   CALL FCVBBDINIT(NLOCAL, MUDQ, MLDQ, MU, ML, DQRELY, IER)
   ```

   The arguments are as follows. `NLOCAL` is the local size of vectors on this processor. `MUDQ` and `MLDQ` are the upper and lower half-bandwidths to be used in the computation of the local Jacobian blocks by difference quotients. These may be smaller than the true half-bandwidths of the Jacobian of the local block of g, when smaller values may provide greater efficiency. `MU` and `ML` are the upper and lower half-bandwidths of the band matrix that is retained as an approximation of the local Jacobian block. These may be smaller than `MUDQ` and `MLDQ`. `DQRELY` is the relative increment factor in y for difference quotients (optional). A value of 0.0 indicates the default, $\sqrt{\text{unit roundoff}}$. `IER` is a return completion flag. A value of 0 indicates success, while a value of -1 indicates that a memory failure occurred or that an input had an illegal value.

   To specify the SPGMR linear system solver and use the CVBBDPRE preconditioner, make the following call:

   ```
   CALL FCVBBDSPGMR(IPRETYPE, IGSTYPE, MAXL, DELT, IER)
   ```

   Its arguments are the same as those of `FCVSPGMR` (see step 4 in §6.2).

   To specify the SPBCG linear system solver and use the CVBBDPRE preconditioner, make the following call:

   ```
   CALL FCVBBDSPBCG(IPRETYPE, MAXL, DELT, IER)
   ```

   Its arguments are the same as those of `FCVSPBCG` (see step 4 in §6.2).

   To specify the SPTFQMR linear system solver and use the CVBBDPRE preconditioner, make the following call:

   ```
   CALL FCVBBDSPTFQMR(IPRETYPE, MAXL, DELT, IER)
   ```

   Its arguments are the same as those of `FCVSPTFQMR` (see step 4 in §6.2).

   Optionally, to specify that SPGMR, SPBCG, or SPTFQMR should use the supplied `FCVJTIMES`, make the call

   ```
   CALL FCVSPILSSETJAC(FLAG, IER)
   ```

with FLAG $\neq 0$ (see step 4 in §6.2 for details).

5. **Problem solution**

6. CVBBDPRE **Optional outputs**

   Optional outputs specific to the SPGMR, SPBCG, or SPTFQMR solver are listed in Table 6.2. To obtain the optional outputs associated with the CVBBDPRE module, make the following call:

   ```
   CALL FCVBBDOPT(LENRWBBD, LENIWBBD, NGEBBD)
   ```

   The arguments returned are as follows. LENRWBBD is the length of real preconditioner work space, in realtype words. LENIWBBD is the length of integer preconditioner work space, in integer words. These sizes are local to the current processor. NGEBBD is the number of $g(t, y)$ evaluations (calls to FCVLOCFN) so far.

7. **Problem reinitialization**

   If a sequence of problems of the same size is being solved using the same linear solver (SPGMR, SPBCG, or SPTFQMR) in combination with the CVBBDPRE preconditioner, then the CVODE package can be re-initialized for the second and subsequent problems by calling FCVREINIT, following which a call to FCVBBDINIT may or may not be needed. If the input arguments are the same, no FCVBBDINIT call is needed. If there is a change in input arguments other than MU or ML, then the user program should make the call

   ```
   CALL FCVBBDREINIT(NLOCAL, MUDQ, MLDQ, DQRELY, IER)
   ```

   This reinitializes the CVBBDPRE preconditioner, but without reallocating its memory. The arguments of the FCVBBDREINIT routine have the same names and meanings as those of FCVBBDINIT. If the value of MU or ML is being changed, then a call to FCVBBDINIT must be made. Finally, if there is a change in any of the linear solver inputs, then a call to FCVBBDSPGMR, FCVBBDSPBCG, or FCVBBDSPTFQMR must be made; in this case the linear solver memory is reallocated.

8. **Memory deallocation**

   To free the internal memory created by the call to FCVBBDINIT, before calling FCVFREE, the user must make the call

   ```
   CALL FCVBBDFREE
   ```

9. **User-supplied routines**

   The following two routines must be supplied for use with the CVBBDPRE module:

   ```
   SUBROUTINE FCVGLOCFN (NLOC, T, YLOC, GLOC, IPAR, RPAR, IER)
   DIMENSION YLOC(*), GLOC(*), IPAR(*), RPAR(*)
   ```

   This routine is to evaluate the function $g(t, y)$ approximating f (possibly identical to f), in terms of $T = t$, and the array YLOC (of length NLOC), which is the sub-vector of y local to this processor. The resulting (local) sub-vector is to be stored in the array GLOC. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVGLOCFN failed unrecoverably (in which case the integration is halted).

   ```
   SUBROUTINE FCVCOMMFN (NLOC, T, YLOC, IPAR, RPAR, IER)
   DIMENSION YLOC(*), IPAR(*), RPAR(*)
   ```

This routine is to perform the inter-processor communication necessary for the FCVGLOCFN routine. Each call to FCVCOMMFN is preceded by a call to the right-hand side routine FCVFUN with the same arguments T and YLOC. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. IER is an error return flag (currently not used; set IER=0). Thus FCVCOMMFN can omit any communications done by FCVFUN if relevant to the evaluation of GLOC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVCOMMFN failed unrecoverably (in which case the integration is halted).

The subroutine FCVCOMMFN must be supplied even if it is not needed and must return IER=0.

Optionally, the user can supply a routine FCVJTIMES for the evaluation of Jacobian-vector products, as described above in step 4 in §6.2.

Chapter 7

Description of the NVECTOR module

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type `N_Vector`) through a set of operations defined by the particular NVECTOR implementation. Users can provide their own specific implementation of the NVECTOR module or use one of 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 defines and implements the vector operations acting on `N_Vector`. These routines are nothing but wrappers for the vector operations defined by a particular NVECTOR implementation, which are accessed through the *ops* field of the `N_Vector` structure. To illustrate this point we show below the implementation of a typical vector operation from the generic NVECTOR module, namely `N_VScale`, which performs the scaling of a vector `x` by a scalar `c`:

```
void N_VScale(realtype c, N_Vector x, N_Vector z)
{
   z->ops->nvscale(c, x, z);
}
```

Table 7.1 contains a complete list of all vector operations defined by the generic NVECTOR module.

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

```
N_Vector *N_VCloneVectorArray(int count, N_Vector w);
N_Vector *N_VCloneEmptyVectorArray(int count, N_Vector w);
```

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

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

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

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

A particular implementation of the NVECTOR module must:

- Specify the *content* field of `N_Vector`.
- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different `N_Vector` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free an `N_Vector` with the new *content* field and with *ops* pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `N_Vector` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the *content* field of the newly defined `N_Vector`.

Table 7.1: Description of the NVECTOR operations

Name	Usage and Description
N_VClone	`v = N_VClone(w);` Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not copy the vector, but rather allocates storage for the new vector.
N_VCloneEmpty	`v = N_VCloneEmpty(w);` Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not allocate storage for 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`: $z_i = cx_i,\ i = 0, \dots, n-1$.
`N_VAbs`	`N_VAbs(x, z);` Sets the components of the `N_Vector` `z` to be the absolute values of the components of the `N_Vector` `x`: $y_i = \lvert x_i \rvert,\ i = 0, \dots, n-1$.
`N_VInv`	`N_VInv(x, z);` Sets the components of the `N_Vector` `z` to be the inverses of the components of the `N_Vector` `x`: $z_i = 1.0/x_i,\ i = 0, \dots, n-1$. This routine may not check for division by 0. It should be called only with an `x` which is guaranteed to have all nonzero components.
`N_VAddConst`	`N_VAddConst(x, b, z);` Adds the scalar `b` to all components of `x` and returns the result in the `N_Vector` `z`: $z_i = x_i + b,\ i = 0, \dots, n-1$.
`N_VDotProd`	`d = N_VDotProd(x, y);` Returns the value of the ordinary dot product of `x` and `y`: $d = \sum_{i=0}^{n-1} x_i y_i$.
`N_VMaxNorm`	`m = N_VMaxNorm(x);` Returns the maximum norm of the `N_Vector` `x`: $m = \max_i \lvert x_i \rvert$.
`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`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i \mathrm{sign}(id_i))^2 \right) / n}$.
`N_VMin`	`m = N_VMin(x);` Returns the smallest element of the `N_Vector` `x`: $m = \min_i x_i$.
`N_VWL2Norm`	`m = N_VWL2Norm(x, w);` Returns the weighted Euclidean ℓ_2 norm of the `N_Vector` `x` with weight vector `w`: $m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}$.
`N_VL1Norm`	`m = N_VL1Norm(x);` Returns the ℓ_1 norm of the `N_Vector` `x`: $m = \sum_{i=0}^{n-1} \lvert x_i \rvert$.
`N_VCompare`	`N_VCompare(c, x, z);` Compares the components of the `N_Vector` `x` to the scalar `c` and returns an `N_Vector` `z` such that: $z_i = 1.0$ if $\lvert x_i \rvert \geq c$ and $z_i = 0.0$ otherwise.

continued on next page

continued from last page	
Name	**Usage and Description**
N_VInvTest	`t = N_VInvTest(x, z);` Sets the components of the `N_Vector z` to be the inverses of the components of the `N_Vector x`, with prior testing for zero values: $z_i = 1.0/x_i$, $i = 0, \ldots, n-1$. This routine returns `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 `sundials_types.h`) is returned.

7.1 The NVECTOR_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of *data*.

```
struct _N_VectorContent_Serial {
  long int length;
  booleantype own_data;
  realtype *data;
};
```

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

- `NV_CONTENT_S`

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

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

  Implementation:

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

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

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

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

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

Implementation:

```
#define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
#define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
#define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
```

- `NV_Ith_S`

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

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

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

  Implementation:

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

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in Table 7.1. Their names are obtained from those in Table 7.1 by appending the suffix `_Serial`. The module NVECTOR_SERIAL provides the following additional user-callable routines:

- `N_VNew_Serial`

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

  ```
  N_Vector N_VNew_Serial(long int vec_length);
  ```

- `N_VNewEmpty_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_VCloneVectorArray_Serial`

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

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

- `N_VCloneVectorArrayEmpty_Serial`

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

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

- `N_VDestroyVectorArray_Serial`

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

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

- `N_VPrint_Serial`

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

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

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_S(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_S(v,i)` within the loop.
- `N_VNewEmpty_Serial`, `N_VMake_Serial`, and `N_VCloneVectorArrayEmpty_Serial` set the field *own_data* = `FALSE`. `N_VDestroy_Serial` and `N_VDestroyVectorArray_Serial` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.
- To maximize efficiency, vector operations in the NVECTOR_SERIAL implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

7.2 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 7.1 Their names are obtained from those in Table 7.1 by appending the suffix `_Parallel`. The module NVECTOR_PARALLEL provides the following additional user-callable routines:

- `N_VNew_Parallel`

  This function creates and allocates memory for a parallel vector.

  ```
  N_Vector N_VNew_Parallel(MPI_Comm comm,
                           long int local_length,
                           long int global_length);
  ```

- `N_VNewEmpty_Parallel`

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

  ```
  N_Vector N_VNewEmpty_Parallel(MPI_Comm comm,
                                long int local_length,
                                long int global_length);
  ```

- `N_VMake_Parallel`

  This function creates and allocates memory for a parallel vector with user-provided data array.

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

- `N_VCloneVectorArray_Parallel`

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

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

- `N_VCloneVectorArrayEmpty_Parallel`

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

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

- `N_VDestroyVectorArray_Parallel`

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

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

- `N_VPrint_Parallel`

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

  `void N_VPrint_Parallel(N_Vector v);`

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the local component array via `v_data = NV_DATA_P(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_P(v,i)` within the loop.
- `N_VNewEmpty_Parallel`, `N_VMake_Parallel`, and `N_VCloneVectorArrayEmpty_Parallel` set the field *own_data* = `FALSE`. `N_VDestroy_Parallel` and `N_VDestroyVectorArray_Parallel` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.
- To maximize efficiency, vector operations in the NVECTOR_PARALLEL implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

7.3 NVECTOR functions used by CVODE

In Table 7.2 below, we list the vector functions in the NVECTOR module within the CVODE package. The table also shows, for each function, which of the code modules uses the function. The CVODE column shows function usage within the main integrator module, while the remaining seven columns show function usage within each of the six CVODE linear solvers (CVSPILS stands for any of CVSPGMR, CVSPBCG, or CVSPTFQMR), the CVBANDPRE and CVBBDPRE preconditioner modules, and the FCVODE module.

There is one subtlety in the CVSPILS column hidden by the table, explained here for the case of the CVSPGMR module). The dot product function `N_VDotProd` is called both within the implementation file `cvode_spgmr.c` for the CVSPGMR solver and within the implementation files `sundials_spgmr.c` and `sundials_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 `cvode_spgmr.c`, they are called within the implementation file `sundials_spgmr.c` and so are required by the CVSPGMR solver module. This issue does not arise for the other three CVODE linear solvers because the generic DENSE and BAND solvers (used in the implementation of CVDENSE and CVBAND) do not make calls to any vector functions and CVDIAG is not implemented using a generic diagonal solver.

At this point, we should emphasize that the CVODE user does not need to know anything about the usage of vector functions by the CVODE code modules in order to use CVODE. The information is presented as an implementation detail for the interested reader.

The vector functions listed in Table 7.1 that are *not* used by CVODE are: `N_VWL2Norm`, `N_VL1Norm`, `N_VWrmsNormMask`, `N_VConstrMask`, `N_VCloneEmpty`, and `N_VMinQuotient`. Therefore a user-supplied NVECTOR module for CVODE could omit these six functions.

Table 7.2: List of vector functions usage by CVODE code modules

	CVODE	CVDENSE	CVBAND	CVDIAG	CVSPILS	CVBANDPRE	CVBBDPRE	FCVODE
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				✓				

Chapter 8

Providing Alternate Linear Solver Modules

The central CVODE 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`: evaluate and preprocess the Jacobian or preconditioner;
- `lsolve`: solve the linear system;
- `lfree`: free the linear solver memory.

A linear solver module must also provide a user-callable specification routine (like those described in §5.5.3) which will attach the above four routines to the main CVODE memory block. The CVODE memory block is a structure defined in the header file `cvode_impl.h`. A pointer to such a structure is defined as the type `CVodeMem`. The four fields in a `CvodeMem` structure that must point to the linear solver's functions are `cv_linit`, `cv_lsetup`, `cv_lsolve`, and `cv_lfree`, respectively. Note that of the four interface routines, only the `lsolve` routine is required. The `lfree` routine must be provided only if the solver specification routine makes any memory allocation. For consistency with the existing CVODE linear solver modules, we recommend that the return value of the specification function be 0 for a successful return or a negative value if an error occurs (the pointer to the main CVODE memory block is `NULL`, an input is illegal, the NVECTOR implementation is not compatible, a memory allocation fails, etc.)

To facilitate data exchange between the four interface functions, the field `cv_lmem` in the CVODE memory block can be used to attach a linear solver-specific memory block.

These four routines that interface between CVODE and the linear solver module necessarily have fixed call sequences. Thus, a user wishing to implement another linear solver within the CVODE package must adhere to this set of interfaces. The following is a complete description of the call list for each of these routines. Note that the call list of each routine includes a pointer to the main CVODE memory block, by which the routine can access various data related to the CVODE solution. The contents of this memory block are given in the file `cvode_impl.h` (but not reproduced here, for the sake of space).

8.1 Initialization function

The type definition of `linit` is

`linit`

Definition `int (*linit)(CVodeMem cv_mem);`

Purpose The purpose of `linit` is to complete initializations for specific linear solver, such as counters and statistics.

Arguments `cv_mem` is the CVODE memory pointer of type `CVodeMem`.

Return value An `linit` function should return 0 if it has successfully initialized the CVODE linear solver and -1 otherwise.

8.2 Setup function

The type definition of `lsetup` is

`lsetup`

Definition
```
int (*lsetup)(CVodeMem cv_mem, int convfail, N_Vector ypred,
              N_Vector fpred, booleantype *jcurPtr,
              N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);
```

Purpose The job of `lsetup` is to prepare the linear solver for subsequent calls to `lsolve`. It may re-compute Jacobian-related data is it deems necessary.

Arguments
`cv_mem` is the CVODE memory pointer of type `CVodeMem`.

`convfail` is an input flag used to indicate any problem that occurred during the solution of the nonlinear equation on the current time step for which the linear solver is being used. This flag can be used to help decide whether the Jacobian data kept by a CVODE linear solver needs to be updated or not. Its possible values are:

- `CV_NO_FAILURES`: this value is passed to `lsetup` if either this is the first call for this step, or the local error test failed on the previous attempt at this step (but the Newton iteration converged).
- `CV_FAIL_BAD_J`: this value is passed to `lsetup` if (a) the previous Newton corrector iteration did not converge and the linear solver's setup 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.
- `CV_FAIL_OTHER`: this value is passed to `lsetup` if during the current internal step try, the previous Newton iteration failed to converge even though the linear solver was using current Jacobian-related data.

`ypred` is the predicted `y` vector for the current CVODE internal step.

`fpred` is the value of the right-hand side at `ypred`, 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 re-evaluation of Jacobian data (based on `convfail` and CVODE state data), it should return `*jcurPtr = TRUE` unconditionally; otherwise an infinite loop can result.

`vtemp1`
`vtemp2`
`vtemp3` are temporary variables of type `N_Vector` provided for use by `lsetup`.

Return value The `lsetup` routine should return 0 if successful, a positive value for a recoverable error, and a negative value for an unrecoverable error.

8.3 Solve function

The type definition of `lsolve` is

`lsolve`

Definition `int (*lsolve)(CVodeMem cv_mem, N_Vector b, N_Vector weight, N_Vector ycur, N_Vector fcur);`

Purpose The routine `lsolve` must solve the linear equation $Mx = b$, where M is some approximation to $I - \gamma J$, $J = (\partial f/\partial y)(t_n, y_{cur})$ (see Eq.(3.5)), and the right-hand side vector b is input.

Arguments
- `cv_mem` is the CVODE memory pointer of type `CVodeMem`.
- `b` is the right-hand side vector b. The solution is to be returned in the vector `b`.
- `weight` is a vector that contains the error weights. These are the W_i of Eq.(3.6).
- `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.

8.4 Memory deallocation function

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 CVODE 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 9

Generic Linear Solvers in SUNDIALS

In this chapter, we describe five generic linear solver code modules that are included in CVODE, but which are of potential use as generic packages in themselves, either in conjunction with the use of CVODE 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.
- The SPGMR package, which includes a solver for the scaled preconditioned GMRES method.
- The SPBCG package, which includes a solver for the scaled preconditioned Bi-CGStab method.
- The SPTFQMR package, which includes a solver for the scaled preconditioned TFQMR method.

For reasons related to installation, the names of the files involved in these generic solvers begin with the prefix `sundials_`. But despite this, each of the solvers is in fact generic, in that it is usable completely independently of SUNDIALS.

For the sake of space, the functions for `DenseMat` and `BandMat` matrices and the functions in SPGMR, SPBCG, and SPTFQMR are only summarized briefly, since they are less likely to be of direct use in connection with CVODE. 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 CVODE and the CVSPGMR, CVSPBCG, or CVSPTFQMR solver.

9.1 The DENSE module

Relative to the SUNDIALS *srcdir*, the files comprising the DENSE generic linear solver are as follows:

- header files (located in *srcdir*/`include/sundials`)
  `sundials_dense.h` `sundials_smalldense.h`
  `sundials_types.h` `sundials_math.h` `sundials_config.h`
- source files (located in *srcdir*/`src/sundials`)
  `sundials_dense.c` `sundials_smalldense.c` `sundials_math.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are relevant to the DENSE package by itself (see §2.4 for details):

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  `#define SUNDIALS_DOUBLE_PRECISION 1`
  `#define SUNDIALS_SINGLE_PRECISION 1`
  `#define SUNDIALS_EXTENDED_PRECISION 1`
- (optional) use of generic math functions: `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the `ABS` macro and `RAbs` function.

The eight files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into a DENSE library or into a larger user code.

9.1.1 Type DenseMat

The type `DenseMat` is defined to be a pointer to a structure with the number of rows, number of columns, and a data field:

```
typedef struct {
  long int M;
  long int N;
  realtype  **data;
} *DenseMat;
```

The M and N fields indicates the number of columns and rows, respectively, 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` $<$ `M` and $0 \leq$ `j` $<$ `N`) is given by the expression `(A->data)[j][i]` or by the expression `(A->data)[0][j*M+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.

9.1.2 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 $M \times N$ `DenseMat A`, $0 \leq$ `i` $< M$, $0 \leq$ `j` $< N$.

- `DENSE_COL`

  Usage : `col_j = DENSE_COL(A,j);`

  `DENSE_COL` references the `j`-th column of the $M \times N$ `DenseMat A`, $0 \leq$ `j` $< N$. The type of the expression `DENSE_COL(A,j)` is `realtype *` . After the assignment in the usage above, `col_j` may be treated as an array indexed from 0 to $M - 1$. The (`i, j`)-th element of `A` is referenced by `col_j[i]`.

9.1.3 Functions

The following functions for `DenseMat` matrices are available in the DENSE package. For full details, see the header file `sundials_dense.h`.

- `DenseAllocMat`: allocation of a `DenseMat` matrix;
- `DenseAllocPiv`: allocation of a pivot array for use with `DenseGETRF`/`DenseGETRS`;
- `DenseGETRF`: LU factorization with partial pivoting;
- `DenseGETRS`: solution of $Ax = b$ using LU factorization (for square matrices A);
- `DenseZero`: load a matrix with zeros;
- `DenseCopy`: copy one matrix to another;
- `DenseScale`: scale a matrix by a scalar;
- `DenseAddI`: increment a square 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.

9.1.4 Small Dense Matrix Functions

The following functions for small dense matrices are available in the DENSE package:

- `denalloc`

  `denalloc(m,n)` allocates storage for an `m` by `n` dense matrix. It returns a pointer to the newly allocated storage if successful. If the memory request cannot be satisfied, then `denalloc` returns `NULL`. The underlying type of the dense matrix returned is `realtype**`. If we allocate a dense matrix `realtype** a` by `a = denalloc(m,n)`, then `a[j][i]` references the (`i`,`j`)-th element of the matrix `a`, $0 \leq$ `i` $<$ `m`, $0 \leq$ `j` $< n$, and `a[j]` is a pointer to the first element in the `j`-th column of `a`. The location `a[0]` contains a pointer to `m` $\times$ `n` contiguous locations which contain the elements of `a`.

- `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.

- `denGETRF`

  `denGETRF(a,m,n,p)` factors the `m` by `n` dense matrix `a`, using Gaussian elimination with row pivoting. It overwrites the elements of `a` with its LU factors and keeps track of the pivot rows chosen in the pivot array `p`.

  A successful LU factorization leaves the matrix `a` and the pivot array `p` with the following information:

  1. `p[k]` contains the row number of the pivot element chosen at the beginning of elimination step `k`, `k` $= 0, 1, ...,$`n`-1.
  2. If the unique LU factorization of `a` is given by $Pa = LU$, where P is a permutation matrix, L is an `m` by `n` lower trapezoidal matrix with all diagonal elements equal to 1, and U is an `n` by `n` upper triangular matrix, then the upper triangular part of `a` (including its diagonal) contains U and the strictly lower trapezoidal part of `a` contains the multipliers, $I - L$. If `a` is square, L is a unit lower triangular matrix.

     `denGETRF` returns 0 if successful. Otherwise it encountered a zero diagonal element during the factorization, indicating that the matrix `a` does not have full column rank. In this case it returns the column index (numbered from one) at which it encountered the zero.

- `denGETRS`

  `denGETRS(a,n,p,b)` solves the `n` by `n` linear system $ax = b$. It assumes that `a` (of size `n` × `n`) has been LU-factored and the pivot array `p` has been set by a successful call to `denGETRF(a,n,n,p)`. The solution x is written into the `b` array.

- `denzero`

  `denzero(a,m,n)` sets all the elements of the `m` by `n` dense matrix `a` to be 0.0;

- `dencopy`

  `dencopy(a,b,m,n)` copies the `m` by `n` dense matrix `a` into the `m` by `n` dense matrix `b`;

- `denscale`

  `denscale(c,a,m,n)` scales every element in the `m` 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,m,n)` prints the `m` 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.

9.2 The BAND module

Relative to the SUNDIALS *srcdir*, the files comprising the BAND generic linear solver are as follows:

- header files (located in *srcdir*/`include/sundials`)
  `sundials_band.h`
  `sundials_types.h sundials_math.h sundials_config.h`
- source files (located in *srcdir*/`src/sundials`)
  `sundials_band.c sundials_math.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are required to use the BAND package by itself (see §2.4 for details):

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  `#define SUNDIALS_DOUBLE_PRECISION 1`
  `#define SUNDIALS_SINGLE_PRECISION 1`
  `#define SUNDIALS_EXTENDED_PRECISION 1`
- (optional) use of generic math functions:
  `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines of the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the `MIN`, `MAX`, and `ABS` macros and `RAbs` function.

The six files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into a BAND library or into a larger user code.

9.2.1 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, $0 \le mu \le size-1$;
- *ml* is the lower half-bandwidth, $0 \le ml \le size-1$;
- *smu* is the storage upper half-bandwidth, $mu \le smu \le size-1$. The `BandGBTRF` 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 `BandGBTRF`.

  – *data[j][i-j+smu]* is the (i,j)-th element, $j-mu \le i \le j+ml$.

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 9.1 for a diagram of the `BandMat` type.

9.2.2 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 $0 \le$ `i`, `j` $\le N-1$. The location (`i`,`j`) should further satisfy `j`−`(A->mu)` $\le$ `i` $\le$ `j`+`(A->ml)`.

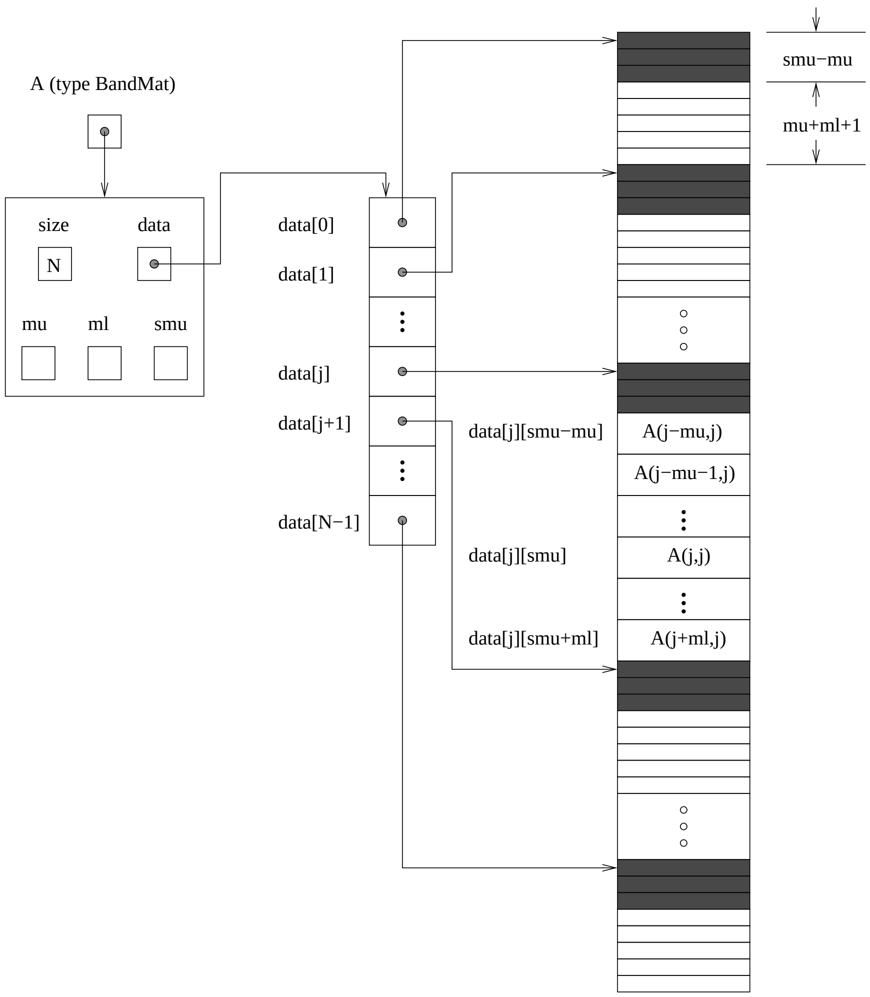


Figure 9.1: Diagram of the storage for a band matrix of type `BandMat`. Here `A` is an $N \times N$ band matrix of type `BandMat` with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to $N-1$ and the (i,j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the `BandGBTRF` and `BandGBTRS` 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)`.

9.2.3 Functions

The following functions for `BandMat` matrices are available in the BAND package. For full details, see the header file `sundials_band.h`.

- `BandAllocMat`: allocation of a `BandMat` matrix;
- `BandAllocPiv`: allocation of a pivot array for use with `BandGBTRF`/`BandGBTRS`;
- `BandGBTRF`: LU factorization with partial pivoting;
- `BandGBTRS`: 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.

9.3 The SPGMR module

The SPGMR package, in the files `sundials_spgmr.h` and `sundials_spgmr.c`, includes an implementation of the scaled preconditioned GMRES method. A separate code module, implemented in `sundials_iterative.(h,c)`, contains auxiliary functions that support SPGMR, as well as the other Krylov solvers in SUNDIALS (SPBCG and SPTFQMR). For full details, including usage instructions, see the header files `sundials_spgmr.h` and `sundials_iterative.h`.

Relative to the SUNDIALS *srcdir*, the files comprising the SPGMR generic linear solver are as follows:

- header files (located in *srcdir*`/include/sundials`)
  `sundials_spgmr.h` `sundials_iterative.h` `sundials_nvector.h`
  `sundials_types.h` `sundials_math.h` `sundials_config.h`
- source files (located in *srcdir*`/src/sundials`)
  `sundials_spgmr.c` `sundials_iterative.c` `sundials_nvector.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are required to use the SPGMR package by itself (see §2.4 for details):

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  `#define SUNDIALS_DOUBLE_PRECISION 1`
  `#define SUNDIALS_SINGLE_PRECISION 1`
  `#define SUNDIALS_EXTENDED_PRECISION 1`
- (optional) use of generic math functions:
  `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the `MAX` and `ABS` macros and `RAbs` and `RSqrt` functions.

The generic NVECTOR files, `sundials_nvector.(h,c)` are needed for the definition of the generic `N_Vector` type and functions. The NVECTOR functions used by the SPGMR module are: `N_VDotProd`, `N_VLinearSum`, `N_VScale`, `N_VProd`, `N_VDiv`, `N_VConst`, `N_VClone`, `N_VCloneVectorArray`, `N_VDestroy`, and `N_VDestroyVectorArray`.

! The SPGMR package can only be used in conjunction with an actual NVECTOR implementation library, such as the NVECTOR_SERIAL or NVECTOR_PARALLEL provided with SUNDIALS.

The nine files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into an SPGMR library or into a larger user code.

9.3.1 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 `sundials_iterative.(h,c)`:

- `ModifiedGS`: performs modified Gram-Schmidt procedure;
- `ClassicalGS`: performs classical Gram-Schmidt procedure;
- `QRfact`: performs QR factorization of Hessenberg matrix;
- `QRsol`: solves a least squares problem with a Hessenberg matrix factored by `QRfact`.

9.4 The SPBCG module

The SPBCG package, in the files `sundials_spbcgs.h` and `sundials_spbcgs.c`, includes an implementation of the scaled preconditioned Bi-CGStab method. For full details, including usage instructions, see the file `sundials_spbcgs.h`.

! The SPBCG package can only be used in conjunction with an actual NVECTOR implementation library, such as the NVECTOR_SERIAL or NVECTOR_PARALLEL provided with SUNDIALS.

The files needed to use the SPBCG module by itself are the same as for the SPGMR module, with `sundials_spbcgs.(h,c)` replacing `sundials_spgmr.(h,c)`.

9.4.1 Functions

The following functions are available in the SPBCG package:

- `SpbcgMalloc`: allocation of memory for `SpbcgSolve`;
- `SpbcgSolve`: solution of $Ax = b$ by the SPBCG method;
- `SpbcgFree`: free memory allocated by `SpbcgMalloc`.

9.5 The SPTFQMR module

The SPTFQMR package, in the files sundials_sptfqmr.h and sundials_sptfqmr.c, includes an implementation of the scaled preconditioned TFQMR method. For full details, including usage instructions, see the file sundials_sptfqmr.h.

The SPTFQMR package can only be used in conjunction with an actual NVECTOR implementation library, such as the NVECTOR_SERIAL or NVECTOR_PARALLEL provided with SUNDIALS.

The files needed to use the SPTFQMR module by itself are the same as for the SPGMR module, with sundials_sptfqmr.(h,c) replacing sundials_spgmr.(h,c).

9.5.1 Functions

The following functions are available in the SPTFQMR package:

- SptfqmrMalloc: allocation of memory for SptfqmrSolve;
- SptfqmrSolve: solution of $Ax = b$ by the SPTFQMR method;
- SptfqmrFree: free memory allocated by SptfqmrMalloc.

Chapter 10

CVODE Constants

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

10.1 CVODE input constants

CVODE **main solver module**		
CV_ADAMS	1	Adams-Moulton linear multistep method.
CV_BDF	2	BDF linear multistep method.
CV_FUNCTIONAL	1	Nonlinear system solution through functional iterations.
CV_NEWTON	2	Nonlinear system solution through Newton iterations.
CV_SS	1	Scalar relative tolerance, scalar absolute tolerance.
CV_SV	2	Scalar relative tolerance, vector absolute tolerance.
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.

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.

10.2 CVODE output constants

CVODE **main solver module**		
CV_SUCCESS	0	Successful function return.
CV_TSTOP_RETURN	1	CVode succeeded by reaching the specified stopping point.
CV_ROOT_RETURN	2	CVode succeeded and found one or more roots.

CV_TOO_MUCH_WORK	-1	The solver took mxstep internal steps but could not reach tout.
CV_TOO_MUCH_ACC	-2	The solver could not satisfy the accuracy demanded by the user for some internal step.
CV_ERR_FAILURE	-3	Error test failures occurred too many times during one internal time step or minimum step size was reached.
CV_CONV_FAILURE	-4	Convergence test failures occurred too many times during one internal time step or minimum step size was reached.
CV_LINIT_FAIL	-5	The linear solver's initialization function failed.
CV_LSETUP_FAIL	-6	The linear solver's setup function failed in an unrecoverable manner.
CV_LSOLVE_FAIL	-7	The linear solver's solve function failed in an unrecoverable manner.
CV_RHSFUNC_FAIL	-8	The right-hand side function failed in an unrecoverable manner.
CV_FIRST_RHSFUNC_ERR	-9	The right-hand side function failed at the first call.
CV_REPTD_RHSFUNC_ERR	-10	The right-hand side function had repetead recoverable errors.
CV_UNREC_RHSFUNC_ERR	-11	The right-hand side function had a recoverable error, but no recovery is possible.
CV_RTFUNC_FAIL	-12	The rootfinding function failed in an unrecoverable manner.
CV_MEM_FAIL	-20	A memory allocation failed.
CV_MEM_NULL	-21	The cvode_mem argument was NULL.
CV_ILL_INPUT	-22	One of the function inputs is illegal.
CV_NO_MALLOC	-23	The CVODE memory block was not allocated by a call to CVodeMalloc.
CV_BAD_K	-24	The derivative order k is larger than the order used.
CV_BAD_T	-25	The time t s outside the last step taken.
CV_BAD_DKY	-26	The output derivative vector is NULL.
CV_TOO_CLOSE	-27	The output and initial times are too close to each other.

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.
CVDENSE_JACFUNC_UNRECVR	-5	The Jacobian function failed in an unrecoverable manner.
CVDENSE_JACFUNC_RECVR	-6	The Jacobian function had a recoverable error.

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, or an input value was illegal.

CVBAND_MEM_FAIL	-4	A memory allocation request failed.
CVBAND_JACFUNC_UNRECVR	-5	The Jacobian function failed in an unrecoverable manner.
CVBAND_JACFUNC_RECVR	-6	The Jacobian function had a recoverable error.

CVDIAG linear solver module

CVDIAG_SUCCESS	0	Successful function return.
CVDIAG_MEM_NULL	-1	The cvode_mem argument was NULL.
CVDIAG_LMEM_NULL	-2	The CVDIAG linear solver has not been initialized.
CVDIAG_ILL_INPUT	-3	The CVDIAG solver is not compatible with the current NVECTOR module.
CVDIAG_MEM_FAIL	-4	A memory allocation request failed.

CVSPILS linear solver modules

CVSPILS_SUCCESS	0	Successful function return.
CVSPILS_MEM_NULL	-1	The cvode_mem argument was NULL.
CVSPILS_LMEM_NULL	-2	The linear solver has not been initialized.
CVSPILS_ILL_INPUT	-3	The solver is not compatible with the current NVECTOR module, or an input value was illegal.
CVSPILS_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_ATIMES_FAIL_REC	5	The Jacobian-times-vector function failed recoverably.
SPGMR_PSET_FAIL_REC	6	The preconditioner setup function failed recoverably.
SPGMR_MEM_NULL	-1	The SPGMR memory is NULL
SPGMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPGMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPGMR_GS_FAIL	-4	Failure in the Gram-Schmidt procedure.
SPGMR_QRSOL_FAIL	-5	The matrix R was found to be singular during the QR solve phase.
SPGMR_PSET_FAIL_UNREC	-6	The preconditioner setup function failed unrecoverably.

SPBCG generic linear solver module

SPBCG_SUCCESS	0	Converged.
SPBCG_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPBCG_CONV_FAIL	2	Failure to converge.
SPBCG_PSOLVE_FAIL_REC	3	The preconditioner solve function failed recoverably.
SPBCG_ATIMES_FAIL_REC	4	The Jacobian-times-vector function failed recoverably.
SPBCG_PSET_FAIL_REC	5	The preconditioner setup function failed recoverably.
SPBCG_MEM_NULL	-1	The SPBCG memory is NULL

SPBCG_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPBCG_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPBCG_PSET_FAIL_UNREC	-4	The preconditioner setup function failed unrecoverably.

SPTFQMR generic linear solver module

SPTFQMR_SUCCESS	0	Converged.
SPTFQMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPTFQMR_CONV_FAIL	2	Failure to converge.
SPTFQMR_PSOLVE_FAIL_REC	3	The preconditioner solve function failed recoverably.
SPTFQMR_ATIMES_FAIL_REC	4	The Jacobian-times-vector function failed recoverably.
SPTFQMR_PSET_FAIL_REC	5	The preconditioner setup function failed recoverably.
SPTFQMR_MEM_NULL	-1	The SPTFQMR memory is NULL
SPTFQMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed.
SPTFQMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPTFQMR_PSET_FAIL_UNREC	-4	The preconditioner setup function failed unrecoverably.

CVBANDPRE preconditioner module

CVBANDPRE_SUCCESS	0	Successful function return.
CVBANDPRE_PDATA_NULL	-11	The preconditioner module has not been initialized.
CVBANDPRE_RHSFUNC_UNRECVR	-12	The right-hand side function failed unrecoverably.

CVBBDPRE preconditioner module

CVBBDPRE_SUCCESS	0	Successful function return.
CVBBDPRE_PDATA_NULL	-11	The preconditioner module has not been initialized.
CVBBDPRE_FUNC_UNRECVR	-12	A user supplied function failed unrecoverably.

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. User Documentation for PVODE, An ODE Solver for Parallel Computers. Technical Report UCRL-ID-130884, LLNL, May 1998.

[6] 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.

[7] S. D. Cohen and A. C. Hindmarsh. CVODE User Guide. Technical Report UCRL-MA-118618, LLNL, September 1994.

[8] S. D. Cohen and A. C. Hindmarsh. CVODE, a Stiff/Nonstiff ODE Solver in C. *Computers in Physics*, 10(2):138–143, 1996.

[9] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.

[10] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.

[11] 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.

[12] A. C. Hindmarsh. Avoiding BDF Stability Barriers in the MOL Solution of Advection-Dominated Problems. *Appl. Num. Math.*, 17:311–318, 1995.

[13] A. C. Hindmarsh. The PVODE and IDA Algorithms. Technical Report UCRL-ID-141558, LLNL, December 2000.

[14] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS, suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, (31):363–396, 2005.

[15] A. C. Hindmarsh and R. Serban. Example Programs for CVODE v2.4.0. Technical report, LLNL, 2005. UCRL-SM-208110.

[16] 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.

[17] 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.

[18] 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.

[19] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.

[20] H. A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.

Index

Adams method, 15
ADAMS_Q_MAX, 40

BAND generic linear solver
  functions, 121
  macros, 119–121
  type BandMat, 119
BAND_COL, 67, 121
BAND_COL_ELEM, 67, 121
BAND_ELEM, 67, 119
BandMat, 29, 66, 119
BDF method, 15
BDF_Q_MAX, 40
Bi-CGStab method, 35, 47, 122
BIG_REAL, 28, 105

CLASSICAL_GS, 47
CV_ADAMS, 31, 62
CV_BAD_DKY, 48
CV_BAD_K, 48
CV_BAD_T, 48
CV_BDF, 31, 62
CV_CONV_FAILURE, 37
CV_ERR_FAILURE, 37
CV_FIRST_RHSFUNC_ERR, 64
CV_FIRST_RHSFUNC_FAIL, 37
CV_FUNCTIONAL, 31, 43
CV_ILL_INPUT, 32, 36, 40–44, 63
CV_LINIT_FAIL, 37
CV_LSETUP_FAIL, 37, 65, 66, 76
CV_LSOLVE_FAIL, 37
CV_MEM_FAIL, 32
CV_MEM_NULL, 32, 36, 38, 40–44, 48, 50–55, 63, 70
CV_NEWTON, 31, 43
CV_NO_MALLOC, 36, 63
CV_NORMAL, 36
CV_NORMAL_TSTOP, 36
CV_ONE_STEP, 36
CV_ONE_STEP_TSTOP, 36
CV_REPTD_RHSFUNC_ERR, 37
CV_RHSFUNC_FAIL, 37, 64
CV_ROOT_RETURN, 36
CV_RTFUNC_FAIL, 37, 71
CV_SS, 31, 43, 63
CV_SUCCESS, 32, 36, 38, 40–44, 48, 50–55, 63, 70, 74
CV_SV, 31, 43, 63
CV_TOO_CLOSE, 37
CV_TOO_MUCH_ACC, 37
CV_TOO_MUCH_WORK, 37
CV_TSTOP_RETURN, 36
CV_UNREC_RHSFUNC_ERR, 37, 64
CV_WARNING, 64
CV_WF, 31, 63
CVBAND linear solver
  Jacobian approximation used by, 45
  memory requirements, 57
  NVECTOR compatibility, 34
  optional input, 45
  optional output, 57–58
  selection of, 34
  use in FCVODE, 87
CVBand, 30, 33, 34, 66
CVBAND_ILL_INPUT, 34
CVBAND_JACFUNC_RECVR, 66
CVBAND_JACFUNC_UNRECVR, 66
CVBAND_LMEM_NULL, 45, 57, 58
CVBAND_MEM_FAIL, 34
CVBAND_MEM_NULL, 34, 45, 57, 58
CVBAND_SUCCESS, 34, 45, 57, 58
CVBandDQJac, 45
CVBandGetLastFlag, 58
CVBandGetNumJacEvals, 57
CVBandGetNumRhsEvals, 57
CVBandGetReturnFlagName, 58
CVBandGetWorkSpace, 57
CVBandJacFn, 66
CVBANDPRE preconditioner
  description, 71
  optional output, 74
  usage, 71–72
  user-callable functions, 72–74
CVBANDPRE_PDATA_NULL, 73, 74
CVBANDPRE_SUCCESS, 74
CVBandPrecAlloc, 72
CVBandPrecFree, 74
CVBandPrecGetNumRhsEvals, 74
CVBandPrecGetReturnFlagName, 74
CVBandPrecGetWorkSpace, 74
CVBandSetJacFn, 45
CVBBDPRE preconditioner

description, 75
optional output, 80–81
usage, 77–78
user-callable functions, 78–80
user-supplied functions, 76–77
CVBBDPRE_PDATA_NULL, 79–81
CVBBDPRE_SUCCESS, 80, 81
CVBBDPrecAlloc, 78
CVBBDPrecFree, 79
CVBBDPrecGetNumGfnEvals, 81
CVBBDPrecGetReturnFlagName, 81
CVBBDPrecGetWorkSpace, 80
CVBBDPrecReInit, 80
CVBBDSpbcg, 79
CVBBDSpgmr, 78
CVBBDSptfqmr, 79
CVBPSpbcg, 73
CVBPSpgmr, 72
CVBPSptfqmr, 73
CVDENSE linear solver
Jacobian approximation used by, 44
memory requirements, 55
NVECTOR compatibility, 33
optional input, 44–45
optional output, 55–57
selection of, 33
use in FCVODE, 87
CVDense, 30, 33, 65
CVDENSE_ILL_INPUT, 34
CVDENSE_JACFUNC_RECVR, 65
CVDENSE_JACFUNC_UNRECVR, 65
CVDENSE_LMEM_NULL, 45, 55, 56
CVDENSE_MEM_FAIL, 34
CVDENSE_MEM_NULL, 34, 45, 55, 56
CVDENSE_SUCCESS, 34, 45, 55, 56
CVDenseDQJac, 44
CVDenseGetLastFlag, 56
CVDenseGetNumJacEvals, 56
CVDenseGetNumRhsEvals, 56
CVDenseGetReturnFlagName, 57
CVDenseGetWorkSpace, 55
CVDenseJacFn, 65
CVDenseSetJacFn, 44
CVDIAG linear solver
Jacobian approximation used by, 34
memory requirements, 58
optional output, 58–59
selection of, 34
use in FCVODE, 86
CVDiag, 30, 33, 34
CVDIAG_ILL_INPUT, 34
CVDIAG_LMEM_NULL, 58, 59
CVDIAG_MEM_FAIL, 34
CVDIAG_MEM_NULL, 34, 58, 59
CVDIAG_SUCCESS, 34, 58, 59
CVDiagGetLastFlag, 59
CVDiagGetNumRhsEvals, 59
CVDiagGetReturnFlagName, 59
CVDiagGetWorkSpace, 58
CVErrHandlerFn, 64
CVEwtFn, 64
CVODE, 1
motivation for writing in C, 2
package structure, 23
relationship to CVODE, PVODE, 1–2
relationship to VODE, VODPK, 1
CVODE linear solvers
built on generic solvers, 33
CVBAND, 34
CVDENSE, 33
CVDIAG, 34
CVSPBCG, 35
CVSPGMR, 35
CVSPTFQMR, 35
header files, 28
implementation details, 25
list of, 23–25
NVECTOR compatibility, 27
selecting one, 33
CVode, 30, 36
cvode.h, 28
cvode_band.h, 29
cvode_dense.h, 28
cvode_diag.h, 29
cvode_spbcgs.h, 29
cvode_spgmr.h, 29
cvode_sptfqmr.h, 29
CVodeCreate, 31
CVodeFree, 31, 32
CVodeGetActualInitStep, 52
CVodeGetCurrentOrder, 52
CVodeGetCurrentStep, 52
CVodeGetCurrentTime, 53
CVodeGetDky, 48
CVodeGetErrWeights, 53
CVodeGetEstLocalErrors, 54
CVodeGetIntegratorStats, 54
CVodeGetLastOrder, 51
CVodeGetLastStep, 52
CVodeGetNonlinSolvStats, 55
CVodeGetNumErrTestFails, 51
CVodeGetNumGEvals, 70
CVodeGetNumLinSolvSetups, 51
CVodeGetNumNonlinSolvConvFails, 55
CVodeGetNumNonlinSolvIters, 54
CVodeGetNumRhsEvals, 51
CVodeGetNumStabLimOrderReds, 53
CVodeGetNumSteps, 50

CVodeGetReturnFlagName, 55
CVodeGetRootInfo, 70
CVodeGetTolScaleFactor, 53
CVodeGetWorkSpace, 50
CVodeMalloc, 31, 62
CVodeReInit, 62
CVodeRootInit, 70
CVodeSetErrFile, 38
CVodeSetErrHandlerFn, 38
CVodeSetEwtFn, 44
CVodeSetFdata, 38
CVodeSetInitStep, 41
CVodeSetIterType, 43
CVodeSetMaxConvFails, 43
CVodeSetMaxErrTestFails, 42
CVodeSetMaxHnilWarns, 40
CVodeSetMaxNonlinIters, 42
CVodeSetMaxNumSteps, 40
CVodeSetMaxOrder, 40
CVodeSetMaxStep, 41
CVodeSetMinStep, 41
CVodeSetNonlinConvCoef, 43
CVodeSetStabLimDet, 41
CVodeSetStopTime, 42
CVodeSetTolerances, 43
CVRhsFn, 31, 63
CVRootFn, 70
CVSPBCG linear solver
  Jacobian approximation used by, 45
  memory requirements, 59
  optional input, 45–48
  optional output, 59–62
  preconditioner setup function, 45, 68
  preconditioner solve function, 45, 68
  selection of, 35
  use in FCVODE, 88
CVSpbcg, 30, 33, 35
CVSPGMR linear solver
  Jacobian approximation used by, 45
  memory requirements, 59
  optional input, 45–48
  optional output, 59–62
  preconditioner setup function, 45, 68
  preconditioner solve function, 45, 68
  selection of, 35
  use in FCVODE, 88
CVSpgmr, 30, 33, 35
CVSPILS_ILL_INPUT, 35, 36, 47, 73, 79
CVSPILS_LMEM_NULL, 46–48, 60–62
CVSPILS_MEM_FAIL, 35, 36, 73, 79
CVSPILS_MEM_NULL, 35, 36, 46–48, 60–62, 73, 79
CVSPILS_SUCCESS, 35, 36, 46–48, 60–62, 73, 79
CVSpilsDQJtimes, 45
CVSpilsGetLastFlag, 62
CVSpilsGetNumConvFails, 60
CVSpilsGetNumJtimesEvals, 61
CVSpilsGetNumLinIters, 60
CVSpilsGetNumPrecEvals, 60
CVSpilsGetNumPrecSolves, 61
CVSpilsGetNumRhsEvals, 61
CVSpilsGetReturnFlagName, 62
CVSpilsGetWorkSpace, 60
CVSpilsJacTimesVecFn, 67
CVSpilsPrecSetupFn, 68
CVSpilsPrecSolveFn, 68
CVSpilsSetDelt, 47
CVSpilsSetGSType, 47
CVSpilsSetJacTimesFn, 46
CVSpilsSetMaxl, 47
CVSpilsSetPreconditioner, 46
CVSpilsSetPrecType, 46
CVSPTFQMR linear solver
  Jacobian approximation used by, 45
  memory requirements, 59
  optional input, 45–48
  optional output, 59–62
  preconditioner setup function, 45, 68
  preconditioner solve function, 45, 68
  selection of, 35
  use in FCVODE, 89
CVSptfqmr, 30, 33, 35

denaddI, 118
denalloc, 117
denallocpiv, 117
dencopy, 118
denfree, 118
denfreepiv, 118
denGETRF, 117
denGETRS, 118
denprint, 118
denscale, 118
DENSE generic linear solver
  functions
    large matrix, 116–117
    small matrix, 117–118
  macros, 116
  type DenseMat, 116
DENSE_COL, 65, 116
DENSE_ELEM, 65, 116
DenseMat, 28, 65, 116
denzero, 118

e_data, 65
eh_data, 64
error control
  order selection, 18
  step size selection, 17–18

error messages, 38
redirecting, 38
user-defined handler, 38, 64

f_data, 38, 64, 76
FCVBAND, 87
FCVBANDSETJAC, 88
FCVBBDFREE, 98
FCVBBDINIT, 97
FCVBBDOPT, 98
FCVBBDREINIT, 98
FCVBBDSPBCG, 97
FCVBBDSPGMR, 97
FCVBBDSPTFQMR, 97
FCVBJAC, 87
FCVBPFREE, 96
FCVBPINIT, 95
FCVBPOPT, 96
FCVBPSPBCG, 95
FCVBPSPGMR, 95
FCVBPSPTFQMR, 95
FCVCOMMFN, 98
FCVDENSE, 87
FCVDENSESETJAC, 87
FCVDIAG, 86
FCVDJAC, 87
FCVDKY, 91
FCVEWT, 86
FCVEWTSET, 86
FCVFREE, 92
FCVFUN, 85
FCVGETERRWEIGHTS, 92
FCVGETESTLOCALERR, 94
FCVGLOCFN, 98
FCVJTIMES, 89, 99
FCVMALLOC, 86
FCVMALLOC, 85
FCVODE, 90
FCVODE interface module
interface to the CVBANDPRE module, 95–96
interface to the CVBBDPRE module, 96–98
optional input and output, 92
rootfinding, 94
usage, 84–92
user-callable functions, 83–84
user-supplied functions, 84
FCVPSET, 90
FCVPSOL, 89
FCVREINIT, 91
FCVSETIIN, 92
FCVSETRIN, 92
FCVSPBCG, 88
FCVSPBCGREINIT, 91
FCVSPGMR, 88
FCVSPGMRREINIT, 91
FCVSPILSSETJAC, 89, 96, 97
FCVSPILSSETPREC, 89
FCVSPTFQMR, 89
FCVSPTFQMRREINIT, 91
FNVINITP, 85
FNVINITS, 85

g_data, 71
generic linear solvers
BAND, 118
DENSE, 115
SPBCG, 122
SPGMR, 121
SPTFQMR, 123
use in CVODE, 25
GMRES method, 35, 121
Gram-Schmidt procedure, 47

half-bandwidths, 34, 66–67, 72, 78
header files, 28, 71, 77
HNIL_WARNS, 92

INIT_STEP, 92
IOUT, 92, 93
itask, 30, **36**
iter, 31, 43
itol, 31, 43, 63

Jacobian approximation function
band
difference quotient, 45
use in FCVODE, 87
user-supplied, 45, 66–67
dense
difference quotient, 44
use in FCVODE, 87
user-supplied, 44, 65–66
diagonal
difference quotient, 34
Jacobian times vector
difference quotient, 45
use in FCVODE, 89
user-supplied, 46, 67–68

linit, 111
lmm, 31, 62
LSODE, 1

MAX_CONVFAIL, 92
MAX_ERRFAIL, 92
MAX_NITERS, 92
MAX_NSTEPS, 92
MAX_ORD, 92
MAX_STEP, 92

maxl, 35, 36, 73, 78, 79
maxord, 40, 62
memory requirements
  CVBAND linear solver, 57
  CVBANDPRE preconditioner, 74
  CVBBDPRE preconditioner, 80
  CVDENSE linear solver, 55
  CVDIAG linear solver, 58
  CVODE solver, 50
  CVSPGMR linear solver, 59
MIN_STEP, 92
MODIFIED_GS, 47
MPI, 3

N_VCloneEmptyVectorArray, 102
N_VCloneVectorArray, 102
N_VCloneVectorArray_Parallel, 108
N_VCloneVectorArray_Serial, 106
N_VCloneVectorArrayEmpty_Parallel, 108
N_VCloneVectorArrayEmpty_Serial, 106
N_VDestroyVectorArray, 102
N_VDestroyVectorArray_Parallel, 109
N_VDestroyVectorArray_Serial, 106
N_Vector, 28, 101
N_VMake_Parallel, 108
N_VMake_Serial, 106
N_VNew_Parallel, 108
N_VNew_Serial, 106
N_VNewEmpty_Parallel, 108
N_VNewEmpty_Serial, 106
N_VPrint_Parallel, 109
N_VPrint_Serial, 106
NLCONV_COEF, 92
nonlinear system
  definition, 15
  Newton convergence test, 17
  Newton iteration, 16–17
NV_COMM_P, 108
NV_CONTENT_P, 107
NV_CONTENT_S, 105
NV_DATA_P, 107
NV_DATA_S, 105
NV_GLOBLENGTH_P, 107
NV_Ith_P, 108
NV_Ith_S, 106
NV_LENGTH_S, 105
NV_LOCLENGTH_P, 107
NV_OWN_DATA_P, 107
NV_OWN_DATA_S, 105
NVECTOR module, 101
nvector_parallel.h, 28
nvector_serial.h, 28

optional input
  band linear solver, 45
  dense linear solver, 44–45
  iterative linear solver, 45–48
  solver, 38–44
optional output
  band linear solver, 57–58
  band-block-diagonal preconditioner, 80–81
  banded preconditioner, 74
  dense linear solver, 55–57
  diagonal linear solver, 58–59
  interpolated solution, 48
  iterative linear solver, 59–62
  solver, 50–55
output mode, 18, 36

portability, 28
  Fortran, 84
PREC_BOTH, 35, 36, 46
PREC_LEFT, 35, 36, 46, 73, 78, 79
PREC_NONE, 35, 36, 46
PREC_RIGHT, 35, 36, 46, 73, 78, 79
preconditioning
  advice on, 18–19, 25
  band-block diagonal, 75
  banded, 71
  setup and solve phases, 25
  user-supplied, 45–46, 68
pretype, 35, 36, 46, 73, 78, 79
PVODE, 1

RCONST, 28
realtype, 28
reinitialization, 62
right-hand side function, 63
Rootfinding, 20, 30, 69, 94
ROUT, 92, 93

SMALL_REAL, 28
SPBCG generic linear solver
  description of, 122
  functions, 122
SPGMR generic linear solver
  description of, 121
  functions, 122
  support functions, 122
SPTFQMR generic linear solver
  description of, 123
  functions, 123
STAB_LIM, 92
Stability limit detection, 19
step size bounds, 41–42
STOP_TIME, 92
SUNDIALS_CASE_LOWER, 84
SUNDIALS_CASE_UPPER, 84
sundials_nvector.h, 28

sundials_types.h, 28
SUNDIALS_UNDERSCORE_NONE, 84
SUNDIALS_UNDERSCORE_ONE, 84
SUNDIALS_UNDERSCORE_TWO, 84

TFQMR method, 36, 47, 123
tolerances, 16, 32, 44, 64

UNIT_ROUNDOFF, 28
User main program
  CVBANDPRE usage, 71
  CVBBDPRE usage, 77
  CVODE usage, 29
  FCVBBD usage, 97
  FCVBP usage, 95
  FCVODE usage, 85

VODE, 1
VODPK, 1

weighted root-mean-square norm, 16