This packet contains the documentation for one large library we shall use in the course and information on obtaining documentation for some others. I have also appended draft manual pages for my rabsonlib package of C library routines.

CVODE

CVODE is a library for solving ordinary differential equations; this packet reproduces the file /usr/local/doc/sundials/cv_guide.pdf, which came with the distribution from http://www.llnl.gov/casc/sundials. (An older version is available on Netlib.)

The program and documentation are copyrighted by the Regents of the University of California and distributed freely under the BSD license, which is reproduced following this page.

MPI

We shall use the Message-Passing Interface (MPI) for inter-node communication and synchronization in the final project, a molecular-dynamics simulation of crack propagation in a solid. The 231-page manual is available at http://www.mpi-forum.org/docs/docs.html (read the version 1.1 documentation; the version 2.0 document, which is an additional 361 pages, assumes everything in 1.1). Chapter 3 is reproduced here; note the copyright notice on p. 2 of the document: “Permission to copy without fee all or part of this material is granted, provided the University of Tennessee copyright notice and the title of this document appear, and notice is given that copying is by permission of the University of Tennessee.” Additional documentation on the MPICH implementation of MPI is available at http://www-unix.mcs.anl.gov/mpi/mpich. There’s a 15-page MPI Reference Card floating around on the network, e.g., at http://www.math.uic.edu/~hanson/mcs572/mpich96refcard.pdf, apparently from the first edition of Gropp, Lusk, and Skjellum, Using MPI: portable programming with the message-passing interface. Dr. Gropp offered permission for me to reproduce it for the course, but I noticed that it uses old-style C declarations and (more substantively) that the second edition of the book deprecates some of the function calls from the old reference card. The second edition is available at the library or from USF computers at http://www.netlibrary.com/urlapi.asp?action=summary&v=1&bookid=18119.

NETLIB (for information only; no documents reprinted here)

The netlib repository holds thousands of free and useful codes for scientific and engineering computing; among the routines we shall use are singleton (for the fast Fourier transform), dsyevd (for diagonalizing a real, symmetric matrix), and zheev (for diagonalizing a complex, Hermitian matrix). These latter two come from lapack, the merger of two huge linear-algebra packages previously called linpack and eispack. Documentation is available on physics with the man command. MPI and CVODE are also redistributed on NETLIB. Programs are available by ftp at ftp.netlib.org and on the Web at http://www.netlib.org.

RABSONLIB

The rabsonlib library contains some subroutines that I use in nearly every C program I write. It is located on physics in /home/5156/rabsonlib. Some are fairly old (1980’s). Consider the library a work in progress. You are free to link to these subroutines or to copy them, if you wish, provided you indicate in any derivative work that I wrote them. The public license to use these subroutines is non-exclusive, and I retain copyright interest in them. They carry no warranty.

MPI AND SGE ON IRCE

Finally, I’ve written a four-page tutorial on running parallel jobs on IRCE.
SUNDIALS Software Release Statement

PLEASE READ THIS SOFTWARE RELEASE STATEMENT BEFORE PRESSING THE "I AGREE" BUTTON BELOW. IF YOU DO NOT AGREE TO THE RESTRICTIONS LISTED BELOW, DO NOT PRESS "I AGREE."

BSD License

Copyright © 2002, The Regents of the University of California. 
Produced at the Lawrence Livermore National Laboratory. 
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the disclaimer below.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the disclaimer (as noted below) in the documentation and/or other materials provided with the distribution.
3. Neither the name of the UC/LLNL nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OF THE UNIVERSITY OF CALIFORNIA, THE U.S. DEPARTMENT OF ENERGY OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Additional BSD Notice

1. This notice is required to be provided under our contract with the U.S. Department of Energy (DOE). This work was produced at the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-ENG-48 with the DOE.
2. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights.
3. Also, reference herein to any specific commercial products, process, or services by trade name, trademark, manufacturer or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or 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.

I agree  I disagree
User Documentation for CVODE v2.4.0

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

March 24, 2006

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
9.5 The SPTFQMR module ........................................... 121
  9.5.1 Functions ............................................. 121
10 CVODE Constants ............................................... 123
  10.1 CVODE input constants .................................... 123
  10.2 CVODE output constants .................................. 123
Bibliography .................................................................. 127
Index ......................................................................... 129
List of Tables

2.1 SUNDIALS libraries and header files (names are relative to libdir for libraries and to incdir for header files) ........................................................................................................... 11

5.1 Optional inputs for cvode, cvdense, cvband, and cvspils .................................................................... 37

5.2 Optional outputs from cvode, cvdense, cvband, cvdiag, and cvspils ................................................. 47

6.1 Keys for setting fcvode optional inputs ............................................................................................... 90

6.2 Description of the fcvode optional output arrays IOUT and ROUT ..................................................... 91

7.1 Description of the NVECTOR operations ............................................................................................ 101

7.2 List of vector functions usage by cvode code modules ..................................................................... 108
List of Figures

4.1 Organization of the SUNDIALS suite ........................................ 22
4.2 Overall structure diagram of the cvode package ........................ 23
9.1 Diagram of the storage for a matrix of type SandMat .................. 118
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 those 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. External, cvode is very similar to the well known solver lsode [28]. vodpk is a variant of vode that uses a preconditioned Krylov (iterative) method, namely GMRES, for the solution of the linear systems. cvode 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 conjunction with Newton iteration: Bi-CGStab (Bi-Conjugate Gradient Stabilized) [28], 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-CGFSstab 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 multiprocessor 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 a 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 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 v2.4.0

cvpsirc and cvstpqmr modules have been added to interface with the Scaled Preconditioned Bi-CGstab (spbicg) 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 pvvode. 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 §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 input and output arrays. Instead, cvode now provides a set of routines (with prefix CvodeGet) 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 systems. 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 (§6.8), (c) do multiple runs of problems of the same size (§5.5.8), (d) supply a new

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 cvd Dense, 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

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

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

from within the sundials directory.

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

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

```
% rm -rf sundials
```

from a shell command prompt.

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

### 2.1 Installation steps

To install the SUNDIALS suite, given a downloaded file named sundials_file.tar.gz, issue the following commands from a shell command prompt, while within the directory where source_tree is to be located.

1. `gunzip sundials_file.tar.gz`
2. `tar -xvf sundials_file.tar` (creates sundials directory)
3. `cd build_tree`
4. `path/to/source_tree/configure` `options` (options can be absent)
5. `make`
6. `make install`
7. `make examples`
8. If system storage space conservation is a priority, then issue
   `make clean`
   and/or
   `make examples_clean`
   from a shell command prompt to remove unused object files.

### 2.2 Configuration options

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

#### General options

- `--prefix=PREFIX`
  Location for architecture-independent files.
  Default: `PREFIX=build_tree`
- `--includedir=DIR`
  Alternate location for installation of header files.
  Default: `DIR=PREFIX/include`
- `--libdir=DIR`
  Alternate location for installation of libraries.
  Default: `DIR=PREFIX/lib`
- `--disable-examples`
  All available example programs are automatically built unless this option is given. The example executables are stored under the following subdirectories of the associated solver:
- `build_tree/solver/examples_par`: serial C examples
- `build_tree/solver/examples_par`: parallel C examples (MPI-enabled)
- `build_tree/solver/fcmix/examples_par`: serial FORTRAN examples
- `build_tree/solver/fcmix/examples_par`: parallel FORTRAN examples (MPI-enabled)

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

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

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

--with-cflags=ARG
Specify additional C compilation flags.

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

--with-lib=ARG
Specify additional libraries to be used (e.g., ARG=-L<lib_dir> or libfoo.a 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-f77
Using this option will disable all Fortran support. The cvode, fkin sol, fida, and fnv vector modules will not be built, regardless of availability.

--with-ffflags=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 cvode, fkin sol, fida, and fnv vector 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

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.

--disable-static
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.

8 CVODE Installation Procedure

--with-f77case=ARG
Use this option to specify whether the external names of the cvode, fkin sol, fida, and fnv vector 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, AND-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-include=INCDir
--with-mpi-library=LIBDir
--with-mpi-flags=ARG
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 compiler 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 F ortran support

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

--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, AND-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-include=INCDir
--with-mpi-library=LIBDir
--with-mpi-flags=ARG
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 compiler 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 F ortran support

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

--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, AND-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-include=INCDir
--with-mpi-library=LIBDir
--with-mpi-flags=ARG
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 compiler 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 F ortran support

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

--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, AND-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-include=INCDir
--with-mpi-library=LIBDir
--with-mpi-flags=ARG
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 compiler 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 F ortran support

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

--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, AND-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-include=INCDir
--with-mpi-library=LIBDir
--with-mpi-flags=ARG
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 compiler 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.
2.3 Configuration examples

Note: The cvode, preissol, and pifea 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 unresolved external symbols, the libraries are still built as static libraries for the purpose of consistency.

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
FF77

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 FF77] must be locally (re)defined in order to use a different compiler. For example, to use xcc (executable name of chosen compiler) as the C language compiler, use CC=xCC in the configure step.

CFLAGS
FFLAGS

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

2.3 Configuration examples

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

```
% configure CC=gcc F77=g77 --with-cflags=-g3 --with-fflags=-g3
```

The above example builds sundials using gcc as the serial C compiler, g77 as the serial Fortran compiler, mpiicc as the parallel C compiler, mpiicc77 as the parallel Fortran compiler, and appends the -g3 compilation flag to the list of default flags.

```
% configure CC=gcc --disable-examples --with-mpicc=no

--with-mpi-root=/usr/apps/mpich/1.2.4
--with-mpi-libs=-lmpich
```

This example again builds sundials using gcc as the serial C compiler, but the --with-mpicc=no option explicitly disables the use of the corresponding MPI compiler script. In addition, since the --with-mpi-root option is given, the compilation flags -1/usr/apps/mpich/1.2.4/include and -1/usr/apps/mpich/1.2.4/lib are passed to gcc when compiling the MPI-enabled functions. The --disable-examples option disables the examples (which means a Fortran compiler is not required). The --with-mpi-libs option is required so that the configure script can check if gcc can link with the appropriate MPI library.

2.4 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 include. The default values for these directories are build/free/lib and build/free/include, respectively, but can be changed using the configure script options --prefix, --includedir and --libdir (see §2.2). For example, a global installation of sundials on a "NIX system could be accomplished using

```
% configure --prefix=/usr/local
```

Although all installed libraries reside under libdir, the public header files are further organized into subdirectories under modir. The installed libraries and exported header files are listed for reference in Table 2.1. The file extension .h 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 modir/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).

2.5 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 source tree to some other location and compiled separately.

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

- public header files located under source/free/solver/include
- implementation header files and source files located under source/free/solver/source
- (optional) Fortran/C interface files located under source/free/solver/ftcmix
- shared public header files located under source/free/shared/include
- shared source files located under source/free/shared/source
- (optional) nvector_serial header and source files located under source/free/nvec_ser
- (optional) nvector_parallel header and source files located under source/free/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.
- 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.
Table 2.1: SUNDIALS libraries and header files (names are relative to `libdir` for libraries and to `incdir` for header files)

<table>
<thead>
<tr>
<th>Section</th>
<th>Libraries</th>
<th>Header files</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SHARED</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Libraries</td>
<td>lib/sundials TYPES.h</td>
<td>lib/sundials TYPES.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials JMDIFF.h</td>
<td>lib/sundials JMDIFF.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials LSODA.h</td>
<td>lib/sundials LSODA.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials TNLBDES.h</td>
<td>lib/sundials TNLBDES.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials GSPGIS.h</td>
<td>lib/sundials GSPGIS.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials NVECTOR.h</td>
<td>lib/sundials NVECTOR.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials DENSE.h</td>
<td>lib/sundials DENSE.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials ITERATIVE.h</td>
<td>lib/sundials ITERATIVE.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials BAND.h</td>
<td>lib/sundials BAND.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials SPBCGS.h</td>
<td>lib/sundials SPBCGS.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials SPTFQMR.h</td>
<td>lib/sundials SPTFQMR.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials SPGMR.h</td>
<td>lib/sundials SPGMR.h</td>
</tr>
<tr>
<td><strong>VECTOR_SERIAL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Libraries</td>
<td>lib/sundials NVECTOR_SERIAL.h</td>
<td>lib/sundials NVECTOR_SERIAL.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials NVECTOR_SERIAL.a</td>
<td>lib/sundials NVECTOR_SERIAL.a</td>
</tr>
<tr>
<td><strong>VECTOR_PARALLEL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Libraries</td>
<td>lib/sundials NVECTOR_PARALLEL.h</td>
<td>lib/sundials NVECTOR_PARALLEL.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials NVECTOR_PARALLEL.a</td>
<td>lib/sundials NVECTOR_PARALLEL.a</td>
</tr>
<tr>
<td><strong>CVODE</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Libraries</td>
<td>lib/sundials CVODE.h</td>
<td>lib/sundials CVODE.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials CVODE.dense.h</td>
<td>lib/sundials CVODE.dense.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials CVODE.qlag.h</td>
<td>lib/sundials CVODE.qlag.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials CVODE.bandpre.h</td>
<td>lib/sundials CVODE.bandpre.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials CVODE.pppnr.h</td>
<td>lib/sundials CVODE.pppnr.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials CVODE.pppmr.h</td>
<td>lib/sundials CVODE.pppmr.h</td>
</tr>
<tr>
<td><strong>CVODES</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Libraries</td>
<td>lib/sundials CVODES.h</td>
<td>lib/sundials CVODES.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials CVODES.dense.h</td>
<td>lib/sundials CVODES.dense.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials CVODES.qlag.h</td>
<td>lib/sundials CVODES.qlag.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials CVODES.bandpre.h</td>
<td>lib/sundials CVODES.bandpre.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials CVODES.pppnr.h</td>
<td>lib/sundials CVODES.pppnr.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials CVODES.pppmr.h</td>
<td>lib/sundials CVODES.pppmr.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials CVODES.impl.h</td>
<td>lib/sundials CVODES.impl.h</td>
</tr>
<tr>
<td><strong>IDA</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Libraries</td>
<td>lib/sundials IDA.h</td>
<td>lib/sundials IDA.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials IDA.dense.h</td>
<td>lib/sundials IDA.dense.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials IDA.qlag.h</td>
<td>lib/sundials IDA.qlag.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials IDA.bandpre.h</td>
<td>lib/sundials IDA.bandpre.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials IDA.pppnr.h</td>
<td>lib/sundials IDA.pppnr.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials IDA.pppmr.h</td>
<td>lib/sundials IDA.pppmr.h</td>
</tr>
<tr>
<td><strong>KINSOL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Libraries</td>
<td>lib/sundials KINSOL.h</td>
<td>lib/sundials KINSOL.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials KINSOL.dense.h</td>
<td>lib/sundials KINSOL.dense.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials KINSOL.qlag.h</td>
<td>lib/sundials KINSOL.qlag.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials KINSOL.bandpre.h</td>
<td>lib/sundials KINSOL.bandpre.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials KINSOL.pppnr.h</td>
<td>lib/sundials KINSOL.pppnr.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials KINSOL.pppmr.h</td>
<td>lib/sundials KINSOL.pppmr.h</td>
</tr>
<tr>
<td></td>
<td>lib/sundials KINSOL.impl.h</td>
<td>lib/sundials KINSOL.impl.h</td>
</tr>
</tbody>
</table>
2.5 Building SUNDIALS without the configure script

/*
 * 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 "2.2.0"

/* 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 '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: 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 */
/* define SUNDIALS_UNDERSCORE_NONE 1 */
/* define SUNDIALS_UNDERSCORE_ONE 1 */
/* 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

\[ \dot{y} = f(t, y), \quad y(0) = y_0, \quad (3.1) \]

where \( y \in \mathbb{R}^N \). 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. \textit{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 \textit{cvode} are variable-order, variable-step multistep methods, based on formulas of the form

\[ \sum_{i=0}^{q} \alpha_i y^{n-i} + h \sum_{i=0}^{q} \beta_i y^{n-i} = 0 \quad (3.2) \]

Here the \( y^n \) are computed approximations to \( y(t_n) \), and \( h_n = t_n - t_{n-1} \) is the step size. The use of \textit{cvode} must choose appropriately one of two multistep methods. For nonstiff problems, \textit{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, \textit{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 \( a_n = -1 \). See [4] and [17].

For either choice of formula, the nonlinear system

\[ G(y^n) \equiv y^n - h \sum_{i=0}^{q} \alpha_i y^{n-i} + a_n = 0 \quad (3.3) \]

where \( a_n = \sum_{i=0}^{q} \alpha_i y^{n-i} + h \sum_{i=0}^{q} \beta_i y^{n-i} \), must be solved (approximately) at each integration step. For this, \textit{cvode} offers the choice of either functional iteration, suitable only for nonstiff systems, and various versions of Newton iteration. Functional iteration, given by

\[ y^{(i+1)} = h \sum_{i=0}^{q} \beta_i f(t_n, y^{(i)}) + a_n, \]

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

\[ M (y^{(i+1)} - y^{(i)}) = -G(y^{(i)}), \quad (3.4) \]

in which

\[ M \equiv I - \gamma J, \quad J = 0, \quad \text{and} \quad \gamma = h_n \alpha_n. \]

The initial guess for the iteration is a predicted value \( y^{(0)} \) computed explicitly from the available history data. For the Newton corrections, \textit{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,
- \textit{spgmr} = scaled preconditioned GMRES (Generalized Minimal Residual method) without restarts,
- \textit{spdfc} = scaled preconditioned Bi-CGSTab (Bi-Conjugate Gradient Stable method), or
- \textit{spffmr} = 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 (\textit{spgsm}, \textit{spdfc}, or \textit{spffmr}) 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 \( G \). 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, \textit{cvode} uses a weighted root-mean-square norm, denoted \( W = \sqrt{\frac{\sum w_i y_i^2}{\sum w_i}} \), for all error-like quantities. The multiplicative weights used are based on the current solution and on the relative and absolute tolerances input by the user, namely

\[ W_i = 1/(\text{rtol} \cdot |y_i| + \text{atol}_i). \quad (3.6) \]

Because \( W_i \) represents a tolerance in the component \( y_i \), a vector whose norm is 1 is regarded as "small." For brevity, we will usually drop the subscript \( W \) and use \( |\cdot| \) for 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 \( M y \) 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 \( \gamma \) of \( y^{(i)} \) at the last update satisfies \( |\gamma|/|y^{(i)} - y^{(i-1)}| > 0.3 \),
- 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.

On a convergence failure with an outdated matrix, and the value \( \gamma \) of \( y \) at the last update satisfies \( |\gamma|/|y^{(i)} - y^{(i-1)}| < 0.2 \), 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 \( y^{(n)} \) will have to satisfy a local error test \( ||y^{(n)} - y^{(n-1)}|| \leq \epsilon \). Letting \( y^* \) denote the exact solution of (3.3), we want to ensure that the iteration error \( y^* - y^{(n)} \) is small relative to \( \epsilon \), specifically that it is less than 0.1\( \epsilon \). (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 \( R = 1 \) and, after computing a correction \( \Delta y = y^{(n)} - y^{(n-1)} \), we update \( R \) if \( \epsilon \) as

\[
R = \min(0.3R, ||\Delta y||/||\Delta y_{\{1\}}||)
\]

Now we use the estimate

\[
||y^* - y^{(n)}|| \approx ||y^{(n+1)} - y^{(n)}|| \approx R||y^{(n)} - y^{(n-1)}||
\]

Therefore the convergence (stopping) test is

\[
R||\Delta y|| < 0.1\epsilon
\]

We allow at most 3 iterations (but this limit can be changed by the user). We also declare the iteration diverged if any \( ||\Delta y||/||\Delta y_{\{1\}}|| > 2 \) with \( m > 1 \). If convergence fails with \( J \) or \( P \) current, we are forced to reduce the step size, and we replace \( h_i \) by \( h_{i-1} \). 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 error must also be controlled, and this also involves the local error test constant. The linear iteration error in the solution vector \( y \) 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 \( (0.1) \epsilon \).

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 usual case, we use the usual approximation

\[
J \approx \left[ f_i(t, y + \sigma \sigma_i) - f_i(t, y) / \sigma \right]
\]

The increments \( \sigma_i \) are given by

\[
\sigma_i = \max \left\{ \sqrt{U[i]}, \sigma_i / W_i \right\}
\]

where \( U \) is the unit roundoff, \( \sigma_i \) is a dimensionless value, and \( W_i \) is the error weight defined in (3.16).

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 \( Ju \). If a routine for these products is not supplied, \( J \) is computed as

\[
Jv = \left[ f_i(t, y + \sigma v) - f_i(t, y) / \sigma \right] v
\]

The increment \( \sigma \) is \( 1/2U \), so that \( \sigma \) has norm 1.

A critical part of \texttt{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 fails. As with any linear multistep method, the local truncation error LTE, at order \( q \) and step size \( h \), satisfies an asymptotic relation

\[
LTE = C \frac{h^{q+1}}{q+1} + O(h^{q+2})
\]

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

\[
LTE \equiv C ||y^* - y^{(n)}|| + O(h^{q+2})
\]

The local error test is simply \( LTE \leq \epsilon \). Using the above, it is performed on the predictor-corrector difference \( \Delta y = y^{(n)} - y^{(n)} \) (with \( y^{(n)} \) the final iterate computed), and takes the form

\[
||\Delta y|| \leq \epsilon \equiv 1/C
\]

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

\[
(h'/h)^{q+1} ||\Delta y|| = \epsilon / h
\]

Here \( 1/h \) 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, \texttt{CVODE} returns to the user with a give-up message.

In addition to adjusting the step size to meet the local error test, \texttt{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 \( h' \) is done exactly as when the error test fails, giving a tentative step size ratio

\[
h'/h = (q/6)||\Delta y||^{1/(q+1)} = q_{k-1}
\]

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

\[
h'/h = (q/6)||\Delta y||^{1/(q+1)} = q_{k-1}
\]

and

\[
h'/h = (1/10)||\Delta y||^{1/(q+2)} = q_{k+1}
\]

The new order and step size are then set according to

\[
q = \max(q_{k-1}, q_{k+1} + 1)
\]

with \( q' \) set to the index achieving the above maximum. However, if we find that \( q < 5 \), 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 100.

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

Normally, \texttt{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 \texttt{CVODE} not to integrate past a given stopping point \( t = t_{\text{stop}} \).

### 3.2 BDF stability limit detection

\texttt{CVODE} includes an algorithm, \texttt{STALO} (STABILITY Limit Detection), which provides protection against potentially unstable behavior of the BDF multistep integration methods in certain situations, as described below.

When the BDF option is selected, \texttt{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 \( \lambda \) 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 stiffness stable. In each case, in order for the method to be stable at step size $h$ on the scalar model problem, the product $h A$ 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 those orders, if an eigenvalue of $A$ 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 $h A$ 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. 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 $g_i$ is computed, CYCDE 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 $t$, CYCDE computes $g_i(t + 4 h)$ for a small increment $h$, slightly further in the direction of integration, and if any $g_i(t + 4 h) = 0$ also, CYCDE stops and reports an error. This way, each time CYCDE 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, CYCDE has an interval $(t_{lo}, t_{hi})$ in which roots of the $g_i(t)$ are to be sought, such that $t_{lo}$ is further ahead in the direction of integration, and all $g_i(t_{hi}) \neq 0$. The endpoint $t_{lo}$ is either $t_{lo}$, the end of the time step last taken, or the next requested output time $t_{lo}$ if this comes sooner. The endpoint $t_{hi}$ is either $t_{hi}$, or the last output time $t_{hi}$ (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 $t_{lo}$ if an exact zero was found. The algorithm checks $g$ at $t_{lo}$ 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 $g_j(t_{lo}) = 0$) or we proceed to the next time interval (starting at $t_{lo}$). 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

$$\tau = 100 \times U \times |t| \times |g| \quad (U = \text{unit roundoff}) .$$

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 $|g_j(t_{lo})|/|g_j(t_{hi})|$, corresponding to the closest to $t_{lo}$ of the secular 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 $[t_{mid}, t_{hi})$, then that root is reported. The loop continues until $|t_{lo} - t_{hi}| < \tau$, and then the reported root location is $t_{lo}$.

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

$$t_{mid} = t_{lo} - \frac{(t_{hi} - t_{lo})g_j(t_{mid})/g_j(t_{lo}) - \alpha g_j(t_{lo})}{\alpha} ,$$

where $v$ is a weight parameter. On the first two passes through the loop, $\alpha$ is set to 1, making $t_{mid}$ the secant method value. Thereafter, $\alpha$ 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, $\alpha$ is set to 1. If the two sides were the same, $\alpha$ 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 $1/2$ and $5/2$ (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 those which also do sensitivity analysis calculations are available or in development. CVODE, 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 \( \frac{dy}{dt} = f(t, y) \);
- CVODES, a solver for stiff and nonstiff ODEs \( \frac{dy}{dt} = f(t, y, p) \) 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, cvodeimpl.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;
- CVSPFS: a parallel solver for linear systems.
4.2 CVODE organization

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.

- cvspfqmr: 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 cvspfqmr, 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, cvrand, cvspgmr, cvspbcg, and cvspfqmr is a set of interface routines built on top of a generic solver module, named dense, band, spgmr, spbcg, and spfqmr, 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
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 [55] may also be helpful. These 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 cvbandpre 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.a
- libdir/libsundials_nvector.a (one or two files),

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

- include/include
- include/stdio
- include/sundials

The directories libdir and incdir are the install library and include directories. For a default installation, these are build/Release/lib and build/Release/include, respectively, where build 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 §2.2).

Additionally, based on the current precision, sundials_types.h defines REAL to be the largest value representable as a realtype, SMALL_REAL to be the smallest value representable as a realtype, and UNIT_ROUNDSFF 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 RCST. 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,

```c
#define A 1.0F
#define B 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 RCST(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 RCST macro internally to declare all of its floating-point constants.

A user program which uses the type realtype and the RCST 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 RCST.) 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, as long as the sundials libraries use the correct precision (for details see §2.2).

5.3 Header files

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

- 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 boolenotype 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 nvector 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;
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 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(args, &argc); to initialize MPI if used by the user’s program, apart from the internal use in nvector_parallel. Here args and argc 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] NVECTOR_S(y0)
   [P] NVECTOR_P(y0)

4. Create CVODE object
   Call cvode_mem = CVodeCreate(lmem, 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 = CVBand(...);
   [P] ier = CVBand(...);
   ier = CVDiag(...);
   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 §5.5.7 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 CVGet* 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);
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 these 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.

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

**Call**

```
cvode_mem = 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` (C function that 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_YW`, or `CV_NF`. Here `itol = SS` indicates scalar relative error tolerance and scalar absolute error tolerance, while `itol = CV_YW` 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_NF`, 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. (1.6). See `CVodeSetErrFn` in §5.5.5.

**Return value**
If an error occurred, `CVodeMalloc` also sends an error message to the error handler function.

**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[i]` (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 `cvode` in the cvode package, and the discussion of it in the cvode Examples document [15]. In that problem, the three components vary between 0 and 1, and have different noise levels, hence the `abstol` vector. It is impossible to give any general advice on `abstol` values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.

3. Finally, it is important to pick all the tolerance values conservatively, because they control the error committed on each individual time-step. The final (global) errors are some sort of accumulation of those per-step errors. A good rule of thumb is to reduce the tolerances by a factor of .01 from the actual desired limits on errors. So if you want .01% accuracy (globally), a good choice is `reltol = 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 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 (1.4). There are six cvode linear solvers currently available for this task: cvdense, cvband, cvdiag, cvspils, cvspbcg, and cvspgmr. The first three are direct solvers and their names indicate the type of approximation used for the Jacobian J = Df/dy: cvdense, cvband, and cvdiag work with dense, band, and diagonal approximations to J, respectively. The last three cvode linear solvers — cvspils, cvspbcg, and cvspgmr — are Krylov iterative solvers, which use scaled preconditioned GMRES, scaled preconditioned Bi-CGSTab, and scaled preconditioned TFQMR, respectively. Together, they are referred to as cvspil (from scaled preconditioned iterative linear solvers).

With any of the Krylov methods, preconditioning can be done on the left only, or on both the left and the right, or not at all. 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. 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_l and P_r (either of which could be the identity matrix), such that the product P_l P_r approximates the Newton matrix $N = I - \gamma J$ (of §5.5).

To specify a cvode linear solver, after the call to CCodeCreate but before any calls to CCode, the user’s program must call one of the functions cvdense, cvband, cvdiag, cvspils, cvspbcg, or cvspgmr, as documented below. The first argument passed to these functions is the cvode memory pointer returned by CCodeCreate. 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 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, spil, spbcg, spgmr, and spftpqr, are described separately in Chapter 9.

Using CVODE for C Applications

5.5 User-callable functions
5.5 User-callable functions

CVDIAG_MEM_NULL The cvode_mem pointer is NULL.
CVDIAG_JLL_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.

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 Spbcg.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_JLL_INPUT The preconditioner type pretype is not valid.
- CVSPILS_MEM_FAIL A memory allocation request failed.

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

CVSpfgmr

Call flag = CVSpfgmr(cvode_mem, pretype, maxl);

Description The function CVSpfgmr selects the CVSPFGMR linear solver. The user’s main function must include the cvode Fgmr.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 cvspfgmr initialization was successful.
- CVSPILS_MEM_NULL The cvode_mem pointer is NULL.
- CVSPILS_JLL_INPUT The preconditioner type pretype is not valid.
- CVSPILS_MEM_FAIL A memory allocation request failed.

Notes The CVSPFGMR 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,STE D, and CV_ONE_STEP modes are similar to CV_NORMAL and CV_ONE_STEP, respectively, except that the integration never proceeds past the value tout (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 error, *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_TSTOPRETURN CVode succeeded by reaching the stopping point specified through the optional input function CVodeSetStopTime (see §5.5.5). If *itask > 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.
5.5 User-callable functions

CVODEMalloc
The CVODE memory was not allocated by a call to CVODEMalloc.

CVODEMalloc
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 CVODEMalloc flag will also be returned if the linear solver function initialization (called by the user after calling CVODEMalloc) failed to set the linear solver-specific "solve" field in cvode_mem. Finally, if the initial time \( t_0 \) and the final time \( t_{\text{final}} \) are too close to each other and the user did not specify an initial step size, CVODE will also return CVODEMalloc. In any case, the user should see the error message for details.

CVODEVWORK
The solver took maxstep internal steps but could not reach tout. The default value for maxstep is MAXSTEP_DEFAULT = 500.

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

CVODEVFAILURE
Error test failures occurred too many times (\( MKDF = 7 \)) during one internal time step or occurred with \( |h|=h_{\text{min}} \).

CVODEVFAIL
Convergence test failures occurred too many times (\( MKDF = 10 \)) during one internal time step or occurred with \( |h|=h_{\text{min}} \).

CVODEVINITFAIL
The linear solver's initialization function failed.

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

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

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

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

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

CVODEVXUNRECSHIFTFUNCERR
The right-hand side 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.

CVODEXROOTFUNCFAIL
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 CVODESTEP 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 yret correspond to the farthest point reached in the integration. On all other error returns, tret and yret 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, beginning 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 §5.6.3).

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
- CV_ERR_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
- CV_ERR_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.
### 5.5 User-callable functions

#### Table 5.1: Optional inputs for CVODE, CVODES, CVBAND, and CVSPILS

<table>
<thead>
<tr>
<th>Optional input</th>
<th>Function name</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error handler function</td>
<td>CVodeSetErrHandlerFn</td>
<td>internal fn.</td>
</tr>
<tr>
<td>Pointer to an error file</td>
<td>CVodeSetErrFile</td>
<td>stderr</td>
</tr>
<tr>
<td>Data for right-hand side function</td>
<td>CVodeSetFdata</td>
<td>NULL</td>
</tr>
<tr>
<td>Maximum order for BDF method</td>
<td>CVodeSetMaxOrd</td>
<td>5</td>
</tr>
<tr>
<td>Maximum order for Adams method</td>
<td>CVodeSetMaxOrd</td>
<td>12</td>
</tr>
<tr>
<td>Maximum no. of internal steps before ( t_{\text{int}} )</td>
<td>CVodeSetMaxNumSteps</td>
<td>500</td>
</tr>
<tr>
<td>Maximum no. of warnings for ( t_n + h = t_{\text{int}} )</td>
<td>CVodeSetMaxHnilWarns</td>
<td>10</td>
</tr>
<tr>
<td>Flag to activate stability limit detection</td>
<td>CVodeSetStabLimDet</td>
<td>FALSE</td>
</tr>
<tr>
<td>Initial step size</td>
<td>CVodeSetInitStep</td>
<td>estimated</td>
</tr>
<tr>
<td>Minimum absolute step size</td>
<td>CVodeSetMinStep</td>
<td>0.0</td>
</tr>
<tr>
<td>Maximum absolute step size</td>
<td>CVodeSetMaxStep</td>
<td>∞</td>
</tr>
<tr>
<td>Value of ( t_{\text{max}} )</td>
<td>CVodeSetMaxStepTime</td>
<td>undefined</td>
</tr>
<tr>
<td>Maximum no. of error test failures</td>
<td>CVodeSetMaxErrTestFails</td>
<td>7</td>
</tr>
<tr>
<td>Maximum no. of nonlinear iterations</td>
<td>CVodeSetMaxNonlinIters</td>
<td>3</td>
</tr>
<tr>
<td>Maximum no. of convergence failures</td>
<td>CVodeSetMaxConvFails</td>
<td>10</td>
</tr>
<tr>
<td>Coefficient in the nonlinear convergence test</td>
<td>CVodeSetNonlinConvCoef</td>
<td>0.1</td>
</tr>
<tr>
<td>Nonlinear iteration type</td>
<td>CVodeSetIterType</td>
<td>none</td>
</tr>
<tr>
<td>Integration tolerances</td>
<td>CVodeSetTolerances</td>
<td>none</td>
</tr>
<tr>
<td>Error computation function</td>
<td>CVodeSetErrFn</td>
<td>internal fn.</td>
</tr>
</tbody>
</table>

#### CVODES 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 |
| Preconditioning type | CVspilsSetPrecType | none |
| Ratio between linear and nonlinear tolerances | CVspilsSetRlNl | 0.05 |
| Type of Gram-Schmidt orthogonalization\(^{(*)}\) | CVspilsSetGSType | classical GS |
| Maximum Krylov subspace size\(^{(b)}\) | CVspilsSetMaxl | 5 |

\(^{(*)}\) Only for CVSPGMR

\(^{(b)}\) Only for CVSPILS and CVSPTRILS

---

**CVodeSetMaxOrd**

**Call**

\[
\text{flag} = \text{CVodeSetMaxOrd}(\text{cvode}, \text{maxord});
\]

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

**Arguments**
- `cvode` (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_MEMNULL` The `cvode` pointer is NULL.

**Notes**
If `f_data` is not specified, a NULL pointer is passed to the `f` function.

---

**CVodeSetMaxNumSteps**

**Call**

\[
\text{flag} = \text{CVodeSetMaxNumSteps}(\text{cvode}, \text{mxsteps});
\]

**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` (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_MEMNULL` The `cvode` pointer is NULL.
- `CV_MEMBADPARAM` The specified value `mxsteps` is negative, or larger than its previous value.

**Notes**
- The default value is `ADAMS_D_MAX = 12` for the Adams-Moulton method and `BDF_D_MAX = 5` for the BDF method.
- Since `mxsteps` affects the memory requirements for the internal cvode memory block, its value can not be increased past its previous value.

---

**CVodeSetMaxHnilWarns**

**Call**

\[
\text{flag} = \text{CVodeSetMaxHnilWarns}(\text{cvode}, \text{mxhnil});
\]

**Description**
The function CVodeSetMaxHnilWarns specifies the maximum number of warning messages issued by the solver that \( t + h = t_{\text{next step}} \).

**Arguments**
- `cvode` (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.
5.5 User-callable functions

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 xminl 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 § 5.2.

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_JILL_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, CVyldet, 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 $\dot{y} / [0.5y^2]_{\text{WAM}} = 1$, where $\dot{y}$ 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_JILL_INPUT Either hmin is not positive or it is larger than the maximum allowable step.

Notes
The default value is 0.0.
5.5 User-callable functions

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.

CVodeSetNonlinConvCost
Call flag = CVodeSetNonlinConvCost(cvode_mem, alscalef);
Description The function CVodeSetNonlinConvCost specifies the safety factor in the nonlinear convergence test (see §3.3).
Arguments cvode_mem (void *) pointer to the CVODE memory block.
alscalef (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_REALTOL 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_ILLEGAL_INPUT The iter value passed is neither CV_REALTOL 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 IVector 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_ILLEGAL_INPUT An input argument has an illegal value.
Notes The integration tolerances are initially specified in the call to CVodeCreate (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.

CVodeSetExtFn
Call flag = CVodeSetExtFn(cvode_mem, efun, e_data);
Description The function CVodeSetExtFn 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 (CVExtFn) is the C function which defines the ext 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 e_data.
It is illegal to call CVodeSetExtFn before a call to CVodeMalloc.

5.5.5.2 Dense linear solver

The CV_DENSE solver needs a function to compute a dense approximation to the Jacobian matrix J(t,y). This function must be of type CV_DenseJacFn. The user can supply his/her own dense Jacobian function, or use the default difference quotient function CV_DenseDQJac that comes with the CV_DENSE solver. To specify a user-supplied Jacobian function djac and associated user data jac_data, CV_DENSE provides the function CV_DenseSetJacFn. The CV_DENSE 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.

CV_DenseSetJacFn
Call flag = CV_DenseSetJacFn(cvode_mem, djac, jac_data);
Description The function CV_DenseSetJacFn 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 (CV_DenseJacFn) user-defined dense Jacobian approximation function.
jac_data (void *) pointer to the user-defined data structure.
5.5 User-callable functions

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

- CVENSE_SUCCESS: The optional value has been successfully set.
- CVENSE_MEM_NULL: The cvode_mem pointer is NULL.
- CVENSE_LSNULL: The cvilens linear solver has not been initialized.

Notes: By default, cvilens 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 cvilens 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 cvilens 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.

\( \text{CVBandSetJacFn} \)

Call: \[ \text{flag} = \text{CVBandSetJacFn}(\text{cvode_mem}, \text{bjac}, \text{jac_data}); \]

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_LSNULL: The cvilens 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 cvilens linear solvers, then the user must supply a preconditioner solve function psetup and specify its name in a call to CVilensSetPreconditioner.

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 CVilensSetPreconditioner. Optionally, a cvilens solver passes the pointer p_data received through CVilensSetPreconditioner to the preconditioner psetup and polve functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program. The pointer p_data may be identical to f_data, if the latter was specified through CVodeSetFdata.

The cvilens 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 CVilensDQJac that comes with the cvilens solvers. A user-defined Jacobian-vector function must be of type CVilensJacTimesVecFn and can be specified through a call to CVilensSetJacTimesVecFn (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 CVilensSetJacTimesVecFn, a pointer to a user-defined data structure, jac_data, which the cvilens 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.

\( \text{CVilensSetPreconditioner} \)

Call: \[ \text{flag} = \text{CVilensSetPreconditioner}(\text{cvode_mem}, \text{psetup}, \text{polve}, \text{p_data}); \]

Description: The function CVilensSetPreconditioner specifies the preconditioner setup and solve functions and the pointer to user data.

Arguments:
- cvode_mem (void *) pointer to the CVODE memory block.
- psetup (CVilensPrecSetupFn) user-defined preconditioner setup function.
- polve (CVilensPrecSolveFn) 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

- CVILENS_SUCCESS: The optional value has been successfully set.
- CVILENS_MEM_NULL: The cvode_mem pointer is NULL.
- CVILENS_LSNULL: The cvilens linear solver has not been initialized.

Notes: The function type CVilensPrecSolveFn is described in §5.6.7. The function type CVilensPrecSetupFn is described in §5.6.8.

\( \text{CVilensSetJacTimesVecFn} \)

Call: \[ \text{flag} = \text{CVilensSetJacTimesVecFn}(\text{cvode_mem}, \text{bjac}, \text{jac_data}); \]

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

Arguments:
- cvode_mem (void *) pointer to the CVODE memory block.
- bjac (CVilensJacTimesVecFn) user-defined Jacobian-vector function.
- jac_data (void *) pointer to the user-defined data structure.

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

- CVILENS_SUCCESS: The optional value has been successfully set.
- CVILENS_MEM_NULL: The cvode_mem pointer is NULL.
- CVILENS_LSNULL: The cvilens linear solver has not been initialized.

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

The function type CVilensJacTimesVecFn is described in §5.6.6.

\( \text{CVilensSetPrecType} \)

Call: \[ \text{flag} = \text{CVilensSetPrecType}(\text{cvode_mem}, \text{pretype}); \]

Description: The function CVilensSetPrecType 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

- CVILENS_SUCCESS: The optional value has been successfully set.
5.5 User-callable functions

CVSPILS\_MEM\_NULL The cvode\_mem pointer is NULL.
CVSPILS\_JLMEM\_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 CVSPILS 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 cvspils linear solver has not been initialized.
CVSPILS\_ILL\_INPUT The Gram-Schmidt orthogonalization type gstype is not valid.

Notes
This option is available only for the CVSPILS 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) the value of the independent variable at which the derivative is requested.

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 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 BiCGStab or TFQMR methods.

Arguments
cv\_mem (void *) pointer to the cvode\_mem memory block.
maxl (int) maximum dimension of the Krylov subspace.

Return value
The return value flag (of type int) is one of
CVSPILS\_SUCCESS The optional value has been successfully set.
CVSPILS\_MEM\_NULL The cv\_mem pointer is NULL.
CVSPILS\_JLMEM\_NULL The cvspils 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 CVSPILS and CVSPTRAM 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., \( d^ky/dt^k \)(t), where \( t_k - h_{\text{ns}} \leq t \leq t_{k+1} \). \( t_k \) denotes the current internal time reached, and \( h_{\text{ns}} \) is the last internal step size successfully used by the solver. The user may request \( k = 0, 1, \ldots, q_y \), where \( q_y \) 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 \( 0, 1, \ldots, q_y \).
CV\_BAD\_T \( t \) is not in the interval \( [t_k - h_{\text{ns}}, t_{k+1}] \).
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 \( t_k \), \( q_y \), and \( h_{\text{ns}} \), 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, beginning 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 L8 (for Linear Solver) has been added here (e.g., lamrelL8).
### 5.5 User-callable functions

**Table 5.2: Optional outputs from cvode, cvdense, cvband, cvdiag, and cvspils**

<table>
<thead>
<tr>
<th>Optional output</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of cvode real and integer workspaces</td>
<td>CVodeGetWorkSpace</td>
</tr>
<tr>
<td>No. of steps</td>
<td>CVodeGetNumSteps</td>
</tr>
<tr>
<td>No. of calls to rhs function</td>
<td>CVodeGetNumRhsEvals</td>
</tr>
<tr>
<td>No. of calls to linear solver setup function</td>
<td>CVodeGetNumLinSolSetup</td>
</tr>
<tr>
<td>No. of local error test failures that have occurred</td>
<td>CVodeGetNumErrTestFails</td>
</tr>
<tr>
<td>Order used during the last step</td>
<td>CVodeGetLastOrder</td>
</tr>
<tr>
<td>Order to be attempted on the next step</td>
<td>CVodeGetCurrentOrder</td>
</tr>
<tr>
<td>Order reductions due to stability limit detection</td>
<td>CVodeGetStabLimitOrderReds</td>
</tr>
<tr>
<td>Actual initial step size used</td>
<td>CVodeGetInitStep</td>
</tr>
<tr>
<td>Step size used for the last step</td>
<td>CVodeGetLastStep</td>
</tr>
<tr>
<td>Step size to be attempted on the next step</td>
<td>CVodeGetCurrentStep</td>
</tr>
<tr>
<td>Current internal time reached by the solver</td>
<td>CVodeGetCurrentTime</td>
</tr>
<tr>
<td>Suggested factor for tolerance scaling</td>
<td>CVodeGetTolScaleFactor</td>
</tr>
<tr>
<td>Error weight vector for state variables</td>
<td>CVodeGetErrWeights</td>
</tr>
<tr>
<td>Estimated local error vector</td>
<td>CVodeGetLocalErrors</td>
</tr>
<tr>
<td>No. of nonlinear convergence failures</td>
<td>CVodeGetNumNonlinSolFails</td>
</tr>
<tr>
<td>All cvode nonlinear solver statistics</td>
<td>CVodeGetNonlinStats</td>
</tr>
<tr>
<td>cvspils nonlinear solver statistics</td>
<td>CVspilsGetNumNonlinSolvFails</td>
</tr>
<tr>
<td>Array showing roots found</td>
<td>CVspilsGetRootInfo</td>
</tr>
<tr>
<td>No. of calls to user root function</td>
<td>CVspilsGetNumEvals</td>
</tr>
<tr>
<td>Name of constant associated with a return flag</td>
<td>CVspilsGetReturnFlagName</td>
</tr>
</tbody>
</table>

**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**
  ```c
  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 words in the cvode workspace.
  - leniw (long int) the number of integer words 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:
  - $\text{base value: } lenrw = 89 + (\text{maxord}+5) \times N + 3 \times nrtfn$.
  - $\text{if } \text{itol} = \text{CV_CF, lenrw} = \text{lenrw} + N$.

  Where $N$ is the number of real words in one SVector ($N = N$).

  The size of the integer workspace (without distinction between int and long_int words) is given by:
  - $\text{base value: } leniw = 40 + (\text{maxord}+5) \times N + nrtfn$.
  - $\text{if } \text{itol} = \text{CV_J, leniw} = \text{leniw} + N$.

  Where $N$ is the number of integer words in one SVector ($N = 1$ for SVector_serial and $N = \text{npes}$ for SVector_parallel and mpas processor).

  For the default value of maxord, with no rootfinding, and with itol $\neq \text{CV_J, these lengths are given roughly by:}$
  - For the Adams method: leniw = 89 + 11N and leniw = 57
  - For the BDF method: leniw = 89 + 10N and leniw = 50

**CVodeGetNumSteps**

- **Call**
  ```c
  flag = CVodeGetNumSteps(cvode_mem, &numsteps);
  ```
- **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.
  - numsteps (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 made 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 made 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 to be used on the last internal step.
Arguments cvode_mem (void *) pointer to the cvode_mem 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.

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_mem 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.

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_mem 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.

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_mem 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 \((h_{\text{min}} \leq h \leq h_{\text{max}})\), or to meet the local error test.
5.5 User-callable functions

**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 §5.2).

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 through a call to CVodeSetStabLimDet, then nslred=0.

**CVodeGetTolScaleFactor**

Call flag = CVodeGetTolScaleFactor(cvode_mem, &tolsfac);

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

Arguments cvode_mem (void *) pointer to the CVODE memory block.
tolsfac (realtype) suggested scaling factor for user 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 (3.16).

Arguments cvode_mem (void *) pointer to the CVODE memory block.
eweight (B_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.
5.5 User-callable functions

Call `flag = CVDenseGetNonlinSolvConvFails(cvode, &nniters, &nncfails);`
Description The function `CVDenseGetNonlinSolvConvFails` returns the number of nonlinear convergence failures that have occurred.
Arguments `cvode` (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` pointer is NULL.

Call `flag = CVDenseGetNonlinSolvStats(cvode, &nniters, &nncfails);`
Description The function `CVDenseGetNonlinSolvStats` returns the CVODE nonlinear solver statistics as a group.
Arguments `cvode` (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` pointer is NULL.

Call `name = CVDenseGetReturnFlagName(flag);`
Description The function `CVDenseGetReturnFlagName` 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 CVODE function.

Call `flag = CVDenseGetWorkSpace(cvode, &lenrwLS, &leniwLS);`
Description The function `CVDenseGetWorkSpace` returns the CVDENSE real and integer workspace sizes.
Arguments `cvode` (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` pointer is NULL.
- CVDENSE_MEM_NULL The CVDENSE linear solver has not been initialized.
### 5.5 User-callable functions

**CVBANDGetNumRhsEvals**

**Call**

```c
flag = CVBANDGetNumRhsEvals(cvode, &nfevalsLS);
```

**Description**
The function `CVBANDGetNumRhsEvals` returns the number of realtype `nfevalsLS` calls to the right-hand side routine for finite-difference Jacobian approximation, and last return value from a `cvband` function.

**Arguments**
- `cvode` (of type `void *`) pointer to the `cvband` memory block.
- `nfevalsLS` (long int) the number of calls to the right-hand side routine for finite-difference Jacobian approximation, and last return value 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` pointer is NULL.
- `CVBAND_LMEM_NULL` The `cvband` linear solver has not been initialized.

**Notes**
In terms of the problem size `N`, the actual size of the real workspace is `3N` realtype words, and the actual size of the integer workspace is `N` integer words.

**CVBandGetWorkSpace**

**Call**

```c
flag = CVBandGetWorkSpace(cvode, &leniwLS, &lenrwLS);
```

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

**Arguments**
- `cvode` (of type `void *`) pointer to the `cvode` memory block.
- `leniwLS` (long int) the number of integer values in the `cvband` workspace.
- `lenrwLS` (long int) the number of realtype 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` pointer is NULL.
- `CVBAND_LMEM_NULL` The `cvband` linear solver has not been initialized.

**Notes**
In the terms of the problem size `N` and Jacobian half-bandwidths, the actual size of the real workspace is `(2 + upper + lower +2)N` realtype words, and the actual size of the integer workspace is `N` integer words.

**CVBandGetNumJacEvals**

**Call**

```c
flag = CVBandGetNumJacEvals(cvode, &njevals);
```

**Description**
The function `CVBandGetNumJacEvals` returns the 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.

**Arguments**
- `cvode` (of type `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` pointer is NULL.
- `CVBAND_LMEM_NULL` The `cvband` linear solver has not been initialized.

**Notes**
If the `cvband` setup function failed (CVode returned `CV.Server.FATAL`), the value `nfevalsLS` is incremented only if the default `CVBandDQJac` difference quotient function is used.

**CVBandGetLastFlag**

**Call**

```c
flag = CVBandGetLastFlag(cvode, &lsflag);
```

**Description**
The function `CVBandGetLastFlag` returns the last return value from a `cvband` routine.

**Arguments**
- `cvode` (of type `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` 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.

**CVDenseGetReturnFlagName**

**Call**

```c
name = CVDenseGetReturnFlagName(flag);
```

**Description**
The function `CVDenseGetReturnFlagName` returns the name of the corresponding constant.

**Arguments**
- `flag` (of type int) the number of integer values in the `cvdense` workspace.

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

**CVDiagGetWorkSpace**

**Call**

```c
flag = CVDiagGetWorkSpace(cvode, &leniwLS, &lenrwLS);
```

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

**Arguments**
- `cvode` (of type `void *`) pointer to the `cvode` memory block.
- `leniwLS` (long int) the number of integer values in the `cvdiag` workspace.
- `lenrwLS` (long int) the number of realtype 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` 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.
5.5 User-callable functions

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

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 right-hand side routine for finite-difference Jacobian approximation, and last return value from a linear solver function.
5.5 User-callable functions

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_MEM_NULL The cvode_mem 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_MEM_NULL The cvode_mem linear solver has not been initialized.

CVSpilsGetNumRhsEvals

Call
flag = CVSpilsGetNumRhsEvals(cvode_mem, &njvevals);

Description
The function CVSpilsGetNumRhsEvals returns the cumulative number of calls 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_MEM_NULL The cvode_mem linear solver has not been initialized.

Notes
If there are changes to the linear solver specifications, make the appropriate
nfevalsLS and the default value for
the linear solver has not been initialized.

CVodeMalloc

The function CVodeMalloc 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.
- nfvalsls (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_MEM_NULL The cvode_mem linear solver has not been initialized.

Notes
The value nfvalsls is incremented only if the default CVSpilsDJ/ timings difference
quadratic function is used.

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 longer 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 lim is unchanged (or changed from CV_ADAMS to CV_JDF)
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.
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 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:

```c
typedef int (*CVRhsFn)(realtype t, N_Vector y, N_Vector ydot, void *f_data);
```

Definition

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

Purpose

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 CVodeReInit.

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 CVodeStatus = CV_BaddirFuncFail is returned).

Notes

- Allocation of memory for \( ydot \) is handled within CVODE.
- 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. This include the situation when this occurs at the very first call to the CVRhsFn (in which case CVode returns CV_FIRST_BADDRFUNC_FAIL) or if 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_UNSREC_BADDRFUNC_FAIL).

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 \( errf \) (see CVodeSetErrFile), the user may provide a function of type CVErErrHandlerFn to process any such messages. The function type CVErErrHandlerFn is defined as follows:

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

Definition

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

Purpose

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 CVodeErrHandlerFn.

Return value

A CVErErrHandlerFn function has no return value.

Notes

- If the error code is negative for errors and positive (CVWARNING) for warnings. If a function returning a pointer to memory (e.g., CVErrFree and alloc) 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 \( w \) containing the weights in the WRMS norm \( \|w\|_W = \sqrt{\sum_{i=1}^N (W_i v_i)^2} \). The function type CVEwtFn is defined as follows:
5.6 User-supplied functions

**CVExtFn**

**Definition**
typedef int (*CVExtFn)(N_Vector y, N_Vector out, 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.
out 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 CvodeSetExtFn.

**Return value**
A CVExtFn function type must return 0 if it successfully set the error weights and −1 otherwise. In case of failure, a message is printed and the integration stops.

**Notes**
Allocation of memory for out is handled within cvode.

The error weight vector must have all components positive. It is the user’s responsibility 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);

**Purpose**
This function computes the dense Jacobian J = ∂f/∂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 output vector f(t,y).
jac_data is a pointer to user data — the same as the jac_data parameter passed to CVDenseSetJacFn.

**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_RECUR), or a negative value if it failed unrecoverably (in which case the integration is halted). Cvode returns CVLSETUP_FAIL and CVdense sets last_flag on CVdense_JACFUNC_RECUR).

**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) returns a pointer to the storage for the (i,j)-th element of the dense matrix J (i, j = 0 . . . 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 Jmn can be loaded with the statement DENSE_ELEM(J, m−1, n−1) = Jmn. Alternatively, DENSE_COL(J, i) returns a pointer to the storage for the i-th column of J (i = 0 . . . N − 1), and the elements of the i-th column are then accessed via ordinary array indexing. Thus Jmn can be loaded with the statements col.p = DENSE_COL(J, m−1); col.p[n−1] = Jmn. 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 §5.1.

If the user’s CVDenseJacFn function use 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, BandMat Jtmp1, BandMat Jtmp2, N_Vector tmp3);

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

**Arguments**
N is the problem size.
mupper and mlower are the upper and lower 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 output vector f(t,y).
jac_data is a pointer to user data — the same as the jac_data parameter passed to CVBandSetJacFn.

**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_RECUR), or a negative value if it failed unrecoverably (in which case the integration is halted). Cvode returns CVLSSETUP_FAIL and CVBand sets last_flag on CVBAND_JACFUNC_RECUR).

**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 accessors macros `CVSpilsBandJacFn` and `CVSpilsBandJacELEM` allow the user to read and write band matrix elements without making specific references to the underlying representation of the BandMat type. `CVSpilsBandJacFn(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 `CVSpilsBandJacFn(J, n-1, n+1) = J_{m,n}`. The elements within the band are those with `-mupper ≤ m-n ≤ mlower`. Alternatively, `CVSpilsBandJacFn(J, i, j)` returns a pointer to the diagonal element of the `j`th column of `J`, and if we assign this address to `realtype *col_j`, then the `i`th element of the `j`th column is given by `CVSpilsBandJacELEM(col_j, i, j)`, counting from 0. Thus for `(m, n)` within the band, `J_{m,n}` can be loaded by setting `col_p = CVSpilsBandJacFn(J, n-1); CVSpilsBandJacELEM(col_p, n-1, n+1) = J_{m,n}`. The elements of the `j`th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type BandMat. The array `col_p` can be indexed from `mupper` to `mlower`. For large problems, it is more efficient to use the combination of `CVSpilsBandJacFn` and `CVSpilsBandJacELEM` than to use the `CVSpilsBandJac`. As in the dense case, these macros all number rows and columns starting from 0, not 1.

The BandMat type and the accessor macros `CVSpilsBandJacFn`, `CVSpilsBandJacELEM` and `CVSpilsBandJacELEM` are documented in §9.2.

If the user’s `CVSpilsJacFn` 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 `sugen`, `sougen`, or `systgen` is selected (`CVSpilsJac` is called in step 7 of §5.4), the user may provide a function of type `CVSpilsJacTimesVecFn` in the following form:

```c
typedef int (*CVSpilsJacTimesVecFn)(realtype t, N_Vector x, N_Vector y, N_Vector fy, void *jac_data, N_Vector tmp);
```

**Definition**

This function computes the product `Jv = (∂f/∂y)v` (or an approximation to it).

- **Arguments**
  - `t` is the current value of the independent variable.
  - `x` is the current value of the independent variable vector.
  - `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 `CVSpilsJac`.
  - `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 Jacobian information (matrix-vector product).

### 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:

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

**Definition**

This function solves the preconditioning system `Pz = r`.

- **Arguments**
  - `t` is the current value of the independent variable.
  - `x` is the current value of the independent variable vector.
  - `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 exponent appearing in the Newton matrix `M = I - γ J).
  - `delta` is the input tolerance to be used if an iterative method is employed in the solution.

To obtain the `N_Vector` etc. call `CVodeGetErrorWeights` (see §5.5.7.1). `ir` is an input flag indicating whether the preconditioner solve function is to use the left preconditioner (`ir = 1`) or the right preconditioner (`ir = 2`). `p_data` is a pointer to user data — the same as the `p_data` parameter passed to the function `CVSpilsSetPreconditioner`. `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:

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

**Definition**

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

- **Arguments**
  - `t` is the current value of the independent variable.
  - `x` is the current value of the independent variable vector.
  - `y` is the current value of the dependent variable vector.
  - `fy` is the current value of the vector `f(t,y)`.
  - `jok` is a boolean flag indicating whether the Jacobian of `f` exists.
  - `jcpTr` is a boolean flag indicating whether the Jacobian transpose of `f` exists.
  - `gamma` is the scalar exponent appearing in the Newton matrix `M = I - γ J`.
  - `p_data` is a pointer to user data — the same as the `p_data` parameter passed to the function `CVSpilsSetPreconditioner`.
  - `tmp1`, `tmp2`, and `tmp3` are `N_Vector` objects that may be used for work space.

**Return value**

The value to be returned by this 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).
5.7 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

$\begin{align*}
\mathbf{t} & \quad \text{is the current value of the independent variable.} \\
\mathbf{y} & \quad \text{is the current value of the dependent variable vector, namely the predicted value of } g(t). \\
\mathbf{fy} & \quad \text{is the current value of the vector } f(t,y). \\
j\text{ok} & \quad \text{is an input flag indicating whether Jacobian-related data needs to be recomputed. The } j\text{ok} \text{ argument provides for the re-use of Jacobian data in the preconditioner solve function. } j\text{ok} = \text{FALSE} \text{ means that Jacobian-related data must be recomputed from scratch. } j\text{ok} = \text{TRUE} \text{ means that Jacobian data, if saved from the previous call to this function, can be reused (with the current value of } g\text{amma). A call with } j\text{ok} = \text{TRUE} \text{ can only occur after a call with } j\text{ok} = \text{FALSE}.}
\end{align*}$

jcarPtr 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 $\gamma$ appearing in the Newton matrix $M = I - \gamma P$.

p_data is a pointer to user data, the same as that passed to CVSpilsSetPreconditioner.

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 $M = I - \gamma P$.

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_RNDSOFPF 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.

5.7.2 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:

```c
int CVRootFn( {
  cvode_mem, rootsfound);
}
```

Description: The function CVodeGetRootInfo returns an array showing which functions were found to have a root. For $i = 0, \ldots, nrtfn - 1$, rootsfound[$i$] = 1 if $g_i$ has a root, and $= 0$ if not.

Arguments: $cvode_mem$ (void *) pointer to the cvode memory block returned by CVodeCreate.

rootsfound (int *) array of length nrtfn with the indices of the user functions $g_i$ found to have a root.

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

- CV_SUCCESS: The call to CVodeGetRootInfo was successful.
- CV_MEM_NULL: The cvode_mem argument was NULL.
- CV_MEM_FAIL: A memory allocation failed.

Notes: If a new IVP is to be solved with a call to CVodeRelInit, 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.

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:

```c
int CVRootFn( {
  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.

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.

Notes: The user must allocate memory for the vector rootsfound.
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, \textsc{cvode} provides a banded preconditioner in the module \textsc{cvbandpre} and a band-block-diagonal preconditioner module \textsc{cvbandppre}.

5.8.1 A serial banded preconditioner module

This preconditioner provides a banded 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 \( t \) to generate a band matrix of bandwidth \( m_0 + m_1 + 1 \), where the number of super-diagonals \( (m_0, \) the upper half-bandwidth) and sub-diagonals \( (m_1, \) 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 \( \frac{\partial f}{\partial y} \) it may be a very crude approximation. The true Jacobian need not be banded, or its true bandwidth may be larger than \( m_0 + m_1 + 1 \), as long as the banded approximation generated here is sufficiently accurate to speed convergence as a preconditioner.

In order to use the \textsc{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 \textsc{cvbandpre} module, the main program must include the header file \textsc{cvbandpre} which declares the needed function prototypes. The following is a summary of the usage of this module and describes the arguments and optional inputs.

### Arguments

- \( t \): is the current value of the independent variable.
- \( y \): is the current value of the dependent variable vector, \( y(t) \).
- \( g \): is a pointer to user data — the same as the \( g \) parameter passed to \textsc{CVodeRootInit}.
- \( g \) \_\text{data} \: is a pointer to user data — the same as the \( g \) \_\text{data} parameter passed to \textsc{CVodeRootInit}.

### Return value

A \textsc{CVRootFn} should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and \textsc{CVode} returns \textsc{CVROOTFUCFAIL}).

### Notes

Allocation of memory for \( g \) \_\text{out} \ is handled within \textsc{CVode}.

### Using CVODE for C Applications

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

   - flag = \textsc{CVBPSpgmr}(\textsc{cvode} \_\text{mem}, \textsc{pretype}, \textsc{maxl}, \textsc{bp} \_\text{data});
   - flag = \textsc{CVBPSpgmc}(\textsc{cvode} \_\text{mem}, \textsc{pretype}, \textsc{maxl}, \textsc{bp} \_\text{data});
   - flag = \textsc{CVBPSpgmr}(\textsc{cvode} \_\text{mem}, \textsc{pretype}, \textsc{maxl}, \textsc{bp} \_\text{data});

Each function \textsc{CVBPSpgm} \_\text{r} or \textsc{CVBPSpgm} \_\text{c} is a wrapper around the corresponding specification function \textsc{CVBPSpg} \_\text{r} or \textsc{CVBPSpgm} \_\text{c} and performs the following actions:

- Attaches the \textsc{cvpilins} linear solver to the main \textsc{CVode} solver memory;
- Sets the preconditioner data structure for \textsc{cvbandpre};
- Sets the preconditioner setup function for \textsc{cvbandpre};
- Sets the preconditioner solve function for \textsc{cvbandpre};

The arguments \textsc{pretype} and \textsc{maxl} are described below. \textsc{The last argument of \textsc{CVBPSpgm} \_\text{r} is the pointer to the \textsc{cvbandpre} data returned by \textsc{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 \textsc{cvpilins} optional input functions.

9. **Advance solution in time**

10. **Deallocate memory for solution vector**

11. **Free the \textsc{cvbandpre} data structure**

   \textsc{CVBandPrecFree} (\textsc{bp} \_\text{data});

12. **Free solver memory**

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

---

### CVBandPrecAlloc

Call \( \textsc{bp} \_\text{data} = \textsc{CVBandPrecAlloc}(\textsc{cvode} \_\text{mem}, \textsc{N}, \textsc{nu}, \textsc{ml}); \)

Description The function \textsc{CVBandPrecAlloc} initializes and allocates memory for the \textsc{cvbandpre} preconditioner.

Arguments

- \textsc{cvode} \_\text{mem} (void * \_pointer to the \textsc{cvode} memory block.
- \textsc{N} (long int) problem dimension.
- \textsc{nu} (long int) upper half-bandwidth of the problem Jacobian approximation.
- \textsc{ml} (long int) lower half-bandwidth of the problem Jacobian approximation.

Return value If successful, \textsc{CVBandPrecAlloc} returns a pointer to the newly created \textsc{cvbandpre} memory block (of type void *). If an error occurred, \textsc{CVBandPrecAlloc} returns NULL.

Notes The banded approximate Jacobian will have its nonzeros only in locations \( (i,j) \) with \(-\textsc{ml} \leq j-i \leq \textsc{nu} \).

---

### CVBPSpgmr

Call \( \text{flag} = \textsc{CVBPSpgmr}(\textsc{cvode} \_\text{mem}, \textsc{pretype}, \textsc{maxl}, \textsc{bp} \_\text{data}); \)

Description The function \textsc{CVBPSpgmr} links the \textsc{cvbandpre} data to the \textsc{cvpilins} linear solver and attaches the latter to the \textsc{cvode} memory block.

Arguments

- \textsc{cvode} \_\text{mem} (void * \_pointer to the \textsc{cvode} memory block.
5.8 Preconditioner modules

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 CVBAND_PRE_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 cvspilsm initialization was successful.
CVSPILS_MSG_NULL The cvode_msg pointer is NULL.
CVSPILS_NULL_INPUT The preconditioner type pretype is not valid.
CVSPILS_NULL_FAIL A memory allocation request failed.
CVBANDPRE_DATA_NULL The cvbandpre preconditioner has not been initialized.

Call: flag = CVBpsbpcg(cvode_msg, pretype, maxl, bp_data);
Description: The function CVBpsbpcg links the cvbandpre data to the cvspilsm linear solver and
attaches the latter to the cvode memory block.
Arguments: cvode_msg (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 CVBAND_PRE_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 cvspilsm initialization was successful.
CVSPILS_MSG_NULL The cvode_msg pointer is NULL.
CVSPILS_NULL_INPUT The preconditioner type pretype is not valid.
CVSPILS_NULL_FAIL A memory allocation request failed.
CVBANDPRE_DATA_NULL The cvbandpre preconditioner has not been initialized.

Call: flag = CVBpsfpcg(cvode_msg, pretype, maxl, bp_data);
Description: The function CVBpsfpcg links the cvbandpre data to the cvspfpcg linear solver and
attaches the latter to the cvode memory block.
Arguments: cvode_msg (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 CVBAND_PRE_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 cvspilsm initialization was successful.
CVSPILS_MSG_NULL The cvode_msg pointer is NULL.
CVSPILS_NULL_INPUT The preconditioner type pretype is not valid.
CVSPILS_NULL_FAIL A memory allocation request failed.
CVBANDPRE_DATA_NULL The cvbandpre preconditioner has not been initialized.

Call: flag = 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:

Call: flag = CVBandPrecGetWorkSpace(bp_data, blenrwBP, bleniwBP);
Description: The function CVBandPrecGetWorkSpace returns the sizes of the cvbandpre real and
integer workspaces.
Arguments: bp_data (void *) pointer to the cvbandpre data structure.
blenrwBP (long int) the number of realtype values in the cvbandpre workspace.
bleniwBP (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_DATA_NULL The cvbandpre preconditioner has not been initialized.

Notes: In terms of problem size N, and smu + min(N - 1, smu + 1), the actual size of the real
workspace is (2N + mu + 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.

Call: flag = CVBandPrecGetNumRhsEvals(bp_data, nfevalsBP);
Description: The function CVBandPrecGetNumRhsEvals returns the number of calls to the user right-hand side function.
Arguments: bp_data (void *) pointer to the cvbandpre data structure.
nfevalsBP (long int) the number of realtype values in the cvbandpre workspace.

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

CV_SUCCESS The optional output value has been successfully set.

Notes: The counter nfevalsBP is distinct from the counter nfevals returned by the corresponding
CVsolveGetNumRhsEvals function, and also from nfevals returned by
CVodeGetNumRhsEvals. The total number of right-hand side function evaluations is
the sum of all these counts.

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 Preconditioner modules

5.8.2 A parallel band-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 [36] and is included in a software module within the CVODE package. This module works with the parallel vector module xvec彤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 local 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 \( y_m \) depends on \( y_m \) and also on components of blocks \( y_m \) associated with neighboring subdomains (so-called ghost-cell data). Let \( y_m \) denote \( y_m \), augmented with those other components on which \( y_m \) depends. Then we have

\[
g(t, y) = [y(t, y_1), y(t, y_2), \ldots, y(t, y_M)]^T
\]

and each of the blocks \( g_m(t, y_m) \) is uncoupled from the others.

The preconditioner associated with this decomposition has the form

\[
P = \text{diag}[P_1, P_2, \ldots, P_M]
\]

where

\[
P_m = I - \gamma J_m
\]

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 \( m_u \) and \( m_d \) defined as the number of non-zero diagonals above and below the main diagonal, respectively. The difference quotient approximation is computed using \( m_u + m_d + 2 \) evaluations of \( g_m \), but only a matrix of bandwidth \( m_u + m_d + 1 \) is retained.

Neither pair of parameters need be the true half-bandwidths of \( J_m \). 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 \( m_u \) and \( m_d \) while keeping \( m_u + m_d + 1 \) 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

\[
PX = b
\]

reduces to solving each of the equations

\[
P_m x_m = b_m
\]

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 \texttt{CVBBDPre} module calls two user-provided functions to construct \( P \): a required function \texttt{glc()} (of type \texttt{CVLocalFn}) which approximates the right-hand side function \( g(t, y) = f(t, y) \) and if it is computed locally, and an optional function \texttt{cfm()} (of type \texttt{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 \texttt{fdata} as that passed by the user to \texttt{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 \texttt{fdata}) for components of \( y \) that are communicated by \texttt{cfm} from the other processors, and that are then used by \texttt{glc} to do any communication.

\begin{verbatim}
CVLocalFn

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

Purpose This function computes \( g(t, y) \). It loads the vector \texttt{glocal} as a function of \( t \) and \( y \).

Arguments

\begin{itemize}
  \item \texttt{Nlocal} is the local vector length.
  \item \texttt{t} is the value of the independent variable.
  \item \texttt{y} is the dependent variable.
  \item \texttt{glocal} is the output vector.
  \item \texttt{fdata} is a pointer to user data — the same as the \texttt{fdata} parameter passed to \texttt{CVodeSetFdata}.
\end{itemize}

Return value A \texttt{CVLocalFn} should return 0 if successful, a positive value if a recoverable error occurred (in which case \texttt{CVode} will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and \texttt{CVode} returns \texttt{CVJSETUPFAIL}).

Notes This function assumes that all inter-processor communication of data needed to calculate \texttt{glocal} has already been done, and this data is accessible within \texttt{fdata}.

The case where \( g \) is mathematically identical to \( f \) is allowed.
\end{verbatim}

\begin{verbatim}
CVCommFn

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

Purpose This function performs all inter-processor communications necessary for the execution of the \texttt{glc} function above, using the input vector \texttt{y}.

Arguments

\begin{itemize}
  \item \texttt{Nlocal} is the local vector length.
  \item \texttt{t} is the value of the independent variable.
  \item \texttt{fdata} is a pointer to user data — the same as the \texttt{fdata} parameter passed to \texttt{CVodeSetFdata}.
\end{itemize}

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

Notes The \texttt{cfm} function is expected to save communicated data in space defined within the structure \texttt{fdata}.

Each call to the \texttt{cfm} function is preceded by a call to the right-hand side function \( f \) with the same \( (t, y) \) arguments. This \texttt{cfm} can omit any communications done by \( f \) if relevant to the evaluation of \texttt{glocal}. If all necessary communication was done in \( f \), then \texttt{cfm} = \texttt{NULL} can be passed in the call to \texttt{CVodeSetComm} (see below).
5.8 Preconditioner modules

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_bbddpre.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 = CVBBDSpsgmr(cvode_mem, pretype, maxl, bbd_data);`
   `flag = CVBBDSpbcg(cvode_mem, pretype, maxl, bbd_data);`
   `flag = CVBBDpMfpmr(cvode_mem, pretype, maxl, bbd_data);`
The function `CVBBDp*` is a wrapper around the corresponding specification function `CVSP*` and performs the following actions:
   - Attaches the `CVSPILS` linear solver to the `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 = unit roundoff`, which can be specified by passing `dqrely = 0.0`.
   - `gloc` (CVLocalFn) the C function which computes the approximation `y(t, y)`.
   - `cfn` (CVCommFn) the optional C function which performs all inter-process communication required for the computation of `y(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` is used 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 y, 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.

   **`CVBBDSpsgmr`**

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

   Description
   The function `CVBBDSpsgmr` links the `cvbbdpre` data to the `CVSPILS` 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
5.8 Preconditioner modules

CVSPILS_SUCCESS The CVSPILS initialization was successful.
CVSPILS_MEM_NULL The cvode_mem pointer is NULL.
CVSPILS_DLL_INPUT The preconditioner type pretype is not valid.
CVSPILS_DLL_FAIL A memory allocation request failed.
CVBBDPRE_DATA_NULL The CVBBDPRE preconditioner has not been initialized.

CVBBDspbcg
Call flag = CVBBDspbcg(cvode_mem, pretype, maxl, bbd_data);
Description The function CVBBDspbcg links the cvmmrun data to the CVSPILS 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 CVSPILS initialization was successful.
CVSPILS_MEM_NULL The cvode_mem pointer is NULL.
CVSPILS_DLL_INPUT The preconditioner type pretype is not valid.
CVSPILS_DLL_FAIL A memory allocation request failed.
CVBBDPRE_DATA_NULL The CVBBDPRE preconditioner has not been initialized.

CVBBDspfmr
Call flag = CVBBDspfmr(cvode_mem, pretype, maxl, bbd_data);
Description The function CVBBDspfmr links the cvmmrun data to the CVSPILS 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 CVSPILS initialization was successful.
CVSPILS_MEM_NULL The cvode_mem pointer is NULL.
CVSPILS_DLL_INPUT The preconditioner type pretype is not valid.
CVSPILS_DLL_FAIL A memory allocation request failed.
CVBBDPRE_DATA_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 cvmmrun 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_j, mudq, or mldq. After solving one problem, and after calling CVBBDPrecReinit to re-initialize CVBBDPRE 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 CVBpgrn, CVBpbcg, or CVBptfmr, and/or one or more of the corresponding CVSp** functions, must also be made.

CVBBDspfcg
Call flag = CVBBDspfcg(bbd_data, mudq, mldq, dqrely, gloc, cfn);
Description The function CVBBDspfcg initializes the CVBBDPRE preconditioner.
Arguments bbd_data (void **) pointer to the CVBBDPRE data structure.
mudq (long int) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
mldq (long int) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
dqrely (realtype) the relative increment in components of y used in the difference-quotient approximations. The default is dqrely = unit roundoff, which can be specified by passing dqrely = 0.
gloc (CVLocalFn) the C function which computes the approximation g(y) = f(t,y) / f(t).
cfn (CVCommFn) the optional C function which performs all inter-process communication required for the computation of g(y).
Return value The return value flag (of type int) is one of
CVBBDPRE_SUCCESS The reinitialization was successful.
CVBBDPRE_DATA_NULL The bbd_data argument is NULL.
Notes If one of the half-bandwidths mudq or mldq is negative or exceeds the value local_j – 1, it is replaced with 0 or local_j – 1 accordingly.

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

CVBBDspfcg
Call flag = CVBBDspfcg(bbd_data, lvrealxVBDP, lvrealxVBDP);
Description The function CVBBDspfcg returns the local CVBBDPRE real and integer workspace sizes.
Arguments bbd_data (void **) pointer to the CVBBDPRE data structure.
lvrealxVBDP (long int) local number of realtype values in the cvbbdpre workspace.
lvrealxVBDP (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_DATA_NULL The cvbbdpre preconditioner has not been initialized.
Notes In terms of local_j and smu = min(local_j – 1, mudq + mldq), the actual size of the real workspace is (2 * mldq + mudq + smu + 2) * local_j * realtype words, and the actual size of the integer workspace is local_j * integer words. These values are local to the current processor.
The workspaces referred to here exist in addition to those given by the corresponding CVSpGetWorkspace function.

```c
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.
```

```c
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
  - FVInitS (defined by nvector_serial) interfaces to N_VectorEmpty_Serial.
  - FVInitP (defined by nvector_parallel) interfaces to N_VectorEmpty_Parallel.

- Interface to the main CVODE module
  - FCVmalloc interfaces to CVodeCreate, CVodeGetDate, and CVodeMalloc.
  - FCVreInit interfaces to CVodeReInit.
  - FCVsetJln and FCVsetTRIN interface to CVodeSet* functions.
  - FCVsetBET interfaces to CVodeSetBETFn.
  - FCVode interfaces to CVode, CVodeGet* functions, and to the optional output functions for the selected linear solver module.
  - FCVdiy interfaces to the interpolated output function CVodeGetDky.
  - FCVsetErrWt interfaces to CVodeGetErrWeights.
  - FCVtestLocalError interfaces to CVodeGetTestLocalErrors.
  - FCVfree interfaces to CVodeFree.

- Interface to the linear solver modules
  - FCVdijg interfaces to CVdiy.
  - FCVDenseJ interfaces to CVdense.
  - FCVdenseSetJac interfaces to CVdenseSetJacFn.
  - FCVband interfaces to CVband.

- Interface to the auxiliary functions
  - FCVsetJtimes interfaces to CVMtimes.
  - FCVsetJtimes vec interfaces to CVJtimesVecFn.
  - FCVsetJac interfaces to CVSpilsJacTimesVecFn.

- Interface to the dense linear solver
  - FCDenseJ interfaces to CVDenseJac.

- Interface to the sparse linear solver
  - FCVBandJ interfaces to CVBandJac.

- Interface to the direct solver
  - FCVBandJ interfaces to CVBandJacFn.

- Interface to the preconditioner
  - FCVBandPrec interfaces to CVBandPrecSetupFn.

- Interface to the dense linear solver
  - FCVDenseSetJac interfaces to CVBandSetJacFn.

- Interface to the direct solver
  - FCVDenseSetJ interfaces to CVBandSetJ.

- Interface to the sparse linear solver
  - FCVDenseSetJac interfaces to CVBandSetJacFn.

- Interface to the direct solver
  - FCVDenseSetJ interfaces to CVBandSetJ.

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, fcvsys.h, and fcvsbb.h. By default, these mapping definitions depend on the C macro _F77_UNDERSCORE defined in the header file sundials_config.h by configure. However, the set of flags SUNDIALS_CASE_UPPER, SUNDIALS_CASE_LOWER, SUNDIALS_UNDERSCORE_ONE, SUNDIALS_UNDERSCORE_ONE are explicitly defined in the header file sundials_config.h when configuring sundials via the options --with-f77underscore and --without-f77underscore and --with-f77case to override the default behavior if necessary (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 (NED, NLOCAL, NGLOBAL), Jacobian half-bandwidths (ML, MU, 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 cvode 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 FCVPHIL(T, Y, YDOT, IPAR, RPAR, IER)

where T is the initial value of t, Y is a real array of length 6 for real optional outputs. RPAR is an integer array of length 21 for integer optional outputs. IPAR is a real array of length 6 for real optional outputs. RTOL is an initial value of the relative tolerance (scalar), ATOL is the absolute tolerance (scalar or array). IATOL specifies the type for absolute tolerance ATOL: 1 for scalar or 2 for array. ITEMPER, ATOL, RTOL, and ATOL are ignored and the user is expected to subsequently call FCVEWT and provide the function FCVWT.

2. nvector module initialization

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

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

where COMM is a return communicator. COMM must be a valid MPI communicator. KEY is a real array of real optional outputs. 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.

[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_Comp_f2c function), then COMM can be any valid MPI communicator. Otherwise, MPI_Comp_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, the user must make the following call:

CALL FCVALLOC(T0, Y0, METH, ITEMPER, ATOL, RTOL, ATOL, & IOTOL, IROOT, IPAR, RPAR, IER)

Call

FCVALLOC

Call

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

Description

Arguments

T0 is the initial value of t. Y0 is an array of initial conditions.
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 spgmr 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, DJAC, H, IPAR, RPAR, &
  WR1, WR2, WR3, IER)
DIMENSION Y(*), FY(*), DJAC(NEQ,*), IPAR(*), RPAR(*), &
  WR1(*), WR2(*), WR3(*)
```

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 FCVODE. The vectors WR1, WR2, and WR3 of length NEQ are provided as work space for use in FCVODE. 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 FCVODE will attempt to correct, or a negative value if FCVODE failed unrecoverably (in which case the integration is halted).

If the user's FCVDJAC uses difference quotient approximations, it may need to use the weight array RWR and current stepsize $h$ in the calculation of suitable increments. The array IER can be obtained by calling FCVDERRWRT using one of the work arrays as temporary storage for RWR. It may also need the unit roundoff, which can be obtained as the optional output EWT.

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

```
CALL FCVSVDEENSEJAC(FLAG, IER)
```

with FLAG ≠ 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 FCVBANDJAC(NEQ, MU, ML, MDIM, T, Y, BJAC, H, IPAR, RPAR, &
  WR1, WR2, WR3, IER)
```

Typically this routine will use only NEQ, MU, ML, T, Y, and BJAC. It must load the MDIM by $R$ array BJAC with the Jacobian matrix at the current $(t,y)$ in band form. Store in BJAC($k,j$) the Jacobian element $J_{ij}$ with $k = i - j + MU + 1$ ($k = 1 \ldots MU + 1$) and $j = 1 \ldots N$. 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 FCVODE. The vectors WR1, WR2, and WR3 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 FCVODE will attempt to correct), or a negative value if FCVBJAC failed unrecoverably (in which case the integration is halted).

If the user’s FCVBANDJAC uses difference quotient approximations, it may need to use the weight array RWR and current stepsize $h$ in the calculation of suitable increments. The array IER can be obtained by calling FCVDERRWRT using one of the work arrays as temporary storage for RWR. It may also need the unit roundoff, which can be obtained as the optional output EWT.

If the FC
dBandJAC routine is provided, then, following the call to FCVBAND, the user must make the call:

```
CALL FCVBANDSETJAC(FLAG, IER)
```

with FLAG ≠ 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 Scalded Preconditioned GMRES solution of the linear systems, the user must make the call:

```
CALL FCVSPOPSR(IPRETYPE, IGSTYPE, MAXL, DLT, 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. DLT is the linear convergence tolerance factor. For all 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 Scalded Preconditioned Bi-CGstab solution of the linear systems, the user must make the call:

```
CALL FCVSPBCG(IPRETYPE, MAXL, DLT, IER)
```

Its arguments are the same as those with the same names for FCVSPOPSR.

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.
The preconditioner (or the product of the left and right preconditioners if both are nontrivial) should be an approximation to the matrix \( J - \gamma I \), where \( I \) is the identity matrix, \( J \) is the system Jacobian, and \( \gamma = GAMA. \) 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:

```fortran
SUBROUTINE 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 FCVSPILSSETJAC(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 FCVPSET and FCVPSET (see below).

### User-supplied routines for SPGMR/SPBCG/SPTFQMR

With treatment of the linear systems by any of the Krylov iterative solvers, there are three optional user-supplied routines — FCVJTIMES, FCVPILSOL, and FCVPILSSET. 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:

```fortran
SUBROUTINE FCVJTIMES(T, Y, FY, GAMMA, DELTA, LR, IPAR, RPAR, WORK, IER)
  
  Typcally this routine will use only NEQ, T, Y, and FY. It must compute the product vector \( Jv \), where the vector \( v \) is stored in \( V \), and store the product in \( FY \). 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:

```fortran
SUBROUTINE FCVPSOL(T, Y, FY, R, Z, GAMMA, DELTA, LR, IPAR, RPAR, WORK, IER)
  
  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 \( LR = 2 \).
```

### Example Usage of the FCVODE Interface Module

5. **Usage of the FCVODE interface module**

For the Scaled Preconditioned Transpose-Row Quasi-Minimal Residual solution of the linear systems, the user must make the call

```fortran
CALL FCVSPSPTFQMR(IPRETYPE, MAIL, DELT, IER)
```

Its arguments are the same as those for FCVSPQMR.

Optional outputs specific to the SPTFQMR case are listed in Table 6.2

For descriptions of the relevant optional user-supplied routines, see below.

### Functions used by SPGMR/SPBCG/SPTFQMR

An optional user-supplied routine, FCVJTIMES (see below), can be provided for Jacobian-vector products. If it is, then, following the call to FCVPSOL, FCVPILSOL, or SPTFQMR, the user must make the call:

```fortran
CALL FCVPSILSETJAC(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 FCVPSILSETJAC(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 FCVPILSOL and FCVPILSSET, respectively. On return, set \( IER = 0 \) if FCVPILSOL 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, and WORK are work arrays of length NEQ for use by this routine.

If the user calls FCVPSILSETJAC the routine FCVJTIMES must be provided, even if it is not needed and must return IER=0.

### Notes

(a) If the user’s FCVJTIMES or FCVPILSOL routine uses difference quotient approximations, it may need to use the error weight array EWT, the current stepsizes \( \delta \), and/or the unit roundoff, in the calculation of suitable increments. Also, if FCVPILSOL uses an iterative method in its solution, the residual vector \( r = r - Pz \) of the system should be made less than \( \Delta \) in weighted \( f_2 \) norm, i.e. \( \sum_{i=1}^{NEQ} \sqrt{EWT(i)[r(i) - \Delta]} < \Delta \).

(b) If needed in FCVPSOL, FCVPILSOL, or FCVJTIMES, 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, FCVPILSOL, or FCVJTIMES, the unit roundoff can be obtained as the optional output RTOL(6) (available after the call to FCVMALLOC) and can be passed using either the RPAR user data array or a common block.

### Problem solution

Carrying out the integration is accomplished by making calls as follows:

```fortran
CALL FCVODE(T0, T, Y, ITASK, IER)
```

The arguments are as follows. TD0 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 TD0 and interpolate), to 2 for one-step mode (return after each internal step taken), to 3 for
normal mode with the additional `tstep` constraint, or to 4 for one-step mode with the additional constraint `tstep`. 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 `IDT` 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:

```fortran
CALL FCVREINIT(T, K, SKY, IER)
```

where `T` is the value of `t` at which solution derivative is desired, and `K` is the derivative order (0 ≤ K ≤ QU). On return, SKY is an array containing the computed K-th derivative of y. The value `T` must lie between TCIK − HS and TCIK. The return flag IER is set to 0 upon successful return or to a negative value to indicate an illegal input.

7. Problem reinitialisation

To re-initialise the CVode solver for the solution of a new problem of the same size as one already solved, make the following call:

```fortran
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

```fortran
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`. In the case of SPBCG, for a change in any inputs, make the call

```fortran
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`.

8. Memory deallocation

To free the internal memory created by the call to `FCVMALLOC`, make the call

```fortran
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:

```fortran
CALL FCVSETIIN(KEY, IVAL, IER)
CALL FCVSETRIN(KEY, RVAL, IER)
```

where KEY is a quoted string indicating which optional 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, `IDT` (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 `IDT` 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:

  ```fortran
  CALL FCVGETERRWEIGHTS (EWT, IER)
  ```

- To compute the return error flag array `IER`, containing the derivative order (0 ≤ K ≤ QU). On return, SKY is an array containing the computed K-th derivative of y. The value `T` must lie between TCIK − HS and TCIK. The return flag IER is set to 0 upon successful return or to a negative value to indicate an illegal input.

- Where IER is an integer return flag which is set to 0 on success and a negative value if a failure occurred.

- For more details on the optional inputs and outputs, see §5.5.5 and §5.5.7.

- To obtain the error weight array `EWT`, containing the multiplicative error weights used the WRMS norms, make the following call:

  ```fortran
  CALL FCVGETERRWEIGHTS (EWT, IER)
  ```

- To compute the return error flag array `IER`, containing the derivative order (0 ≤ K ≤ QU). On return, SKY is an array containing the computed K-th derivative of y. The value `T` must lie between TCIK − HS and TCIK. The return flag IER is set to 0 upon successful return or to a negative value to indicate an illegal input.

- Where IER is an integer return flag which is set to 0 on success and a negative value if a failure occurred.

- For more details on the optional inputs and outputs, see §5.5.5 and §5.5.7.
Table 6.2: Description of the FCVODE optional output arrays IOUT and ROOT

<table>
<thead>
<tr>
<th>Optional output</th>
<th>CVODE function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>CVODE name solve</td>
</tr>
<tr>
<td>1</td>
<td>LEWS</td>
</tr>
<tr>
<td>2</td>
<td>LEWS</td>
</tr>
<tr>
<td>3</td>
<td>RT</td>
</tr>
<tr>
<td>4</td>
<td>NFE</td>
</tr>
<tr>
<td>5</td>
<td>NRTF</td>
</tr>
<tr>
<td>6</td>
<td>NCFN</td>
</tr>
<tr>
<td>7</td>
<td>NRTF</td>
</tr>
<tr>
<td>8</td>
<td>NRTF</td>
</tr>
<tr>
<td>9</td>
<td>QU</td>
</tr>
<tr>
<td>10</td>
<td>QCUR</td>
</tr>
<tr>
<td>11</td>
<td>UMOR</td>
</tr>
<tr>
<td>12</td>
<td>UGE</td>
</tr>
</tbody>
</table>

CVDense linear solver

<table>
<thead>
<tr>
<th>Integer output array IOUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEWS</td>
</tr>
<tr>
<td>LEWS</td>
</tr>
<tr>
<td>RT</td>
</tr>
<tr>
<td>NFE</td>
</tr>
<tr>
<td>NRTF</td>
</tr>
<tr>
<td>NRTF</td>
</tr>
<tr>
<td>NRTF</td>
</tr>
<tr>
<td>QU</td>
</tr>
<tr>
<td>QCUR</td>
</tr>
<tr>
<td>UMOR</td>
</tr>
<tr>
<td>UGE</td>
</tr>
</tbody>
</table>

CVBand linear solver

<table>
<thead>
<tr>
<th>Real output array ROOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

To obtain the estimated local errors, following a successful call to FCVODE, 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 FCWALLOC but prior to calling FCVODE, to allocate and initialize memory for the FCVROOT module:

CALL FCVROOTINIT (NRTFN, IER)

To specify the functions whose roots are to be found, the user must define the following routine:

SUBROUTINE FCVROOTFN (T, Y, G, IPAR, RPAR, 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 free the memory resources allocated by a prior call to FCVROOTFREE, the following call must be made:

CALL FCVROOTFREE (RTPS, IER)

The arguments are as follows: RTPS is the number of root functions. IER is a return completion flag; its values are 0 for success, -1 if the FCVROOT memory was NULL, and -11 if a memory allocation failed.

For additional information on the rootfinding feature, see §5.7.
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 CVSPGMR and SPGMR optional input functions.
- FCVBPSPILSSETJAC interfaces to CVBandPrecInit.
- FCVBPSPDOPT interfaces to CVBANDPRE optional output functions.
- FCVBPSPFREE 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 module initialization 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 FCVBPSPGMR (see step 4 in §6.2).

   - To specify the SPCCG linear system solver and use the CVBANDPRE preconditioner, make the following call:
     
     CALL FCVBPSPCCG(IPRETYPE, MAXL, DELT, IER)

     Its arguments are the same as those of FCVBPSPCCG (see step 4 in §6.2).

5. Problem solution

6. CVBANDPRE Optional outputs

Optional outputs specific to the SPGMR, SPCCG, or SPFCGMR solver are listed in Table 6.2. To obtain the optional outputs associated with the CVBANDPRE module, make the following call:

   CALL FCVBPSPDOPT(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 FCVBPSPFREE

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 CVBANDPRE functions, are as follows:

- FCVBBDINIT interfaces to CVBBDprecAlloc.
- FCVBBDSPGMR interfaces to CVBBDSPGMR and SPGMR optional input functions.
- FCVBBDSPCCG interfaces to CVBBDSPCCG and SPCCG optional input functions.
- FCVBBDSPFMR interfaces to CVBBDSPFMR and SPFMR optional input functions.
- FCVBBDSPDOPT interfaces to CVBBDprecInit.
- FCVBBDSPDOPT interfaces to CVBBDprecOptional output functions.
- FCVBBDSPFREE 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 CVBANDPRE or CVODE):
6.6 Usage of the FCVBBD interface to CVBBDPRE

<table>
<thead>
<tr>
<th>FCVBBD routine (Fortran)</th>
<th>CVODE function (C)</th>
<th>CVODE function type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCVBBDNS</td>
<td>FCVgluc</td>
<td>CVlglucFn</td>
</tr>
<tr>
<td>FCVBBMP</td>
<td>FCVfn</td>
<td>CVfnSp</td>
</tr>
<tr>
<td>FCVJtimes</td>
<td>FCVtimes</td>
<td>CVspilsJacTimesVecFn</td>
</tr>
</tbody>
</table>

As with the rest of the CVODE 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:
     
     ```fortran
     CALL FCVBBDINIT(NLOCAL, MUDQ, MLQ, MU, DQRELY, IER)
     ```
   The arguments are as follows: NLOCAL is the local size of vectors on this processor. MUDQ and MLQ 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 MLQ. DQRELY is the relative increment factor in y for difference quotients (optional). A value of 0.0 indicates the default, \( y_{\text{increment}} = 0.0 \). 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:

   ```fortran
   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:

   ```fortran
   CALL FCVBBDSPBCG(IPRETYPE, MAXL, DELT, IER)
   ```
   Its arguments are the same as those of FCVSPBCG (see step 4 in §6.2).

   To specify the spgmr linear system solver and use the cvbbdpre preconditioner, make the following call:

   ```fortran
   CALL FCVBBDSPGMR(IPRETYPE, MAXL, DELT, IER)
   ```
   Its arguments are the same as those of FCVSPGMR (see step 4 in §6.2).

   Optionally, to specify that spgmr, spbcg, or spgmr should use the supplied FCVJtimes, make the call

   ```fortran
   CALL FCVSPILSSETJAC(FLAG, IER)
   ```

5. Problem solution

6. CVODE Optional outputs

   Optional outputs specific to the spgmr, spbcg, or spgmr solver are listed in Table 6.2. To obtain the optional outputs associated with the cvbbdpre module, make the following call:

   ```fortran
   CALL FCVBBDEOPT(LENBWBBD, LENIWBBD, NGEHBBD)
   ```
   The arguments returned are as follows. LENBWBBD is the length of real preconditioner work space, in realtype words. LENIWBBD is the length of integer preconditioner work space, in integer words. NGEHBBD is the number of \( g(t,y) \) evaluations (calls to FCVLOGCF) 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 spgmr) in combination with the cvbbdpre preconditioner, then the cvbbdpre package can be re-initialized for the second and subsequent problems by calling FCVBBDREINIT, following which a call to FCVBBDEINIT may or may not be needed. If the input arguments are the same, no FCVBBDEINIT call is needed. If there is a change in input arguments other than \( MU \) or \( ML \), then the user program should make the call

   ```fortran
   CALL FCVBBDEINIT(MUDQ, MLQ, DQRELY, IER)
   ```

   This reinitializes the cvbbdpre preconditioner, but without reallocating its memory. The arguments of the FCVBBDEINIT routine have the same names and meanings as those of FCVBBDEINIT. If the value of \( MU \) or \( ML \) is being changed, then a call to FCVBBDEINIT must be made. Finally, if there is a change in any of the linear solver inputs, then a call to FCVBBDSPGMR, FCVBBDSPBCG, or FCVBBDSPGMR 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 FCVBBDEINIT, before calling FCVFREE, the user must make the call

   ```fortran
   CALL FCVFREE()
   ```

9. User-supplied routines

   The following two routines must be supplied for use with the cvbbdpre module:

   ```fortran
   SUBROUTINE FCVLOGCFN(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 FCVALLOC. 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 FCVLOGCFN failed uncontrollably (in which case the integration is halted).

   ```fortran
   SUBROUTINE FCVCOMMF(NLOC, T, YLOC, GLOC, IPAR, RPAR, IER)
   DIMENSION YLOC(*), IPAR(*), RPAR(*)
   ```
   with FLAG \( \neq 0 \) (see step 4 in §6.2 for details).
This routine is to perform the inter-processor communication necessary for the FCVGLCPR routine. Each call to FCVGLCPR 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 FCVGLCPR 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 FCVGLCPR failed unrecoverably (in which case the integration is halted).

The subroutine FCVGLCPR 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 (*nvmult)(N_Vector, long int *, long int *);
    realtype (*nvgetarraypointer)(N_Vector);
    realtype (*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 (*nvnorm)(N_Vector);
    realtype (*nvnormnrmask)(N_Vector, N_Vector, N_Vector);
    realtype (*nvninm)(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
.

realtype (*nvwil2norm)(N_Vector, N_Vector);
realtype (*nvil1norm)(N_Vector);
void (*nvcompare)(realtype, N_Vector, N_Vector);
booltype (*nvinvtest)(N_Vector, N_Vector);
booltype (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
realtype (*nvminquotient)(N_Vector, N_Vector);

100 Description of the NVECTOR module

}
Table 7.1: Description of the NVECTOR operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VCreate</td>
<td>v = N_VCreate(); 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.</td>
</tr>
<tr>
<td>N_VCloseEmpty</td>
<td>v = N_VCloseEmpty(); 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.</td>
</tr>
<tr>
<td>N_VDestroy</td>
<td>N_VDestroy(v); Destroys the N_Vector v and frees memory allocated for its internal data.</td>
</tr>
<tr>
<td>N_VSpace</td>
<td>N_VSpace(nvSpec, nrv, dlv); Returns storage requirements for one N_Vector. liv contains the number of realtype words and liv contains the number of integer words.</td>
</tr>
<tr>
<td>N_VGetArrayPointer</td>
<td>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.</td>
</tr>
<tr>
<td>N_VSetArrayPointer</td>
<td>vdata = 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.</td>
</tr>
<tr>
<td>N_VLinearSum</td>
<td>N_VLinearSum(x, 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 ).</td>
</tr>
<tr>
<td>N_VConst</td>
<td>N_VConst(c, z); Sets all components of the N_Vector z to c; ( z_i = c, i = 0, \ldots, n-1 ).</td>
</tr>
<tr>
<td>N_VProd</td>
<td>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 ). The y may not be tested for 0 values. It should only be called with an x that is guaranteed to have all nonzero components.</td>
</tr>
<tr>
<td>N_VDiv</td>
<td>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 ). ( 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.</td>
</tr>
<tr>
<td>N_VScale</td>
<td>N_VScale(c, x, z); Scales the N_Vector x by the scalar c and returns the result in z; ( z_i = cx_i ), ( i = 0, \ldots, n-1 ).</td>
</tr>
<tr>
<td>N_VAbs</td>
<td>N_VAbs(x, z); Sets the components of the N_Vector x to be the absolute values of the components of the N_Vector x; ( y_i =</td>
</tr>
<tr>
<td>N_VInv</td>
<td>N_VInv(x, z); Sets the components of the N_Vector x to be the inverses of the components of the N_Vector x; ( z_i = 1 / x_i ), ( i = 0, \ldots, n-1 ). This routine may not check for division by 0. It should be called only with an x which is guaranteed to have all nonzero components.</td>
</tr>
<tr>
<td>N_VDotProd</td>
<td>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 ).</td>
</tr>
<tr>
<td>N_VMaxNorm</td>
<td>m = N_VMaxNorm(x); Returns the maximum norm of the N_Vector x; ( m = \max</td>
</tr>
<tr>
<td>N_VWmaxNorm</td>
<td>m = N_VWmaxNorm(x, w); Returns the weighted maximum norm of the N_Vector x with weight vector w (</td>
</tr>
<tr>
<td>N_VWrmsNorm</td>
<td>m = N_VWrmsNorm(x, w); Returns the weighted root-mean-square norm of the N_Vector x with weight vector w (</td>
</tr>
<tr>
<td>N_VWrmsNormMask</td>
<td>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{\sum_{i=0}^{n-1} (x_i w_i)^2} /</td>
</tr>
<tr>
<td>N_VMin</td>
<td>m = N_VMin(x); Returns the smallest element of the N_Vector x; ( m = \min</td>
</tr>
<tr>
<td>N_VWl2Norm</td>
<td>m = N_VWl2Norm(x, w); Returns the weighted Euclidean l2 norm of the N_Vector x with weight vector w (</td>
</tr>
<tr>
<td>N_VLinNorm</td>
<td>m = N_VLinNorm(x); Returns the l1 norm of the N_Vector x; ( m = \sum_{i=0}^{n-1}</td>
</tr>
<tr>
<td>N_VCompare</td>
<td>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 = \begin{cases} 1.0 &amp; \text{if }</td>
</tr>
</tbody>
</table>

continued from last page
7.1 The NVECTOR_SERIAL implementation

The serial implementation of the nvector module provided with sundials, nvector_serial, defines the content field of NVector 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 macro gives access to the contents of the serial vector NVector.
  - The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial NVector content structure.

  Implementation:
  ```c
  #define NV_CONTENT_S(v) ( (N_VectorContent_Serial*)(v)->content )
  ```

- **NV_DATA_S, NV_DATA_S, NV_DATA_S, NV_LENGTH_S**
  - These macros give individual access to the parts of the content of a serial NVector.
    - The assignment `v_data = NV_DATA_S(v)` sets `v_data` to be a pointer to the first component of the data for the NVector `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

- **NV_Own_S**
  - This function creates and allocates memory for a serial NVector. Its only argument is the vector length.

  ```c
  N_Vector N_VNew_Serial(long int vec_length);
  ```

- **NV_OwnEmpty_Serial**
  - This function creates a new serial NVector with an empty (NULL) data array.

  ```c
  N_Vector N_VNewEmpty_Serial(long int vec_length);
  ```

- **NV_Make_Serial**
  - This function creates and allocates memory for a serial vector with user-provided data array.

  ```c
  N_Vector N_VMake_Serial(long int vec_length, realtype *v_data);
  ```

- **NV_CloneVectorArray_Serial**
  - This function creates (by cloning) an array of count serial vectors.

  ```c
  N_Vector *N_VCloneVectorArray_Serial(int count, N_Vector w);
  ```

- **NV_DestroyVectorArray_Serial**
  - This function frees memory allocated for the array of count variables of type NVector created with NV_CloneVectorArray_Serial or with NV_CloneVectorArrayEmpty_Serial.

  ```c
  void N_VDestroyVectorArray_Serial(N_Vector *vs, int count);
  ```

- **NV_Print_Serial**
  - This function prints the content of a serial vector to stdout.

  ```c
  void N_VPrint_Serial(N_Vector v);
  ```
## 7.2 The NVECTOR_PARALLEL implementation

The parallel implementation of the nvector module provides 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.

```c
struct N_VectorContent_Parallel {
  long int local_length;
  long int global_length;
  bool 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_VectorContent_Parallel.
  - Implementation:
    ```c
    #define NV_CONTENT_P(v) ( (N_VectorContent_Parallel)(v->content) )
    ```

- **NV_DATA_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 `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 call `NV_DATA_P(v)` returns the pointer to the first component of the local data for the N_Vector v.
  - Implementation:
    ```c
    #define NV_DATA_P(v) ( NV_CONTENT_P(v)->data )
    ```

- **NV_OWN_DATA_P**
  - This macro gives access to the local data array of an N_Vector.
  - The assignment `v_own_data = NV_OWN_DATA_P(v)` sets `v_own_data` to the length of the local part of v.
  - The call `NV_OWN_DATA_P(v)` returns the pointer to the local data array of the N_Vector v.
  - Implementation:
    ```c
    #define NV_OWN_DATA_P(v) ( (N_VectorContent_Parallel)(v->content)->own_data )
    ```

- **NV_LOCLENGTH_P**
  - These macros give access to the individual components of the local data array of an N_Vector.
  - The assignment `v_loclen = NV_LOCLENGTH_P(v)` sets `v_loclen` to be the length of the local part of v.
  - The call `NV_LOCLENGTH_P(v)` returns the length of the local part of v.
  - Implementation:
    ```c
    #define NV_LOCLENGTH_P(v) ( (N_VectorContent_Parallel)(v->content)->local_length )
    ```

- **NV_GLOBLENGTH_P**
  - This macro gives access to the local data array of an N_Vector.
  - 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)` returns the global length of the vector v.
  - Implementation:
    ```c
    #define NV_GLOBLENGTH_P(v) ( (N_VectorContent_Parallel)(v->content)->global_length )
    ```

- **NV_COMM_P**
  - This macro provides access to the MPI communicator used by the NVECTOR_PARALLEL vectors.
  - Implementation:
    ```c
    #define NV_COMM_P(v) ( (N_VectorContent_Parallel)(v->content)->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 call `NV_Ith_P(v,i)` returns the value of the i-th component of the local part of v.
  - Here i ranges from 0 to n-1, where n is the local length.
  - Implementation:
    ```c
    #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_VNewParallel**
  - This function creates and allocates memory for a parallel vector.
  - Implementation:
    ```c
    N_Vector N_VNewParallel(MPI_Comm comm, long int global_length);
    ```

- **N_VNewEmptyParallel**
  - This function creates a new parallel N_Vector with an empty (NULL) data array.
  - Implementation:
    ```c
    N_Vector N_VNewEmptyParallel(MPI_Comm comm, long int local_length, long int global_length);
    ```

- **N_VCloneVectorArrayParallel**
  - This function creates (by cloning) an array of count parallel vectors.
  - Implementation:
    ```c
    N_Vector N_VCloneVectorArrayParallel(int count, N_Vector v);
    ```

- **N_VCloneVectorArrayEmptyParallel**
  - This function creates (by cloning) an array of count parallel vectors, each with an empty (NULL) data array.
7.3 NVECT ORM 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 (cvspgmr, cvband, cvode, cvspils, cvbbdpre, cvspgmr, and cvspils, respectively). The cvband module, cvode solver, and cvspgmr are called within the implementation file.

Table 7.2: List of vector functions usage by cvode code modules

<table>
<thead>
<tr>
<th>Function</th>
<th>cvode</th>
<th>cvspgmr</th>
<th>cvband</th>
<th>cvode</th>
<th>cvspils</th>
<th>cvbbdpre</th>
<th>cvspgmr</th>
<th>cvspils</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VClone</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_Destroy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VPrint</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VMake</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VMakeEmpty</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VPrintParallel</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VGetArrayPointer</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VAddConst</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VPrint_Parallel</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VMin</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VPrint_N</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VCompare</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VAdd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VNorm</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VProd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VAbs</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VScale</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VTest</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VDiv</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VGetArrayPointer</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VMin</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VProd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VScale</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VTest</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VAbs</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VDiv</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VGetArrayPointer</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VMin</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VProd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VScale</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VTest</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VAbs</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VDiv</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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 = N_VDATA(v) and then access v_data[i] within the loop than it is to use N_VTh_i(v, i) within the loop.
- N_VMakeEmpty, N_VMakeParallel, and N_VCloneVectorArrayEmptyParallel set the field own_data = FALSE. N_Destroy, Parallel and N_VDestroyVectorArrayParallel 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 the vectors. It is the user’s responsibility to ensure that such routines are called with N_VVector arguments that were all created with the same internal representations.

7.3 NVECT ORM functions used by CVODE

The vector functions listed in Table 7.2 are:

- N_VClone
- N_Destroy
- N_VPrint
- N_VMake
- N_VMakeParallel
- N_VCloneVectorArrayEmptyParallel
- N_VDestroyVectorArrayParallel
- N_VPrintParallel

These functions are called within the implementation file.

**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 = N_VDATA(v) and then access v_data[i] within the loop than it is to use N_VTh_i(v, i) within the loop.
- N_VMakeEmpty, N_VMakeParallel, and N_VCloneVectorArrayEmptyParallel set the field own_data = FALSE. N_Destroy, Parallel and N_VDestroyVectorArrayParallel 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 the 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.
Chapter 8

Providing Alternate Linear Solver Modules

The central CVODE module interfaces with the linear solver module by way of calls to four routines. These are denoted here by limit, lsetup, lsolve, and lfree. Briefly, their purposes are as follows:

- limit: 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_limit, 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_lsetup 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 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 limit is

\[
\text{limit} \quad \text{Definition:} \quad \text{int (*limit)}(\text{cvodeMem cv_sam})
\]

Purpose: The purpose of limit is to complete initializations for specific linear solver, such as counters and statistics.

Arguments: cv_sam is the cvode memory pointer of type CVodeMem.

Return value: An limit 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

\[
\text{lsetup} \quad \text{Definition:} \quad \text{int (*lsetup)}(\text{cvodeMem cv_sam}, \text{int convfail}, \text{NVector ypred}, \text{NVector fpred}, \text{booleantype *jcurPtr}, \text{NVector vtemp1}, \text{NVector vtemp2}, \text{NVector vtemp3})
\]

Purpose: The job of lsetup is to prepare the linear solver for subsequent calls to lsolve. It may re-compute Jacobian-related data if it deems necessary.

Arguments: cv_sam 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 correction 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 correction 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 of ypred i.e. f(t_{n+1}, y_{n+1}).

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 NVector 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

```c
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 \( I - \gamma J \), \( J \) is the Jacobian of \( f \) at \( (t_n, y_{cur}) \) (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

```c
void (*lfree)(CVodeMem cv_mem);
```

Purpose
The routine lfree should free up any memory allocated by the linear solver.

Arguments
- \( cv\_mem \) 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 SPFRQMR package, which includes a solver for the scaled preconditioned Bi-CGSTab method.
- The DenseFactor 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, spfrqmr, and sptrqmr 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, cvspfrqmr, or cvsptrqmr solver.

9.1 The DENSE module

Relative to the sundials source tree, the files comprising the dense generic linear solver are as follows:

- header files (located in source/tree/shared/include)
  sundials dense h sundials smalldense h
  sundials types h sundials math h sundials config h
- source files (located in source/tree/shared/source)
  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 §2.5 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 NCONST, while the sundials math.h header file is needed for the ABS macro and Abs function.

The eight files listed above can be extracted from the sundials source tree 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 a size and a data field:

typedef struct {
  long int size;
  realtype **data;
} *DenseMat;

The size field indicates the number of columns (which is the same as the number of rows) of a dense matrix, while the data field is a two dimensional array used for component storage. The elements of a dense matrix are stored columnwise (ie columns are stored one on top of the other in memory). If A is of type DenseMat, then the (i,j)-th element of A (with \( 0 \leq i, j \leq \text{size} - 1 \)) is given by the expression \( (A->data)[j][i] \) or by the expression \( (A->data)[i][j] \). 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_{i,j}; or a_{j,i} = DENSE_ELEM(A,i,j);
  DENSE_ELEM references the (i,j)-th element of the \( N \times N \) DenseMat A, \( 0 \leq i, j \leq N - 1 \).

- DENSE_COL
  Usage: col_{j} = DENSE_COL(A,j);
  DENSE_COL references the j-th column of the \( N \times N \) DenseMat A, \( 0 \leq j \leq N - 1 \). The type of the expression DENSE_COL(A,j) is booleantype. After the assignment in the usage above, col_{j} may be treated as an array indexed from 0 to \( N - 1 \). The (i,j)-th element of A is referenced by col_{j}[i].

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 DenseFactor/DenseBacksolve;
9.1 The DENSE module

- DenseFactor: LU factorization with partial pivoting;
- DenseBacksolve: solution of $Ax = b$ using LU factorization;
- DenseZero: load a matrix with zeros;
- DenseCopy: copy one matrix to another;
- DenseScale: scale a matrix by a scalar;
- DenseAdd: increment a matrix by the identity matrix;
- DenseFreeMat: free memory for a DenseMat matrix;
- DenseFreePiv: free memory for a pivot array;
- DensePrint: print a DenseMat matrix to standard output.

9.1.4 Small Dense Matrix Functions

The following functions for small dense matrices are available in the DENSE package:

- denalloc
  denalloc(n) allocates storage for an $n \times n$ dense matrix. If $n$ has been successfully allocated, the denalloc returns NULL. The underlying type of the dense matrix returned is realtype*.
  If we allocate a dense matrix realtype* $a$ by $a = $ denalloc(n), a[i][j] references the $(i,j)$-th element of the matrix $a$, $0 \leq i,j < n-1$, and a[i] is a pointer to the first element in the $i$-th column of $a$. The location $a[0]$ contains a pointer to $n^2$ contiguous locations which contain the elements of $a$.

- denallocpiv
  denallocpiv(n) allocates an array of $n$ integers. It returns a pointer to the first element in the array if successful. It returns NULL if the memory request could not be satisfied.

- gefa
  gefa(a,n,p) factors the $n \times n$ dense matrix $a$. It overwrites the elements of $a$ with its LU factors and keeps track of the pivot rows chosen in the pivot array $p$.
  A successful LU factorization leaves the matrix $a$ and the pivot array $p$ with the following information:
  1. $p[k]$ contains the row number of the pivot element chosen at the beginning of the 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 a lower triangular matrix with all $1$'s on the diagonal, and $U$ is an upper triangular matrix, then the upper triangular part of $a$ (including its diagonal) contains $U$ and the strictly lower triangular part of $a$ contains the multipliers, $I - L$.
  gefa returns 0 if successful. Otherwise it encountered a zero diagonal element during the factorization. In this case it returns the column index (numbered from one) at which it encountered the zero.

- gesl
  gesl(a,n,p,b) solves the $n \times n$ linear system $ax = b$. It assumes that $a$ has been LU-factored and the pivot array $p$ has been set by a successful call to gefa(a,n,p). The solution $x$ is written into the $b$ array.

9.2 The BAND module

Relative to the sundials source tree, the files comprising the BAND generic linear solver are as follows:

- header files (located in source_tree/shared/include)
  sundials_band.h sundials_math.h sundials_config.h

- source files (located in source_tree/shared/source)
  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 §2.5 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 RCOPY, while the sundials_math.h header file is needed for the MIN, MAX, and ABS macros and RAABS function.

The six files listed above can be extracted from the sundials source tree and compiled by themselves into a BAND library or into a larger user code.
9.2 The BAND module

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:

```c
typedef struct {
    long int size;
    long int mu, ml, smu;
    realtype **data;
} *BandMat;
```

The fields in the above structure are:

- **size** is the number of columns (which is the same as the number of rows);
- **mu** is the upper half-bandwidth, $0 \leq \mu \leq \text{size} - 1$;
- **ml** is the lower half-bandwidth, $0 \leq \mu \leq \text{size} - 1$;
- **smu** is the storage upper half-bandwidth, $\mu \leq \text{smu} \leq \text{size} - 1$. The BandFactor routine writes the LU factors into the storage for A. The upper triangular factor U, however, may have an upper half-bandwidth as big as $\min(\text{size} - 1, \mu + \mu)$ 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

- $\text{data}[0]$ is a pointer to $(\text{smu} - \text{mu} + 1) \times \text{size}$ contiguous locations which hold the elements within the band of A.
- $\text{data}[\text{mu}]$ 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 $\text{smu} - \text{mu}$ (to access the uppermost element within the band in the j-th column) to $\text{smu} - \text{mu}$ (to access the lowest element within the band in the j-th column). Indices from 0 to $\text{smu} - \text{mu} - 1$ give access to extra storage elements required by BandFactor.
- $\text{data}[\text{mu} + \text{ml}]$ is the $(i, j)$-th element, $\mu - \text{ml} \leq i \leq j + \text{ml}$.

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: $\text{BAND_ELEM}(A, i, j) = a_{ij}$; or $a_{ij} = \text{BAND_ELEM}(A, i, j)$;
  BAND_ELEM references the $(i, j)$-th element of the $N \times N$ band matrix $A$, where $0 \leq i, j \leq N - 1$. The location $(i, j)$ should further satisfy $j - (A - \text{mu}) \leq i \leq j + (A - \text{mu})$.

---

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 BandFactor and BandBacksolve routines.
9.3 The SPGMR module

- *BAND_CUL*
  Usage: \( j = BAND \_CUL(A,j) \);
  \( BAND \_CUL \) 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 \( BAND \_CUL(A,j) \) is realtype. The pointer returned by the call \( BAND \_CUL(A,j) \) can be treated as an array which is indexed from \( -(k-u) \) to \( (k-u) \).

- *BAND_CUL_ELEM*
  Usage: \( a_{ij} = BAND \_CUL_ELEM(col_i, col_j) \);
  This macro references the \((i,j)\)-th entry of the band matrix \( A \) when used in conjunction with \( BAND \_CUL \) to reference the \( j \)-th column through \( col_j \). The index \((i,j)\) should satisfy \(- (k-u) \leq j \leq (k-u) \).

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 BandFactor/BandBacksolve;
- BandFactor: LU factorization with partial pivoting;
- BandBacksolve: solution of \( Ax = b \) using LU factorization;
- BandZero: load a matrix with zeros;
- BandCopy: copy one matrix to another;
- BandScale: scale a matrix by a scalar;
- BandAddI: increment a matrix by the identity matrix;
- BandFreeMat: free memory for a BandMat matrix;
- BandFreePiv: free memory for a pivot array;
- BandPrint: print a BandMat matrix to standard output.

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, contains auxiliary functions that support spgmr, as well as the other Krylov solvers in sundials (spbcgs and sptfqmr). For full details, including usage instructions, see the header files sundials\_spgmr.h and sundials\_iterative.h.

Relative to the sundials source\_free, the files comprising the spgmr generic linear solver are as follows:

- header files (located in source\_free/shared/include):
  sundials\_spgmr.h sundials\_iterative.h sundials\_vector.h
  sundials\_types.h sundials\_path.h sundials\_config.h

- source files (located in source\_free/shared/source):
  sundials\_spgmr.c sundials\_iterative.c sundials\_vector.c

Only two of the preprocessing directives in the header file sundials\_config.h are required to use the spgmr package by itself (see §2.5 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\_path.h header file is needed for the MAX and ABS macros and RAbs and RSign functions.

The generic \texttt{Spgmr} files, sundials\_vector.h, are needed for the definition of the generic \texttt{Spgmr} type and functions. The \texttt{Spgmr} functions used by the \texttt{spgmr} module are: \texttt{SpgmrProd}, \texttt{SpgmrLinear}, \texttt{SpgmrDiv}, \texttt{SpgmrConst}, \texttt{SpgmrClone}, \texttt{SpgmrCloneVectorArray}, \texttt{SpgmrDestroy}, \texttt{SpgmrDestroyVectorArray}.

The spgmr package can only be used in conjunction with an actual \texttt{Spgmr} implementation library, such as the \texttt{Spgmr\_serial} or \texttt{Spgmr\_parallel} provided with sundials.

The nine files listed above can be extracted from the sundials source\_free 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:

- \texttt{SpgmrMalloc}: allocation of memory for \texttt{Spgmr}\_\_\_\_\_solve;
- \texttt{Spgmr\_\_\_\_\_solve}: solution of \( Ax = b \) by the \texttt{spgmr} method;
- \texttt{Spgmr\_\_\_\_\_\_free}: free memory allocated by \texttt{Spgmr\_\_\_\_\_solve}.

The following functions are available in the support package sundials\_iterative.h:

- \texttt{ModifiedGS}: performs modified Gram-Schmidt procedure;
- \texttt{ClassicalGS}: performs classical Gram-Schmidt procedure;
- \texttt{QRfact}: performs QR factorization of Hessenberg matrix;
- \texttt{QRsol}: solves a least squares problem with a Hessenberg matrix factored by \texttt{QRfact}.

9.4 The SPBCG module

The spbcgs package, in the files sundials\_spbcgs.h and sundials\_spbcgs.c, includes an implementation of the scaled preconditioned Bi-CGSTab method. A separate code module, implemented in sundials\_iterative.h, contains auxiliary functions that support spbcgs, as well as the other Krylov solvers in sundials (spgmr and sptfqmr). For full details, including usage instructions, see the header files sundials\_spbcgs.h.

The spbcgs package can only be used in conjunction with an actual \texttt{spgmr} implementation library, such as the \texttt{Spgmr\_\_\_\_\_\_serial} or \texttt{Spgmr\_\_\_\_\_\_parallel} provided with sundials.

The files needed to use the spbcgs module by itself are the same as for the spgmr module, with sundials\_spbcgs.h replacing sundials\_spgmr.h.
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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_ADAMS</td>
<td>1 Adams-Moulton linear multistep method.</td>
</tr>
<tr>
<td>CV_BDF</td>
<td>2 BDF linear multistep method.</td>
</tr>
<tr>
<td>CV_FUNCTIONAL</td>
<td>1 Nonlinear system solution through functional iterations.</td>
</tr>
<tr>
<td>CV_NVENTOR</td>
<td>2 Nonlinear system solution through Newton iterations.</td>
</tr>
<tr>
<td>CV_SS</td>
<td>1 Scalar relative tolerance, scalar absolute tolerance.</td>
</tr>
<tr>
<td>CV_JR</td>
<td>2 Scalar relative tolerance, vector absolute tolerance.</td>
</tr>
<tr>
<td>CV_NORMAL</td>
<td>1 Solver returns at specified output time.</td>
</tr>
<tr>
<td>CV_NORMAL_TSTOP</td>
<td>3 Solver returns at specified output time, but does not proceed past the specified stopping time.</td>
</tr>
<tr>
<td>CV_ONE_STEP</td>
<td>2 Solver returns after each successful step.</td>
</tr>
<tr>
<td>CV_ONE_STEP_TSTOP</td>
<td>Solver returns after each successful step, but does not proceed past the specified stopping time.</td>
</tr>
</tbody>
</table>

### 10.2 CVODE output constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_SUCCESS</td>
<td>0 Successful function return.</td>
</tr>
<tr>
<td>CV_TSTOP_RETURN</td>
<td>1 CVode succeeded by reaching the specified stopping point.</td>
</tr>
<tr>
<td>CV_RADT_RETURN</td>
<td>2 CVode succeeded and found one or more roots.</td>
</tr>
</tbody>
</table>

### 10.2 CVODE output constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_SUCCESS</td>
<td>0 Successful function return.</td>
</tr>
<tr>
<td>CV_TSTOP_RETURN</td>
<td>1 CVode succeeded by reaching the specified stopping point.</td>
</tr>
<tr>
<td>CV_RADT_RETURN</td>
<td>2 CVode succeeded and found one or more roots.</td>
</tr>
</tbody>
</table>

### 124 CVODE Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_TOO_MUCH_WORK</td>
<td>-1 The solver took nstep internal steps but could not reach tout.</td>
</tr>
<tr>
<td>CV_TOO_MUCH_ACC</td>
<td>-2 The solver could not satisfy the accuracy demanded by the user for some internal step.</td>
</tr>
<tr>
<td>CV_ERR_FAILURE</td>
<td>-3 Error test failures occurred too many times during one internal time step or minimum step size was reached.</td>
</tr>
<tr>
<td>CV_CNV_FAIL</td>
<td>-4 Convergence test failures occurred too many times during one internal time step or minimum step size was reached.</td>
</tr>
<tr>
<td>CV_LIMIT_FAIL</td>
<td>-5 The linear solver’s initialization function failed.</td>
</tr>
<tr>
<td>CV_LSETUP_FAIL</td>
<td>-6 The linear solver’s setup function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CV_LSOLVE_FAIL</td>
<td>-7 The linear solver’s solve function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CV_LENSFUNC_FAIL</td>
<td>-8 The right-hand side function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CV_FIRST_MSFUNC_FAIL</td>
<td>-9 The right-hand side function failed at the first call.</td>
</tr>
<tr>
<td>CV_REP_MSFUNC_FAIL</td>
<td>-10 The right-hand side function had repeated recoverable errors.</td>
</tr>
<tr>
<td>CV_UNREC_MSFUNC_FAIL</td>
<td>-11 The right-hand side function had a recoverable error, but no recovery is possible.</td>
</tr>
<tr>
<td>CV_XFUNC_FAIL</td>
<td>-12 The rootfinding function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CV_MEM_FAIL</td>
<td>-20 A memory allocation failed.</td>
</tr>
<tr>
<td>CV_MEM_NULL</td>
<td>-21 The cvode_xen argument was NULL.</td>
</tr>
<tr>
<td>CV_JAC_NULL</td>
<td>-22 One of the function inputs is illegal.</td>
</tr>
<tr>
<td>CV_JAC_NULL</td>
<td>-23 The cvode memory block was not allocated by a call to cvodeMalloc.</td>
</tr>
<tr>
<td>CV_JAC_AD</td>
<td>-24 The derivative order is larger than the order used.</td>
</tr>
<tr>
<td>CV_JAC_T</td>
<td>-25 The time t is outside the last step taken.</td>
</tr>
<tr>
<td>CV_JAC_KY</td>
<td>-26 The output derivative vector is NULL.</td>
</tr>
</tbody>
</table>

### cvdense linear solver module

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVENSE_SUCCEED</td>
<td>0 Successful function return.</td>
</tr>
<tr>
<td>CVENSE_MEM_NULL</td>
<td>-1 The cvode_xen argument was NULL.</td>
</tr>
<tr>
<td>CVENSE_MEM_NULL</td>
<td>-2 The cvdense linear solver has not been initialized.</td>
</tr>
<tr>
<td>CVENSE_MEM_INPUT</td>
<td>-3 The cvdense solver is not compatible with the current nvector module.</td>
</tr>
<tr>
<td>CVENSE_MEM_FAIL</td>
<td>-4 A memory allocation request failed.</td>
</tr>
<tr>
<td>CVENSE_JACFUNC_UNRECVR</td>
<td>-5 The Jacobian function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CVENSE_JACFUNC_UNRECVR</td>
<td>-6 The Jacobian function had a recoverable error.</td>
</tr>
</tbody>
</table>

### cvband linear solver module

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVBAND_SUCCEED</td>
<td>0 Successful function return.</td>
</tr>
<tr>
<td>CVBAND_MEM_NULL</td>
<td>-1 The cvode_xen argument was NULL.</td>
</tr>
<tr>
<td>CVBAND_MEM_NULL</td>
<td>-2 The cvband linear solver has not been initialized.</td>
</tr>
<tr>
<td>CVBAND_MEM_INPUT</td>
<td>-3 The cvband solver is not compatible with the current nvector module, or an input value was illegal.</td>
</tr>
<tr>
<td>CVBAND_MEM_FAIL</td>
<td>-4 A memory allocation request failed.</td>
</tr>
<tr>
<td>CVBAND_JACFUNC_UNRECVR</td>
<td>-5 The Jacobian function failed in an unrecoverable manner.</td>
</tr>
</tbody>
</table>
10.2 CVODE output constants

CVBAND_JACFUNC_RECVR -6 The Jacobian function had a recoverable error.

CVBAND linear solver module

CVDIAG_SUCCESS 0 Successful function return.
CVDIAG_MEM_NULL -1 The cvode_mem argument was NULL.
CVDIAG_MEM_NULL -2 The CVDIAG linear solver has not been initialized.
CVDIAG_MEM_FAIL -4 A memory allocation request failed.

CVDIAG linear solver module

CVDIAG_SUCCESS 0 Successful function return.
CVDIAG_MEM_NULL -1 The cvode_mem argument was NULL.
CVDIAG_MEM_FAIL -4 A memory allocation request failed.

CVDIAG 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_ATIMES_FAIL_REC 3 The preconditioner solve function failed recoverably.
SPGMR_ATIMES_FAIL 4 The Jacobian-times-vector function failed recoverably.
SPGMR_PSET_FAIL_REC 5 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_PSET_FAIL_UNREC -3 The preconditioner setup function failed unrecoverably.

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_ATIMES_FAIL_REC 3 The preconditioner solve function failed recoverably.
SPGMR_ATIMES_FAIL 4 The Jacobian-times-vector function failed recoverably.
SPGMR_PSET_FAIL_REC 5 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_PSET_FAIL_UNREC -3 The preconditioner setup function failed unrecoverably.

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_ATIMES_FAIL_REC 3 The preconditioner solve function failed recoverably.
SPGMR_ATIMES_FAIL 4 The Jacobian-times-vector function failed recoverably.
SPGMR_PSET_FAIL_REC 5 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_PSET_FAIL_UNREC -3 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_JACFUNC_UNKNECVR -12 A user supplied function failed unrecoverably.

CVBANDPRE preconditioner module

CVBANDPRE_SUCCESS 0 Successful function return.
CVBANDPRE_PDATA_NULL -11 The preconditioner module has not been initialized.
CVBANDPRE_JACFUNC_UNKNECVR -12 A user supplied function failed unrecoverably.
Bibliography


memory requirements
  CvBand linear solver, 55
  CvBandPre conditioner, 72
  CvBandPre conditioner, 78
  CvDiag linear solver, 53
  CvDiag linear solver, 56
  Cvode solver, 48
  CvSpcDm linear solver, 57
MIN_STEP, 90
MUtilities_C, 45
MPI, 3

N_VCloseEmptyVectorArray, 100
N_VCloseVectorArray, 109
N_VCloseVectorArrayParallel, 106
N_VCloseVectorArraySerial, 184
N_VCloneVectorArrayEmpty Parallel, 106
N_VCloseVectorArrayEmptySerial, 104
N_VDestroyVectorArray, 100
N_VDestroyVectorArrayParallel, 107
N_VDestroyVectorArraySerial, 104
N_Vector, 26, 99
N MakeParallel, 106
N MakeSerial, 104
N NNewParallel, 106
N NNewSerial, 104
N NNewEmptyParallel, 106
N NNewEmptySerial, 104
N VPrint Parallel, 107
N VPrintSerial, 104
MCGVY_CDEF, 90
nonlinear system
deinition, 15
Newton convergence test, 17
Newton iteration, 16-17
N VCMPP, 106
N VCONTENT_T, 105
N VCONTENT, 103
N VDATA_P, 105
N VDATA_S, 103
N VGLOBALLENGTHP, 105
N VGLOBALLENGTHS, 103
N VLENGTH_P, 106
N VLENGTH_S, 104
N VGLOBALLENGTHP, 105
N VGLOBAL LENGTHS, 103
N VLOCALLENGTHP, 105
N VLOCALLENGTH_S, 105
N V2D_DATA_P, 105
N V2D_DATA_S, 103
N VVECTOR module, 90
N vvector Parallel, 26
N vvectorSerial, 26
optional input
  hand linear solver, 43
dense linear solver, 42-43
iterative linear solver, 43-46
solver, 40-42
optional output
  band linear solver, 55-56
  band-block-diagonal preconditioner, 78-79
  banded preconditioner, 72
dense linear solver, 53-55
diagonal linear solver, 56-57
interpolated solution, 46
iterative linear solver, 57-60
solver, 48-53
output mode, 18, 34
portability, 26
  Fortran, 82
  PRECISION, 33, 34, 44
  PRECISION, 33, 34, 44, 71, 76, 77
  PRECISION, 33, 33, 44
  PRECISION, 33, 44, 71, 76, 77
  preconditioning
    advice on, 23, 31
    banded, 69
    setup and solve phases, 23
    user-supplied, 43-44, 66
  pretype, 33, 34, 44, 71, 76, 77
  PVODE, 1
RCHEST, 26
realtype, 26
reinitialization, 60
right-hand side function, 61
RootFinding, 19, 28, 67, 92
ROOT, 90, 91
SMALL_REAL, 26
svdco generic linear solver
description of, 120
functions, 121
svdcon generic linear solver
description of, 119
functions, 120
support functions, 120
svdprep generic linear solver
description of, 121
functions, 121
STAR_LINK, 90
Stability limit detection, 18
step size bounds, 39-40
STOP_TIME, 90
SUNDIALS_CASE_LOWER, 82
SUNDIALS_CASE_UPPER, 82
sundials_vector, 26
sundials_types, 26
SUNDIALS_ORDERSOLVE, 82
SUNDIALS_ORDERSOLVE, 82
TFQMR method, 34, 45, 121
tolerances, 16, 30, 42, 62
UNIT_SUNOFF, 26
User main program
cvBandPre usage, 69
cvBandPre usage, 75
cvode usage, 27
fcvode usage, 95
fcvode usage, 93
fcvode usage, 83
vode, 1
vode, 1
weighted root-mean-square norm, 16
MPI: A Message-Passing Interface Standard

Message Passing Interface Forum

November 15, 2003
This work was supported in part by ARPA and NSF under grant ASC-9310330, the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, and by the Commission of the European Community through Esprit project P6643.

Version 1.1: June, 1995. Beginning in March, 1995, the Message Passing Interface Forum reconvened to correct errors and make clarifications in the MPI document of May 5, 1994, referred to below as Version 1.0. These discussions resulted in Version 1.1, which is this document. The changes from Version 1.0 arc minor. A version of this document with all changes marked is available. This paragraph is an example of a change.

Version 1.0: June, 1994. The Message Passing Interface Forum (MPIF), with participation from over 40 organizations, has been meeting since January 1993 to discuss and define a set of library interface standards for message passing. MPIF is not sanctioned or supported by any official standards organization.

The goal of the Message Passing Interface, simply stated, is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

This is the final report, Version 1.0, of the Message Passing Interface Forum. This document contains all the technical features proposed for the interface. This copy of the draft was processed by LATEX on November 15, 2003.

Please send comments on MPI to mpi-comments@cs.utk.edu. Your comment will be forwarded to MPIF committee members who will attempt to respond.

©1993, 1994, 1995 University of Tennessee, Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted, provided the University of Tennessee copyright notice and the title of this document appear, and notice is given that copying is by permission of the University of Tennessee.
Chapter 3
Point-to-Point Communication

3.1 Introduction

Sending and receiving of messages by processes is the basic MPI communication mechanism. The basic point-to-point communication operations are send and receive. Their use is illustrated in the example below.

```c
#include "mpi.h"
main(argc, argv)
{
    char **argv;
    int argc;
    char **argv;

    int myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Status status;

    char message[20];
    strcpy(message, "Hello, there");

    if (myrank == 0) /* code for process zero */
    {
        MPI_Send(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD);
        printf("sent :\n", message);
    }
    else /* code for process one */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
}
```

In this example, process zero (myrank = 0) sends a message to process one using the send operation MPI_SEND. The operation specifies a send buffer in the sender memory from which the message data is taken. In the example above, the send buffer consists of the storage containing the variable message in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send operation. The message sent will contain the 13 characters of this variable. In addition, the send operation associates an envelope with the message. This envelope specifies the message destination and contains distinguishing information that can be used by the receive operation to select a particular message. The last three parameters of the send operation specify the envelope for the message sent.

Process one (myrank = 1) receives this message with the receive operation MPI_RECV. The message to be received is selected according to the value of its envelope, and the message data is stored into the receive buffer. In the example above, the receive buffer consists of the storage containing the string message in the memory of process one. The first three parameters of the receive operation specify the location, size and type of the receive buffer. The next three parameters are used for selecting the incoming message. The last parameter is used to return information on the message just received.

The next sections describe the blocking send and receive operations. We discuss send, receive, blocking communication semantics, type matching requirements, type conversion in heterogeneous environments, and more general communication modes. Nonblocking communication is addressed next, followed by channel-like constructs and send-receive operations. We then consider general datatypes that allow one to transfer efficiently heterogeneous and noncontiguous data. We conclude with the description of calls for explicit packing and unpacking of messages.

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

```c
MPI_SEND(buf, count, datatype, dest, tag, comm)
```

- IN buf initial address of send buffer (choice)
- IN count number of elements in send buffer (nonnegative integer)
- IN datatype datatype of each send buffer element (handle)
- IN dest rank of destination (integer)
- IN tag message tag (integer)
- IN comm communicator (handle)

```c
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

3.2.2 Message data

The send buffer specified by the MPI_SEND operation consists of count successive entries of the type indicated by datatype, starting with the entry at address buf. Note that we specify
the message length in terms of number of elements, not number of bytes. The former is machine independent and closer to the application level.

The data part of the message consists of a sequence of count values, each of the type indicated by datatype. count may be zero, in which case the data part of the message is empty. The basic datatypes that can be specified for message data values correspond to the basic datatypes of the host language. Possible values of this argument for Fortran and the corresponding Fortran types are listed below.

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>Fortran datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>BYTE</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>PACKED</td>
</tr>
</tbody>
</table>

Possible values for this argument for C and the corresponding C types are listed below.

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

The datatypes MPI_BYTE and MPI_PACKED do not correspond to a Fortran or C datatype. A value of type MPI_BYTE consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. The use of the type MPI_PACKED is explained in Section 3.13.

MPI requires support of the datatypes listed above, which match the basic datatypes of Fortran 77 and ANSI C. Additional MPI datatypes should be provided if the host language

3.2. BLOCKING SEND AND RECEIVE OPERATIONS

has additional data types: MPI_LONG_LONG_INT, for (64 bit) C integers declared to be of type longlong int; MPI_DOUBLE_COMPLEX for double precision complex in Fortran declared to be of type DOUBLE COMPLEX; MPI_REAL2, MPI_REAL4 and MPI_REAL8 for Fortran reals, declared to be of type REAL*2, REAL*4 and REAL*8, respectively; MPI_INTEGER1, MPI_INTEGER2 and MPI_INTEGER4 for Fortran integers, declared to be of type INTEGER*1, INTEGER*2 and INTEGER*4, respectively; etc.

Rationale. One goal of the design is to allow for MPI to be implemented as a library, with no need for additional preprocessing or compilation. Thus, one cannot assume that a communication call has information on the datatype of variables in the communication buffer; this information must be supplied by an explicit argument. The need for such datatype information will become clear in Section 3.3.2. (End of rationale.)

3.2.3 Message envelope

In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which we collectively call the message envelope. These fields are

- source
- destination
- tag
- communicator

The message source is implicitly determined by the identity of the message sender. The other fields are specified by arguments in the send operation.

The message destination is specified by the dest argument.

The integer-valued message tag is specified by the tag argument. This integer can be used by the program to distinguish different types of messages. The range of valid tag values is 0,...,UB, where the value of UB is implementation dependent. It can be found by querying the value of the attribute MPI_TAG_MAX, as described in Chapter 7. MPI requires that UB be no less than 32767.

The comm argument specifies the communicator that is used for the send operation.

Communicators are explained in Chapter 5; below is a brief summary of their usage.

A communicator specifies the communication context for a communication operation. Each communication context provides a separate "communication universe:" messages are always received within the context they were sent, and messages sent in different contexts do not interfere.

The communicator also specifies the set of processes that share this communication context. This process group is ordered and processes are identified by their rank within this group. Thus, the range of valid values for dest is 0, ... , n-1, where n is the number of processes in the group. (If the communicator is an inter-communicator, then destinations are identified by their rank in the remote group. See Chapter 5.)

A predefined communicator MPI_COMM_WORLD is provided by MPI. It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in the group of MPI_COMM_WORLD.
3.2. BLOCKING SEND AND RECEIVE OPERATIONS

Advice to users. Users that are comfortable with the notion of a flat name space for processes, and a single communication context, as offered by most existing communication libraries, need only use the predefined variable MPI_COMM_WORLD as the comm argument. This will allow communication with all the processes available at initialization time.

Users may define new communicators, as explained in Chapter 5. Communicators provide an important encapsulation mechanism for libraries and modules. They allow modules to have their own disjoint communication universe and their own process numbering scheme. (End of advice to users.)

Advice to implementers. The message envelope would normally be encoded by a fixed-length message header. However, the actual encoding is implementation dependent. Some of the information (e.g., source or destination) may be implicit, and need not be explicitly carried by messages. Also, processes may be identified by relative ranks, or absolute ids, etc. (End of advice to implementers.)

3.2.4 Blocking receive

The syntax of the blocking receive operation is given below.

```c
MPI_RECV (buf, count, datatype, source, tag, comm, status)
```

Advice to users. The template function described in Section 3.8 can be used to receive messages of unknown length. (End of advice to users.)

Advice to implementers. Even though no specific behavior is mandated by MPI for erroneous programs, the recommended handling of overflow situations is to return in status information about the source and tag of the incoming message. The receive operation will return an error code. A quality implementation will also ensure that no memory that is outside the receive buffer will ever be overwritten.

In the case of a message shorter than the receive buffer, MPI is quite strict in that it allows no modification of the other locations. A more lenient statement would allow for some optimizations but this is not allowed. The implementation must be ready to end a copy into the receiver memory exactly at the end of the receive buffer, even if it is an odd address. (End of advice to implementors.)

The selection of a message by a receive operation is governed by the value of the message envelope. A message can be received by a receive operation if its envelope matches the source, tag and comm values specified by the receive operation. The receiver may specify a wildcard MPI_ANY_SOURCE value for source, and/or a wildcard MPI_ANY_TAG value for tag, indicating that any source and/or tag are acceptable. It cannot specify a wildcard value for comm. Thus, a message can be received by a receive operation only if it is addressed to the receiving process, has a matching communicator, has matching source unless source=MPI_ANY_SOURCE in the pattern, and has a matching tag unless tag=_MPI_ANY_TAG in the pattern.

The message tag is specified by the tag argument of the receive operation. The argument source, if different from MPI_ANY_SOURCE, is specified as a rank within the process group associated with that same communicator (remote process group, for intercommunicators). Thus, the range of valid values for the source argument is \( \{0, \ldots, n-1\} \cup \{MPI\_\text{ANY\_SOURCE}\} \), where \( n \) is the number of processes in this group.

Note the asymmetry between send and receive operations: A receive operation may accept messages from an arbitrary sender, on the other hand, a send operation must specify the process group associated with that same communicator (remote process group, for intercommunicators). This matches a “push” communication mechanism, where data transfer is effected by the sender (rather than a “pull” mechanism, where data transfer is effected by the receiver).

Source = destination is allowed, that is, a process can send a message to itself. (However, it is unsafe to do so with the blocking send and receive operations described above, since this may lead to deadlock. See Sec. 3.5.)

Advice to implementors. Message context and other communicator information can be implemented as an additional tag field. It differs from the regular message tag in that wild card matching is not allowed on this field, and that value setting for this field is controlled by communicator manipulation functions. (End of advice to implementors.)

3.2.5 Return status

The source or tag of a received message may not be known if wildcard values were used in the receive operation. Also, if multiple requests are completed by a single MPI function (see Section 3.7.5), a distinct error code may need to be returned for each request. The information is returned by the status argument of MPI_RECV. The type of status is MPI_defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.
CHAPTER 3. POINT-TO-POINT COMMUNICATION

In C, status is a structure that contains three fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR; the structure may contain additional fields. Thus, status(MPI_SOURCE, status(MPI_TAG) and status(MPI_ERROR) contain the source, tag, and error code, respectively, of the received message.

In Fortran, status is an array of INTEGER of size MPI_STATUS_SIZE. The constants MPI_SOURCE, MPI_TAG and MPI_ERROR are the indices of the entries that store the source, tag and error fields. Thus, status(MPI_SOURCE), status(MPI_TAG) and status(MPI_ERROR) contain, respectively, the source, tag and error code of the received message.

In general, message passing calls do not modify the value of the error code field of status variables. This field may be updated only by the functions in Section 3.7.5 which return multiple statuses. The field is updated if and only if such function returns with an error code of MPI_ERR_IN_STATUS.

Rationale. The error field in status is not needed for calls that return only one status, such as MPI_WAIT, since that would only duplicate the information returned by the function itself. The current design avoids the additional overhead of setting it in such cases. The field is needed for calls that return multiple statuses, since each request may have had a different failure. (End of rationale.)

The status argument also returns information on the length of the message received. However, this information is not directly available as a field of the status variable and a call to MPI_GET_COUNT is required to “decode” this information.

MPI_GET_COUNT(status, datatype, count)

IN status return status of receive operation (Status)
IN datatype datatype of each receive buffer entry (handle)
OUT count number of received entries (integer)

int MPI_get_count(MPI_status *status, MPI_DataType datatype, int *count)

INTEGER status(MPI_STATUS_SIZE), DATATYPE, COUNT, ERROR

Returns the number of entries received. (Again, we count entries, each of type datatype, not bytes.) The datatype argument should match the argument provided by the receive call that set the status variable. (We shall later see, in Section 3.12.6, that MPI_GET_COUNT may return, in certain situations, the value MPI_UNDEFINED.)

Rationale. Some message passing libraries use INOUT count, tag and source arguments, thus using them both to specify the selection criteria for incoming messages and return the actual envelope values of the received message. The use of a separate status argument prevents errors that are often attached with INOUT argument (e.g., using the MPI_ANY_TAG constant as the tag in a receive). Some libraries use calls that refer implicitly to the “last message received.” This is not thread safe.

The datatype argument is passed to MPI_GET_COUNT so as to improve performance. A message might be received without counting the number of elements it contains, and the count value is often not needed. Also, this allows the same function to be used after a call to MPI_PROBE. (End of rationale.)

3.3 DATA TYPE MATCHING AND DATA CONVERSION

All send and receive operations use the buf, count, datatype, source, dest, tag, comm and status arguments in the same way as the blocking MPI_SEND and MPI_RECV operations described in this section.

3.3.1 Type matching rules

One can think of message transfer as consisting of the following three phases.

1. Data is pulled out of the send buffer and a message is assembled.
2. A message is transferred from sender to receiver.
3. Data is pulled from the incoming message and disassembled into the receive buffer.

Type matching has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type specified for that entry by the send operation; the type specified by the send operation has to match the type specified by the receive operation; and the type of each variable in the receive buffer has to match the type specified for that entry by the receive operation. A program that fails to observe these three rules is erroneous.

To define type matching more precisely, we need to deal with two issues: matching of types of the host language with types specified in communication operations; and matching of types at sender and receiver.

The types of a send and receive match (phase two) if both operations use identical names. That is, MPI_INTEGER matches MPI_INTEGER, MPI_REAL matches MPI_REAL, and so on. There is one exception to this rule, discussed in Sec. 3.13, the type MPI_PACKED can match any other type.

The type of a variable in a host program matches the type specified in the communication operation if the datatype name used by that operation corresponds to the basic type of the host program variable. For example, an entry with type name MPI_INTEGER matches a Fortran variable of type INTEGER. A table giving this correspondence for Fortran and C appears in Sec. 3.2.2. There are two exceptions to this last rule: an entry with type name MPI_BYTE or MPI_PACKED can be used to match any byte of storage (on a byte-addressable machine), irrespective of the datatype of the variable that contains this byte. The type MPI_PACKED is used to send data that has been explicitly packed, or receive data that will be explicitly unpacked, see Section 3.13. The type MPI_BYTE allows one to transfer the binary value of a byte in memory unchanged.

To summarize, the type matching rules fall into the three categories below.

• Communication of typed values (e.g., with datatype different from MPI_BYTE), where the datatypes of the corresponding entries in the sender program, in the send call, in the receive call and in the receiver program must all match.

• Communication of untyped values (e.g., of datatype MPI_BYTE), where both sender and receiver use the datatype MPI_BYTE. In this case, there are no requirements on the types of the corresponding entries in the sender and the receiver programs, nor is it required that they be the same.

• Communication involving packed data, where MPI_PACKED is used.
The following examples illustrate the first two cases.

Example 3.1 Sender and receiver specify matching types.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
  CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE
  CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

This code is correct if both a and b are real arrays of size $\geq 10$. (In Fortran, it might be correct to use this code even if $a$ or $b$ have size $< 10$ e.g., when $a(1)$ can be equivalenced to an array with ten reals.)

Example 3.2 Sender and receiver do not specify matching types.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
  CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE
  CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

This code is erroneous, since sender and receiver do not provide matching datatype arguments.

Example 3.3 Sender and receiver specify communication of untyped values.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
  CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
ELSE
  CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

This code is correct, irrespective of the type and size of $a$ and $b$ (unless this results in an out of bound memory access).

Advice to users. If a buffer of type MPI_BYTE is passed as an argument to MPI_SEND, then MPI will send the data stored at contiguous locations, starting from the address indicated by the buf argument. This may have unexpected results when the data layout is not as a casual user would expect it to be. For example, some Fortran compilers implement variables of type CHARACTER as a structure that contains the character length and a pointer to the actual string. In such an environment, sending and receiving a Fortran CHARACTER variable using the MPI_BYTE type will not have the anticipated result of transferring the character string. For this reason, the user is advised to use typed communications whenever possible. (End of advice to users.)

3.3 DATA TYPE MATCHING AND DATA CONVERSION

Type MPI_CHARACTER

The type MPI_CHARACTER matches one character of a Fortran variable of type CHARACTER, rather than the entire character string stored in the variable. Fortran variables of type CHARACTER or substrings are transferred as if they were arrays of characters. This is illustrated in the example below.

Example 3.4 Transfer of Fortran CHARACTERS.

```
CHARACTER*10 a
CHARACTER*10 b
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
  CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE
  CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

The last five characters of string $b$ at process 1 are replaced by the first five characters of string $a$ at process 0.

Rationale. The alternative choice would be for MPI_CHARACTER to match a character of arbitrary length. This runs into problems. A Fortran character variable is a constant length string, with no special termination symbol. There is no fixed convention on how to represent characters, and how to store their length. Some compilers pass a character argument to a routine as a pair of arguments, one holding the address of the string and the other holding the length of string. Consider the case of an MPI communication call that is passed a communication buffer with type defined by a derived datatype (Section 3.12). If this communicator buffer contains variables of type CHARACTER then the information on their length will not be passed to the MPI routine.

This problem forces us to provide explicit information on character length with the MPI call. One could add a length parameter to the type MPI_CHARACTER, but this does not add much convenience and the same functionality can be achieved by defining a suitable derived datatype. (End of rationale.)

Advice to implementors. Some compilers pass Fortran CHARACTER arguments as a structure with a length and a pointer to the actual string. In such an environment, the MPI call needs to dereference the pointer in order to reach the string. (End of advice to implementors.)

3.3.2 Data conversion

One of the goals of MPI is to support parallel computations across heterogeneous environments. Communication in a heterogeneous environment may require data conversions. We use the following terminology.

type conversion changes the datatype of a value, e.g., by rounding a REAL to an INTEGER.
representation conversion changes the binary representation of a value, e.g., from Hex floating point to IEEE floating point.

    The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical or character values, and to convert a floating point value to the nearest value that can be represented on the target system.

    Overflow and underflow exceptions may occur during floating point conversions. Conversion of integers or characters may also lead to exceptions when a value that can be represented in one system cannot be represented in the other system. An exception occurring during representation conversion results in a failure of the communication. An error occurs either in the send operation, or the receive operation, or both.

    If a value sent in a message is untyped (i.e., of type MPI_BYTE), then the binary representation of the byte stored at the receiver is identical to the binary representation of the byte loaded at the sender. This holds true, whether sender and receiver run in the same or in distinct environments. No representation conversion is required. (Note that representation conversion may occur when values of type MPI_CHARACTER or MPI_CHAR are transferred, for example, from an EBCDIC encoding to an ASCII encoding.)

    No conversion need occur when an MPI program executes in a homogeneous system, where all processes run in the same environment.

    Consider the three examples, 3.1–3.3. The first program is correct, assuming that a and b are REAL arrays of size $\geq 10$. If the sender and receiver execute in different environments, then the ten real values that are fetched from the send buffer will be converted to the representation for reals on the receiver site before they are stored in the receive buffer. While the number of real elements fetched from the send buffer equal the number of real elements stored in the receive buffer, the number of bytes stored need not equal the number of bytes loaded. For example, the sender may use a four byte representation and the receiver an eight byte representation for reals.

    The second program is erroneous, and its behavior is undefined.

    The third program is correct. The exact same sequence of forty bytes that were loaded from the send buffer will be stored in the receive buffer, even if sender and receiver are in a different environment. The message sent has exactly the same length (in bytes) and the same binary representation as the message received. If a and b are of different types, or if they are of the same type but different data representations are used, then the bits stored in the receive buffer may encode values that are different from the values they encoded in the send buffer.

    Data representation conversion also applies to the envelope of a message: source, destination and tag are all integers that may need to be converted.

    Advice to implementors. The current definition does not require messages to carry data type information. Both sender and receiver provide complete data type information. In a heterogeneous environment, one can either use a machine independent encoding such as XDR, or have the receiver convert from the sender representation to its own, or even have the sender do the conversion.

    Additional type information might be added to messages in order to allow the system to detect mismatches between datatype at sender and receiver. This might be particularly useful in a slower but safer debug mode. (End of advice to implementors.)
A buffered mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. However, unlike the standard send, this operation is local, and its completion does not depend on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete. An error will occur if there is insufficient buffer space. The amount of available buffer space is controlled by the user — see Section 3.6. Buffer allocation by the user may be required for the buffered mode to be effective.

A send that uses the synchronous mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but also indicates that the receiver has reached a certain point in its execution, namely that it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is non-local.

A send that uses the ready communication mode may be started only if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined. On some systems, this allows the removal of a hand-shake operation that is otherwise required and results in improved performance. The completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.

Three additional send functions are provided for the three additional communication modes. The communication mode is indicated by a one letter prefix: B for buffered, S for synchronous, and R for ready.

MPI_BSSEND (buf, count, datatype, dest, tag, comm)

IN buf initial address of send buffer (choice)
IN count number of elements in send buffer (integer)
IN datatype datatype of each send buffer element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)

Send in buffered mode:

MPI_SSSEND (buf, count, datatype, dest, tag, comm)

IN buf initial address of send buffer (choice)
IN count number of elements in send buffer (integer)
IN datatype datatype of each send buffer element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)

Send in synchronous mode:

MPI_RSSEND (buf, count, datatype, dest, tag, comm)

IN buf initial address of send buffer (choice)
IN count number of elements in send buffer (integer)
IN datatype datatype of each send buffer element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)

Send in ready mode:

There is only one receive operation, which can match any of the send modes. The receive operation described in the last section is blocking: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started).

In a multi-threaded implementation of MPI, the system may de-schedule a thread that is blocked on a send or receive operation, and schedule another thread for execution in the same address space. In such a case it is the user’s responsibility not to access or modify a
communication buffer until the communication completes. Otherwise, the outcome of the computation is undefined.

Rationale. We prohibit read accesses to a send buffer while it is being used, even though the send operation is not supposed to alter the content of this buffer. This may seem more stringent than necessary, but the additional restriction causes little loss of functionality and allows better performance on some systems—consider the case where data transfer is done by a DMA engine that is not cache-coherent with the main processor. \( \text{(End of rationale.)} \)

Advice to implementors. Since a synchronous send cannot complete before a matching receive is posted, one will not normally buffer messages sent by such an operation.

It is recommended to choose buffering over blocking the sender, whenever possible, for standard sends. The programmer can signal his or her preference for blocking the sender until a matching receive occurs by using the synchronous send mode.

A possible communication protocol for the various communication modes is outlined below.

\textbf{ready send}: The message is sent as soon as possible.

\textbf{synchronous send}: The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message.

\textbf{standard send}: First protocol may be used for short messages, and second protocol for long messages.

\textbf{buffered send}: The sender copies the message into a buffer and then sends it with a nonblocking send (using the same protocol as for standard send).

Additional control messages might be needed for flow control and error recovery. Of course, there are many other possible protocols.

Ready send can be implemented as a standard send. In this case there will be no performance advantage (or disadvantage) for the use of ready send.

A standard send can be implemented as a synchronous send. In such a case, no data buffering is needed. However, many (most?) users expect some buffering.

In a multi-threaded environment, the execution of a blocking communication should block only the executing thread, allowing the thread scheduler to de-schedule this thread and schedule another thread for execution. \( \text{(End of advice to implementors.)} \)

### 3.5 Semantics of point-to-point communication

A valid MPI implementation guarantees certain general properties of point-to-point communication, which are described in this section.

\textbf{Order} Messages are non-overtaking. If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives. It guarantees that message-passing code is deterministic, if processes are single-threaded and the wildcard MPI\_ANY\_SOURCE is not used in receives. (Some of the calls described later, such as MPI\_CANCEL or MPI\_WAITANY, are additional sources of nondeterminism.)

If a process has a single thread of execution, then any two communications executed by this process are ordered. On the other hand, if the process is multi-threaded, then the semantics of thread execution may not define a relative order between two send operations executed by two distinct threads. The operations are logically concurrent, even if one physically precedes the other. In such a case, the two messages sent can be received in any order. Similarly, if two receive operations that are logically concurrent receive two successively sent messages, then the two messages can match the two receives in either order.

\begin{example}
\textbf{Example 3.5} An example of non-overtaking messages.
\begin{verbatim}
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
ELSE ! rank.EQ.1
  CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
END IF
The message sent by the first send must be received by the first receive, and the message sent by the second send must be received by the second receive.
\end{verbatim}
\end{example}

\textbf{Progress} If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system. The send operation will complete, unless the receive is satisfied by another message, and completes; the receive operation will complete, unless the message sent is consumed by another matching receive that was posted at the same destination process.

\begin{example}
\textbf{Example 3.6} An example of two, intertwined matching pairs.
\begin{verbatim}
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE ! rank.EQ.1
  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
Both processes invoke their first communication call. Since the first send of process zero uses the buffered mode, it must complete, irrespective of the state of process one. Since no matching receive is posted, the message will be copied into buffer space. (If insufficient buffer space is available, then the program will fail.) The second send is then invoked. At
\end{verbatim}
\end{example}
that point, a matching pair of send and receive operation is enabled, and both operations must complete. Process one next invokes its second receive call, which will be satisfied by the buffered message. Note that process one received the messages in the reverse order they were sent.

Example 3.7 An exchange of messages.

```fortran
CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

This program will succeed even if no buffer space for data is available. The standard send operation can be replaced, in this example, with a synchronous send.

Example 3.8 An attempt to exchange messages.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
  ELSE ! rank.EQ.1
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

The receive operation of the first process must complete before its send, and can complete only if the matching send of the second processor is executed. The receive operation of the second process must complete before its send and can complete only if the matching send of the first process is executed. This program will always deadlock. The same holds for any other send mode.

Example 3.9 An exchange that relies on buffering.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
  ELSE ! rank.EQ.1
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

The message sent by each process has to be copied out before the send operation returns and the receive operation starts. For the program to complete, it is necessary that at least one of the two messages sent be buffered. Thus, this program can succeed only if the communication system can buffer at least count words of data.

Advice to users. When standard send operations are used, then a deadlock situation may occur where both processes are blocked because buffer space is not available. The same will certainly happen, if the synchronous mode is used. If the buffered mode is used, and not enough buffer space is available, then the program will not complete either. However, rather than a deadlock situation, we shall have a buffer overflow error.

A program is “safe” if no message buffering is required for the program to complete. One can replace all sends in such programs with synchronous sends, and the program will still run correctly. This conservative programming style provides the best

Resource limitations Any pending communication operation consumes system resources that are limited. Errors may occur when lack of resources prevent the execution of an MPI call. A quality implementation will use a (small) fixed amount of resources for each pending send in the ready or synchronous mode and for each pending receive. However, buffer space may be consumed to store messages sent in standard mode, and must be consumed to store messages sent in buffered mode, when no matching receive is available. The amount of space available for buffering will be much smaller than program data memory on many systems. Then, it will be easy to write programs that overrun available buffer space.

MPI allows the user to provide buffer memory for messages sent in the buffered mode. Furthermore, MPI specifies a detailed operational model for the use of this buffer. An MPI implementation is required to do no worse than implied by this model. This allows users to avoid buffer overflows when they use buffered sends. Buffer allocation and use is described in Section 3.6.

A buffered send operation that cannot complete because of a lack of buffer space is erroneous. When such a situation is detected, an error is signalled that may cause the program to terminate abnormally. On the other hand, a standard send operation that will result. Additional synchronization has to be added to the program so as to prevent this from occurring. If standard sends are used, then the program will be automatically throttled, as its send operations will block when buffer space is unavailable.

In some situations, a lack of buffer space leads to deadlock situations. This is illustrated by the examples below.

Example 3.8 An attempt to exchange messages.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
  ELSE ! rank.EQ.1
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

This program will succeed even if no buffer space for data is available. The standard send operation can be replaced, in this example, with a synchronous send.

Example 3.9 An exchange that relies on buffering.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
  ELSE ! rank.EQ.1
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

The receive operation of the first process must complete before its send, and can complete only if the matching send of the second processor is executed. The receive operation of the second process must complete before its send and can complete only if the matching send of the first process is executed. This program will always deadlock. The same holds for any other send mode.
portability, since program completion does not depend on the amount of buffer space available or in the communication protocol used.

Many programmers prefer to have more leeway and be able to use the "unsafe" programming style shown in example 3.9. In such cases, the use of standard sends is likely to provide the best compromise between performance and robustness: quality implementations will provide sufficient buffering so that "common practice" programs will not deadlock. The buffered send mode can be used for programs that require more buffering, or in situations where the programmer wants more control. This mode might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

Nonblocking message-passing operations, as described in Section 3.7, can be used to avoid the need for buffering outgoing messages. This prevents deadlocks due to lack of buffer space, and improves performance, by allowing overlap of computation and communication, and avoiding the overheads of allocating buffers and copying messages into buffers. (End of advice to users.)

### 3.6 Buffer allocation and usage

A user may specify a buffer to be used for buffering messages sent in buffered mode. Buffering is done by the sender.

```c
MPI_Buffer_attach( buffer, size)
```

| IN buffer | initial buffer address (choice) |
| IN size   | buffer size, in bytes (integer) |

```c
int MPI_Buffer_attach( void* buffer, int size)
```

**MPI_BUFFER_ATTACH** (BUFFER, SIZE, IERROR)

```c
<type> BUFFER(*), INTEGER SIZE, IERROR
```

Provides to MPI a buffer in the user’s memory to be used for buffering outgoing messages. The buffer is used only by messages sent in buffered mode. Only one buffer can be attached to a process at a time.

```c
MPI_Buffer_detach( buffer_addr, size)
```

| OUT buffer_addr | initial buffer address (choice) |
| OUT size       | buffer size, in bytes (integer) |

```c
int MPI_Buffer_detach( void* buffer_addr, int* size)
```

**MPI_BUFFER_detach** (BUFFER_ADDR, SIZE, IERROR)

```c
<type> BUFFER_ADDR(*), INTEGER SIZE, IERROR
```

Detach the buffer currently associated with MPI. The call returns the address and the size of the detached buffer. This operation will block until all messages currently in the buffer have been transmitted. Upon return of this function, the user may re-use or deallocate the space taken by the buffer.

**Example 3.10** Calls to attach and detach buffers.

```c
#define BUFFSIZE 10000

int size
char *buff;

MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach(&buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach( buff, size);
/* Buffer of 10000 bytes available again */

Advice to users. Even though the C functions MPI_Buffer_attach and MPI_Buffer_detach both have a first argument of type void*, these arguments are used differently: A pointer to the buffer is passed to MPI_Buffer_attach; the address of the pointer is passed to MPI_Buffer_detach, so that this call can return the pointer value. (End of advice to users.)

**Rationale.** Both arguments are defined to be of type void* (rather than void* and void**, respectively), so as to avoid complex type casts. E.g., in the last example, `buff`, which is of type char**, can be passed as argument to MPI_Buffer_detach without type casting. If the formal parameter had type void** then we would need a type cast before and after the call. (End of rationale.)

The statements made in this section describe the behavior of MPI for buffered-mode sends. When no buffer is currently associated, MPI behaves as if a zero-sized buffer is associated with the process.

MPI must provide as much buffering for outgoing messages as if outgoing message data were buffered by the sending process, in the specified buffer space, using a circular, contiguous-space allocation policy. We outline below a model implementation that defines this policy. MPI may provide more buffering, and may use a better buffer allocation algorithm than described below. On the other hand, MPI may signal an error whenever the simple buffering allocator described below would run out of space. In particular, if no buffer is explicitly associated with the process, then any buffered send may cause an error.

**Rationale.** MPI does not provide mechanisms for querying or controlling buffering done by standard mode sends. It is expected that vendors will provide such information for their implementations.

There is a wide spectrum of possible implementations of buffered communication: buffering can be done at sender, at receiver, or both; buffers can be dedicated to one sender-receiver pair, or be shared by all communications; buffering can be done in real or in virtual memory; it can use dedicated memory, or memory shared by other processes; buffer space may be allocated statically or be changed dynamically, etc. It does not seem feasible to provide a portable mechanism for querying or controlling buffering that would be compatible with all these choices, yet provide meaningful information. (End of rationale.)
3.6.1 Model implementation of buffered mode

The model implementation uses the packing and unpacking functions described in Section 3.13 and the nonblocking communication functions described in Section 3.7.

We assume that a circular queue of pending message entries (PME) is maintained. Each entry contains a communication request handle that identifies a pending nonblocking send, a pointer to the next entry and the packed message data. The entries are stored in successive locations in the buffer. Free space is available between the queue tail and the queue head.

A buffer send call results in the execution of the following code.

- Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request, update queue head to point to that entry.
- Compute the number, \( n \), of bytes needed to store an entry for the new message. An upper bound on \( n \) can be computed as follows: A call to the function `MPI_PACK_SIZE(count, datatype, comm, size)`, with the `count`, `datatype` and `comm` arguments used in the `MPI_BSEND` call, returns an upper bound on the amount of space needed to buffer the message data (see Section 3.13). The MPI constant `MPI_BSEND_OVERHEAD` provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).
- Find the next contiguous empty space of \( n \) bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space is not found then raise buffer overflow error.
- Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data. `MPI_PACK` is used to pack data.
- Post nonblocking send (standard mode) for packed data.
- Return

3.7 Nonblocking communication

One can improve performance on many systems by overlapping communication and computation. This is especially true on systems where communication can be executed autonomously by an intelligent communication controller. Light-weight threads are one mechanism for achieving such overlap. An alternative mechanism that often leads to better performance is to use nonblocking communication. A nonblocking `send` call initiates the send operation, but does not complete it. The `send` call will return before the message was copied out of the send buffer. A separate `send complete` call is needed to complete the communication. This allows for efficient memory-to-memory copying, as information is provided early on the location of the receive buffer.

Nonblocking sends can be matched with blocking receives, and vice-versa. Nonblocking send start calls can use the same four modes as blocking sends: `standard`, `buffered`, `synchronous` and `ready`. These carry the same meaning. Sends of all modes, `ready` excepted, can be started whether a matching receive has been posted or not; a nonblocking `ready` send can be started only if a matching receive is posted. In all cases, the `send` start call is local: it returns immediately, irrespective of the status of other processes. If the call causes some system resource to be exhausted, then it will fail and return an error code. Quality implementations of MPI should ensure that this happens only in “pathological” cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

The `send complete` call returns when data has been copied out of the send buffer. It may carry additional meaning, depending on the send mode.

If the send mode is `synchronous`, then the send can complete only if a matching receive has started. That is, a receive has been posted, and has been matched with the send. In this case, the `send complete` call is non-local. Note that a synchronous, nonblocking send may complete, if matched by a nonblocking receive, before the receive complete call occurs. (It can complete as soon as the `send` “knows” the `send` will complete, but before the receiver “knows” the `send` will complete.)

If the send mode is `buffered` then the message must be buffered if there is no pending receive. In this case, the `send complete` call is local, and must succeed irrespective of the status of a matching receive.

If the send mode is `standard` then the `send complete` call may return before a matching receive occurred, if the message is buffered. On the other hand, the `send complete` may not complete until a matching receive occurred, and the message was copied into the receive buffer.

Nonblocking sends can be matched with blocking receives, and vice-versa.

Advice to users. The completion of a `send` operation may be delayed, for `standard` mode, and must be delayed, for `synchronous` mode, until a matching receive is posted. The use of nonblocking sends in these two cases allows the sender to proceed ahead of the receiver, so that the computation is more tolerant of fluctuations in the speeds of the two processes.

Nonblocking sends in the `buffered` and `ready` modes have a more limited impact. A nonblocking send will return as soon as possible, whereas a blocking send will return after the data has been copied out of the sender memory. The use of nonblocking sends is advantageous in these cases only if data copying can be concurrent with computation.

The message-passing model implies that communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication (data can be moved directly to the receive buffer, and there is no need to queue a pending send request). However, a receive operation can complete only after the matching send has occurred. The use of nonblocking receives allows one to achieve lower communication overheads without blocking the receiver while it waits for the send. (End of advice to users.)
3.7.1 Communication Objects

Nonblocking communications use opaque request objects to identify communication operations and match the operation that initiates the communication with the operation that terminates it. These are system objects that are accessed via a handle. A request object identifies various properties of a communication operation, such as the send mode, the communication buffer that is associated with it, its context, the tag and destination arguments to be used for a send, or the tag and source arguments to be used for a receive. In addition, this object stores information about the status of the pending communication operation.

3.7.2 Communication initiation

We use the same naming conventions as for blocking communication: a prefix of B, S, or R is used for buffered, synchronous or ready mode. In addition a prefix of I (for immediate) indicates that the call is nonblocking.

\[\text{MPI\_ISEND(bu\text{f}, count, datatype, dest, tag, comm, request)}\]

\[\begin{align*}
\text{IN} & \quad \text{buf} & \text{initial address of send buffer (choice)} \\
\text{IN} & \quad \text{count} & \text{number of elements in send buffer (integer)} \\
\text{IN} & \quad \text{datatype} & \text{datatype of each send buffer element (handle)} \\
\text{IN} & \quad \text{dest} & \text{rank of destination (integer)} \\
\text{IN} & \quad \text{tag} & \text{message tag (integer)} \\
\text{IN} & \quad \text{comm} & \text{communicator (handle)} \\
\text{OUT} & \quad \text{request} & \text{communication request (handle)}
\end{align*}\]

\[\text{int MPI\_Ibsend(void\text{*} buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm, MPI\_Request *request)}\]

\[\begin{align*}
\text{INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR}
\end{align*}\]

Start a buffered mode, nonblocking send.

\[\text{MPI\_ISSEND(bu\text{f}, count, datatype, dest, tag, comm, request)}\]

\[\begin{align*}
\text{IN} & \quad \text{buf} & \text{initial address of send buffer (choice)} \\
\text{IN} & \quad \text{count} & \text{number of elements in send buffer (integer)} \\
\text{IN} & \quad \text{datatype} & \text{datatype of each send buffer element (handle)} \\
\text{IN} & \quad \text{dest} & \text{rank of destination (integer)} \\
\text{IN} & \quad \text{tag} & \text{message tag (integer)} \\
\text{IN} & \quad \text{comm} & \text{communicator (handle)} \\
\text{OUT} & \quad \text{request} & \text{communication request (handle)}
\end{align*}\]

\[\text{int MPI\_Issend(void\text{*} buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm, MPI\_Request *request)}\]

\[\begin{align*}
\text{INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR}
\end{align*}\]

Start a synchronous mode, nonblocking send.
3.7.3 Communication Completion

The functions MPI_WAIT and MPI_TEST are used to complete a nonblocking communication. The completion of a send operation indicates that the sender is now free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the message has been received, rather, it may have been buffered by the communication subsystem. However, if a synchronous mode send was used, the completion of the send operation indicates that a matching receive was initiated, and that the message will eventually be received by this matching receive.

The completion of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it, and that the status object is set. It does not indicate that the matching send operation has completed (but indicates, of course, that the send was initiated).

We shall use the following terminology: A null handle is a handle with value MPI_REQUEST_NULL. A persistent request and the handle to it are inactive if the request is not associated with any ongoing communication (see Section 3.9). A handle is active if it is neither null nor inactive. An empty status is a status which is set to return tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE, and is also internally configured so that calls to MPI GET_COUNT and MPI GET_ELEMENTS return count = 0. We set a status variable to empty when the value returned by it is not significant. Status is set in this way so as to prevent errors due to accesses of stale information.

A call to MPI_WAIT returns when the operation identified by request is complete. If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to MPI_WAIT and the request handle is set to MPI REQUEST NULL. MPIWAIT is a non-local operation.

The call returns, in status, information on the completed operation. The content of the status object for a receive operation can be accessed as described in Section 3.2.5. The status object for a send operation may be queried by a call to MPI TEST CANCELLED (see Section 3.8).

One is allowed to call MPI_WAIT with a null or inactive request argument. In this case the operation returns immediately with empty status.

Advice to users. Successful return of MPI_WAIT after a MPI_IRESEND implies that the user send buffer can be reused — i.e., data has been sent out or copied into a buffer attached with MPI BUFFER ATTACH. Note that, at this point, we can no longer cancel the send (see Sec. 3.8). If a matching receive is never posted, then the buffer cannot be freed. This runs somewhat counter to the stated goal of MPI CANCEL.
**3.7. NONBLOCKING COMMUNICATION**

A request object can be deallocated without waiting for the associated communication to complete, by using the following operation.

```fortran
INTEGER REQUEST, IERROR

CALL MPI_REQUEST_FREE(REQUEST, IERROR)
```

Mark the request object for deallocation and set request to MPI_REQUEST_NULL. An ongoing communication that is associated with the request will be allowed to complete. The request will be deallocated only after its completion.

**Rationale.** The MPI_REQUEST_FREE mechanism is provided for reasons of performance and convenience on the sending side. (End of rationale.)

**Advice to users.** Once a request is freed by a call to MPI_REQUEST_FREE, it is not possible to check for the successful completion of the associated communication with calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the communication, an error code cannot be returned to the user — such an error must be treated as fatal. Questions arise as to how one knows when the operations have completed when using MPI_REQUEST_FREE. Depending on the program logic, there may be other ways in which the program knows that certain operations have completed and this makes usage of MPI_REQUEST_FREE practical. For example, an active send request could be freed when the logic of the program is such that the receiver sends a reply to the message sent — the arrival of the reply informs the sender that the send has completed and the send buffer can be reused. An active receive request should never be freed as the receiver will have no way to verify that the receive has completed and the receive buffer can be reused. (End of advice to users.)

**Example 3.12** An example using MPI_REQUEST_FREE.

```fortran
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank)
IF(rank.EQ.0) THEN
   DO i=1, n
      CALL MPI_ISEND(outval, 1, MPI_REAL, i, 0, req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
   END DO
END IF
```

**Example 3.11** Simple usage of nonblocking operations and MPI_WAIT.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
   CALL MPI_ISEND(a(1), 10, MPI_REAL, i, tag, comm, request, ierr)
END IF
```

Advice to implementors. In a multi-threaded environment, a call to MPI_WAIT should block only the calling thread, allowing the thread scheduler to schedule another thread for execution. (End of advice to implementors.)

The request will be deallocated only after its completion. An ongoing communication that is associated with the request will be allowed to complete. (End of rationale.)

**Advice to users.** The use of the nonblocking MPI_TEST call allows the user to schedule alternative activities within a single thread of execution. An event-driven thread scheduler can be emulated with periodic calls to MPI_TEST. (End of advice to users.)

**Rationale.** The function MPI_TEST returns flag = true exactly in those situations where the function MPI_WAIT returns; both functions return in such case the same value in status. Thus, a blocking Wait can be easily replaced by a nonblocking Test. (End of rationale.)

**Example 3.12** An example using MPI_REQUEST_FREE.

```fortran
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank)
IF(rank.EQ.0) THEN
   DO i=1, n
      CALL MPI_ISEND(outval, 1, MPI_REAL, i, 0, req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
   END DO
END IF
```

Advice to implementors. In a multi-threaded environment, a call to MPI_WAIT should block only the calling thread, allowing the thread scheduler to schedule another thread for execution. (End of advice to implementors.)

The request will be deallocated only after its completion. An ongoing communication that is associated with the request will be allowed to complete. (End of rationale.)

**Advice to users.** The use of the nonblocking MPI_TEST call allows the user to schedule alternative activities within a single thread of execution. An event-driven thread scheduler can be emulated with periodic calls to MPI_TEST. (End of advice to users.)

**Rationale.** The function MPI_TEST returns flag = true exactly in those situations where the function MPI_WAIT returns; both functions return in such case the same value in status. Thus, a blocking Wait can be easily replaced by a nonblocking Test. (End of rationale.)

**Example 3.12** An example using MPI_REQUEST_FREE.

```fortran
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank)
IF(rank.EQ.0) THEN
   DO i=1, n
      CALL MPI_ISEND(outval, 1, MPI_REAL, i, 0, req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
   END DO
END IF
```
The first send of process zero will match the first receive of process one, even if both messages
are sent before process one executes either receive.

Progress A call to MPI_WAIT that completes a receive will eventually terminate and return if a matching send has been started, unless the send is satisfied by another receive. In particular, if the matching send is nonblocking, then the receive should complete even if no call is executed by the sender to complete the send. Similarly, a call to MPI_WAIT that completes a send will eventually return if a matching receive has been started, unless the receive is satisfied by another send, and even if no call is executed to complete the receive.

Example 3.14 An illustration of progress semantics.

Example 3.13 Message ordering for nonblocking operations.

CALL MPI_Comm_Rank(comm, rank, ierr)
IF (rank.EQ.0) THEN
CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
CALL MPI_SSEND(b, 1, MPI_REAL, 1, 0, comm, ierr)
ELSE
CALL MPI_RECV(a, 1, MPI_REAL, 0, 0, comm, ierr)
CALL MPI_RECV(b, 1, MPI_REAL, 0, 0, comm, ierr)
END IF

This code should not deadlock in a correct MPI implementation. The first synchronous
send of process zero must complete after process one posts the matching (nonblocking)
receive even if process one has not yet reached the completing wait call. Thus, process zero
will continue and execute the second send, allowing process one to complete execution.

If an MPI_TEST that completes a receive is repeatedly called with the same arguments,
and a matching send has been started, then the call will eventually return flag = true, unless
the send is satisfied by another receive. If an MPI_TEST that completes a send is repeatedly
called with the same arguments, and a matching receive has been started, then the call will
eventually return flag = true, unless the receive is satisfied by another send.

3.7.5 Multiple Completions

It is convenient to be able to wait for the completion of any, some, or all the operations
in a list, rather than having to wait for a specific message. A call to MPI_WAITANY or
MPI_TESTANY can be used to wait for the completion of one out of several operations. A
call to MPI_WAITALL or MPI_TESTALL can be used to wait for all pending operations in
a list. A call to MPI_WAITONE or MPI_TESTONE can be used to complete all enabled
operations in a list.

MPI_WAITANY (count, array_of_requests, index, status)
one in Fortran.) If the request was allocated by a nonblocking communication operation, then it is deallocated and the request handle is set to MPI_REQUEST_NULL.

The array of requests list may contain null or inactive handles. If the list contains no active handles (list has length zero or all entries are null or inactive), then the call returns immediately with index = MPI_UNDEFINED, and an empty status.

The execution of MPI_WAItAny(count, array of requests, index, status) has the same effect as the execution of MPI_WAIT(array of requests[i], status), where i is the value returned by index (unless the value of index is MPI_UNDEFINED). MPI_WAItAny with an array containing one active entry is equivalent to MPI_WAIT.

MPI_WAItAny(count, array of requests, index, flag, status)
IN count list length (integer)
INOUT array_of_requests array of requests (array of handles)
OUT index index of operation that completed, or MPI_UNDEFINED if none completed (integer)
OUT flag true if one of the operations is complete (logical)
OUT status status object (Status)

The function MPI_WAItAny returns with flag = true exactly in those situations where the function MPI_WAItAny returns; both functions return in that case the same values in the remaining parameters. Thus, a blocking MPI_WAItAny can be easily replaced by a nonblocking MPI_WAItestAny. The same relation holds for the other pairs of Wait and Test functions defined in this section. (End of rationale.)

MPI_WAIt[(count, array of requests, array of statuses)
IN count list length (integer)
INOUT array_of_requests array of requests (array of handles)
OUT array_of_statuses array of status objects (array of Status)

int MPI_Wait(int count, MPI_Request *array of requests, MPI_Status *array of statuses)

int MPI_WaIt[(count, array of requests, array of_statuses, IERROR)
INTEGER COUNT, ARRAY OF REQUESTS, ARRAY OF STATUS, IERROR
INTEGER ARRAY OF REQUESTS(*)
INTEGER ARRAY OF SETTINGS,(MPI_STATUS_SIZE, +), IERROR

Blocks until all communication operations associated with active handles in the list complete, and return the status of all these operations (this includes the case where no handle in the list is active). Both arrays have the same number of valid entries. The i-th entry in array of statuses is set to the return status of the i-th operation. Requests that were created by nonblocking communication operations are deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. The list may contain null or inactive handles. The call sets to empty all the status of each such entry.

The error-free execution of MPI_WAItAny(count, array of requests, array of statuses) has the same effect as the execution of MPI_WAIt(count, array of requests[i], &array of statuses[i]), for i = 0, 1, ..., count-1, in some arbitrary order. MPI_WAIt with an array of length one is equivalent to MPI_WAIt.

When one or more of the communications completed by a call to MPI_WAItAny fail, it is desirable to return specific information on each communication. The function MPI_WAIt will return in such case the error code MPI_ERR_IN_STATUS and will set the error field of each status to a specific error code. This code will be MPI_SUCCESS, if the specific communication completed; it will be another specific error code, if it failed; or it can be MPI_ERR_PENDING if it has neither failed nor completed. The function MPI_WAItAny will return MPI_SUCCESS if no request had an error, or will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

Rationale. This design streamlines error handling in the application. The application code need only test the (single) function result to determine if an error has occurred. It needs to check each individual status only when an error occurred. (End of rationale.)
CHAPTER 3. POINT-TO-POINT COMMUNICATION

MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)

IN count lists length (integer)
INOUT array_of_requests array of requests (array of handles)
OUT flag (logical)
OUT array_of_statuses array of status objects (array of Status)

Returns flag = true if all communications associated with active handles in the array have completed (this includes the case where no handle in the list is active). In this case, each status entry that corresponds to an active handle is set to the status of the corresponding communication; if the request was allocated by a nonblocking communication call, then it is deallocated, and the handle is set to MPI_REQUEST_NULL. Each status entry that corresponds to a null or inactive handle is set to empty. Otherwise, flag = false is returned, no request is modified and the values of the status entries are undefined. This is a local operation.

Errors that occurred during the execution of MPI_TESTALL are handled as errors in MPI_WAITSOME.

MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN incount length of array_of_requests (integer)
INOUT array_of_requests array of requests (array of handles)
OUT outcount number of completed requests (integer)
OUT array_of_indices array of indices of operations that completed (array of integers)
OUT array_of_statuses array of status objects for operations that completed (array of Status)

Waits until at least one of the operations associated with active handles in the list have completed. Returns in outcount the number of requests from the list array_of_requests that have completed. Returns in the first outcount locations of the array array_of_indices the indices of these operations (index within the array array_of_requests; the array is indexed from zero in C and from one in Fortran). Returns in the first outcount locations of the array array_of_status the status for these completed operations. If a request that completed was allocated by a nonblocking communication call, then it is deallocated, and the associated handle is set to MPI_REQUEST_NULL. If the list contains no active handles, then the call returns immediately with outcount = MPI_UNDEFINED.

When one or more of the communications completed by MPI_WAITSOME fails, then it is desirable to return specific information on each communication. The arguments outcount, array_of_indices and array_of_statuses will be adjusted to indicate completion of all communications that have succeeded or failed. The call will return the error code MPI_ERR_INTRSTATUS and the error field of each status returned will be set to indicate success or to indicate the specific error that occurred. The call will return MPI_SUCCESS if no request resulted in an error, and will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

MPI_TESTSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN incount length of array_of_requests (integer)
INOUT array_of_requests array of requests (array of handles)
OUT outcount number of completed requests (integer)
OUT array_of_indices array of indices of operations that completed (array of integers)
OUT array_of_statuses array of status objects for operations that completed (array of Status)

Behaves like MPI_WAITSOME, except that it returns immediately. If no operation has completed it returns outcount = 0. If there is no active handle in the list it returns outcount = MPI_UNDEFINED.

MPI_WAITSOME is a local operation, which returns immediately, whereas MPI_WAITSOME will block until a communication completes; if it was passed a list that contains at least one active handle. Both calls fulfill a fairness requirement. If a request for a receive repeatedly appears in a list of requests passed to MPI_WAITSOME or MPI_TESTSOME, and a matching send has been posted, then the receive will eventually succeed, unless the send is satisfied by another receive, and similarly for send requests.

Errors that occur during the execution of MPI_TESTSOME are handled as for MPI_WAITSOME.

Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use of MPI_TESTANY. The former returns information on all completed communications,
with the latter, a new call is required for each communication that completes.

A server with multiple clients can use MPI_WAITSOME so as not to starve any
client. Clients send messages to the server with service requests. The server calls
MPI_WAITSOME with one receive request for each client, and then handles all re-
quests that completed. If a call to MPI_WAITANY is used instead, then one client
could starve while requests from another client always sneak in first. (End of advice
to users.)

Advice to implementors. MPI_TESTSOME should complete as many pending com-
munications as possible. (End of advice to implementors.)

Example 3.15 Client-server code (starvation can occur).

```fortran
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank > 0) THEN  ! client code
DO WHILE(.TRUE.)
   CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
   CALL MPI_WAITSOME(size, request_list, numdone, indices, statuses, ierr)
   DO i=1, numdone
      CALL MPI_IRECV(a(1,indices(i)), n, MPI_REAL, indices(i), tag,
                     comm, requests(indices(i)), ierr)
      CALL DO_SERVICE(a(1,indices(i))) ! handle one message
   END DO
END DO
ELSE ! rank=0 -- server code
DO i=1, size-1
   CALL MPI_ISEND(a, n, MPI_REAL, i, tag, comm, request_list(i), ierr)
   CALL MPI_WAIT(request_list(i), ierr)
END DO
DO WHILE(.TRUE.)  ! rank=0 -- server code
   CALL MPI_IRECV(a(1,i), n, MPI_REAL, i tag, comm, request_list(i), ierr)
   CALL DO_SERVICE(a(1,i)) ! handle one message
   END DO
END IF
```

Example 3.16 Same code, using MPI_WAITANY.

```fortran
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank > 0) THEN  ! client code
DO WHILE(.TRUE.)
   CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
   CALL MPI_WAIT(request, status, ierr)
END DO
ELSE ! rank=0 -- server code
DO i=1, size-1
   CALL MPI_ISEND(a(1,i), n, MPI_REAL, i, tag, comm, request_list(i), ierr)
END DO
DO WHILE(.TRUE.)
   CALL MPI_IRECV(a(1,indices(i)), n, MPI_REAL, indices(i), tag,
                  comm, requests(indices(i)), ierr)
   CALL DO_SERVICE(a(1,indices(i))) ! handle one message
   END DO
END IF
```

3.8 Probe and Cancel

The MPI_PROBE and MPI_IProbe operations allow incoming messages to be checked for,
without actually receiving them. The user can then decide how to receive them, based on
the information returned by the probe (basically, the information returned by status). In
particular, the user may allocate memory for the receive buffer, according to the length
of the probed message.

The MPI_CANCEL operation allows pending communications to be canceled. This is
required for cleanup. Posting a send or a receive ties up user resources (send or receive
buffers), and a cancel may be needed to free these resources gracefully.

```fortran
CALL MPI_PROBE(source, tag, comm, flag, status)
IN source source rank, or MPI_ANY_SOURCE (integer)
IN tag tag value or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT flag (logical)
OUT status status object (Status)
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
               MPI_Status *status)
MPI_IProbe(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
MPI_IProbe(source, tag, comm, flag, status) returns flag = true if there is a message
that can be received and that matches the pattern specified by the arguments source, tag,
and comm. The call matches the same message that would have been received by a call to
MPI_RECV(…, source, tag, comm, status) executed at the same point in the program, and
returns in status the same value that would have been returned by MPI_RECV(). Otherwise,
the call returns flag = false, and leaves status undefined.

If MPI_IProbe returns flag = true, then the content of the status object can be sub-
sequently accessed as described in section 3.2.5 to find the source, tag and length of the
probed message.
A subsequent receive executed with the same context, and the source and tag returned in status by MPI_PROBE will receive the message that was matched by the probe, if no other intervening receive occurs after the probe. If the receiving process is multi-threaded, it is the user’s responsibility to ensure that the last condition holds.

The source argument of MPI_PROBE can be MPI_ANY_SOURCE, and the tag argument can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source and/or with an arbitrary tag. However, a specific communication context must be provided with the comm argument.

It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is received.

```fortran
MPI_PROBE(source, tag, comm, status)
IN source source rank, or MPI_ANY_SOURCE (integer)
IN tag tag value, or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT status status object (Status)

1 int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
2
3 MPI_PROBE(SOURCE, TAG, COMM, STATUS, ERROR)
4 INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), ERROR

MPI_PROBE behaves like MPI_PROBE except that it is a blocking call that returns only after a matching message has been found.

The implementation of MPI_PROBE and MPI_Wait needs to guarantee progress: if a call to MPI_PROBE has been issued by a process and a send that matches the probe has been initiated by some process, then the call to MPI_PROBE will return, unless the message is received by another concurrent receive operation (that is executed by another thread at the probing process). Similarly, if a process busy waits with MPI_PROBE and a matching message has been issued, then the call to MPI_PROBE will eventually return flag = true unless the message is received by another concurrent receive operation.

Example 3.17 Use blocking probe to wait for an incoming message.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
ELSE IF(rank.EQ.1) THEN
  CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
ELSE
  DO i=1, 2
    CALL MPI_PROBE(MPI_ANY_SOURCE, 0, comm, status, ierr)
    IF (status(MPI_SOURCE) = 0) THEN
      CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, 0, status, ierr)
    ELSE
      CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE, 0, status, ierr)
    END IF
  END DO
END IF
END DO
END IF
```

Example 3.18 A similar program to the previous example, but now it has a problem.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
ELSE IF(rank.EQ.1) THEN
  CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
ELSE
  DO i=1, 2
    CALL MPI_PROBE(MPI_ANY_SOURCE, 0, comm, status, ierr)
    IF (status(MPI_SOURCE) = 0) THEN
      CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE, 0, status, ierr)
    ELSE
      CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, 0, status, ierr)
    END IF
  END DO
END IF
END IF
```

Example 3.19 Use blocking probe to wait for an incoming message.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
ELSE IF(rank.EQ.1) THEN
  CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
ELSE
  DO i=1, 2
    CALL MPI_PROBE(MPI_ANY_SOURCE, 0, comm, status, ierr)
    IF (status(MPI_SOURCE) = 0) THEN
      CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, 0, status, ierr)
    ELSE
      CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE, 0, status, ierr)
    END IF
  END DO
END IF
END IF
```

3.8 Probe and Cancel

Advice to implementors. A call to MPI_PROBE(source, tag, comm, status) will match the message that would have been received by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point. Suppose that this message has source s, tag t and communicator c. If the tag argument in the probe call has value MPI_ANY_TAG then the message probed will be the earliest pending message from source s with communicator c and any tag; in any case, the message probed will be the earliest pending message from source s with tag t and communicator c (this is the message that would have been received, so as to preserve message order). This message continues as the earliest pending message from source s with tag t and communicator c until it is received. A receive operation subsequent to the probe that uses the same communicator as the probe and uses the tag and source values returned by the probe, must receive this message, unless it has already been received by another receive operation. (End of advice to implementors.)
3.9. Persistent communication requests

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a persistent communication request once and, then, repeatedly using the request to initiate and complete messages. The persistent request thus created can be thought of as a communication port or a “half-channel.” It does not provide the full functionality of a conventional channel, since there is no binding of the send port to the receive port. This construct allows reduction of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another. It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa.

A persistent communication request is created using one of the four following calls. These calls involve no communication.

\[
\text{MPI\_SEND\_INIT}(\text{buf}, \text{count}, \text{datatype}, \text{dest}, \text{tag}, \text{comm}, \text{request})
\]

\[
\begin{array}{ll}
\text{IN} & \text{buf} \quad \text{initial address of send buffer (choice)} \\
\text{IN} & \text{count} \quad \text{number of elements sent (integer)} \\
\text{IN} & \text{datatype} \quad \text{type of each element (handle)} \\
\text{IN} & \text{dest} \quad \text{rank of destination (integer)} \\
\text{IN} & \text{tag} \quad \text{message tag (integer)} \\
\text{IN} & \text{comm} \quad \text{communicator (handle)} \\
\text{OUT} & \text{request} \quad \text{communication request (handle)}
\end{array}
\]

\[
\text{MPI\_TEST\_CANCEL\_FAILED}(\text{status}, \text{flag})
\]

\[
\begin{array}{ll}
\text{IN} & \text{status} \quad \text{status object (Status)} \\
\text{OUT} & \text{flag} \quad \text{(logical)}
\end{array}
\]

\[
\text{int \ MPI\_Status\_get\_flag}(\text{mpi\_status}, \text{flag})
\]

\[
\text{MPI\_TEST\_CANCEL\_FREE}(\text{request})
\]

\[
\text{int \ MPI\_Cancel\_FREE}(\text{request})
\]

Advice to users. Cancel can be an expensive operation that should be used only exceptionally. (End of advice to users.)

Advice to implementors. If a send operation uses an “eager” protocol (data is transferred to the receiver before a matching receive is posted), then the cancellation of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement MPI\_CANCEL, this is still a local operation, since its completion does not depend on the code executed by other processes. If processing is required on another process, this should be transparent to the application (hence the need for an interrupt and an interrupt handler). (End of advice to implementors.)
MPI\_SEND\_INIT(buf, count, datatype, dest, tag, comm, request)

\begin{verbatim}
INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
\end{verbatim}

Creates a persistent communication request for a standard mode send operation, and
binds to it all the arguments of a send operation.

MPI\_BSEND\_INIT(buf, count, datatype, dest, tag, comm, request)

\begin{verbatim}
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
\end{verbatim}

Creates a persistent communication request for a buffered mode send.

MPI\_SSEND\_INIT(buf, count, datatype, dest, tag, comm, request)

\begin{verbatim}
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
\end{verbatim}

Creates a persistent communication object for a ready mode send operation.

MPI\_RSEND\_INIT(buf, count, datatype, dest, tag, comm, request)

\begin{verbatim}
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
\end{verbatim}

Creates a persistent communication object for a synchronous mode send operation.

MPI\_RECV\_INIT(buf, count, datatype, dest, source, tag, comm, request)

\begin{verbatim}
INTEGER COUNT, DATATYPE, DEST, SOURCE, TAG, COMM, REQUEST, IERROR
\end{verbatim}

Creates a persistent communication request for a receive operation. The argument buf
is marked as OUT because the user gives permission to write on the receive buffer by passing
the argument to MPI\_RECV\_INIT.

A persistent communication request is inactive after it was created — no active commu-
nication is attached to the request.

A communication (send or receive) that uses a persistent request is initiated by the
function MPI\_START.
3.10. SEND-RECEIVE

A send operation initiated with MPI\_START can be matched with any receive operation and, likewise, a receive operation initiated with MPI\_START can receive messages generated by any send operation.

The send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the communication subsystem takes care of these issues. The send-receive operation can be used in conjunction with the functions described in Chapter 6 in order to perform shifts on various logical topologies. Also, a send-receive operation is useful for implementing remote procedure calls.

A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation; a send-receive operation can receive a message sent by a regular send operation.

Create (Start Complete) Free, where * indicates zero or more repetitions. If the same communication object is used in several concurrent threads, it is the user’s responsibility to coordinate calls so that the correct sequence is obeyed.
CHAPTER 3. POINT-TO-POINT COMMUNICATION

3.12 Derived datatypes

The special value MPI_PROC_NULL can be used instead of a rank wherever a source or a
destination argument is required in a call. A communication with process MPI_PROC_NULL
has no effect. A send to MPI_PROC_NULL succeeds and returns as soon as possible. A receive
from MPI_PROC_NULL succeeds and returns as soon as possible with no modifications to the
receive buffer. When a receive with source = MPI_PROC_NULL is executed then the status
object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG and count = 0.

3.12 Derived datatypes

Up to here, all point to point communication have involved only contiguous buffers contain-
ing a sequence of elements of the same type. This is too constraining on two accounts. One
often wants to pass messages that contain values with different datatypes (e.g., an integer
count, followed by a sequence of real numbers); and one often wants to send noncontiguous
data (e.g., a sub-block of a matrix). One solution is to pack noncontiguous data into a
continous buffer at the sender site and unpack it back at the receiver site. This has the
disadvantage of requiring additional memory-to-memory copy operations at both sites, even
when the communication subsystem has scatter-gather capabilities. Instead, MPI provides
mechanisms to specify more general, mixed, and noncontiguous communication buffers. It
is up to the implementation to decide whether data should be first packed in a contiguous
buffer before being transmitted, or whether it can be collected directly from where it resides.

The general mechanisms provided here allow one to transfer directly, without copying,
objects of various shape and size. It is not assumed that the MPI library is cognizant of
the objects declared in the host language. Thus, if one wants to transfer a structure, or an
array section, it will be necessary to provide in MPI a definition of a communication buffer
that mimics the definition of the structure or array section in question. These facilities can
be used by library designers to define communication functions that can transfer objects
defined in the host language — by decoding their definitions as available in a symbol table
or a dope vector. Such higher-level communication functions are not part of MPI.

More general communication buffers are specified by replacing the basic datatypes that
have been used so far with derived datatypes that are constructed from basic datatypes using
the constructors described in this section. These methods of constructing derived datatypes
can be applied recursively.

A general datatype is an opaque object that specifies two things:

- A sequence of basic datatypes
- A sequence of integer (byte) displacements

The displacements are not required to be positive, distinct, or in increasing order.
Therefore, the order of items need not coincide with their order in store, and an item may
appear more than once. We call such a pair of sequences (or sequence of pairs) a type
map. The sequence of basic datatypes (displacements ignored) is the type signature of
the datatype.

Let

\[ T_{\text{map}} = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\} \]

be such a type map, where type_i are basic types, and disp_i are displacements. Let

\[ T_{\text{typemap}} = \{type_0, ..., type_{n-1}\} \]
be the associated type signature. This type map, together with a base address buf, specifies a communication buffer: the communication buffer that consists of n entries, where the i-th entry is at address buf + disp[i] and has type type[i]. A message assembled from such a communication buffer will consist of n values, of the types defined by Type[n].

We can use a handle to a datatype as an argument in a send or receive operation, instead of a basic datatype argument. The operation MPI_SEND(buf, 1, datatype, ...) will use the send buffer defined by the base address buf and the general datatype associated with datatype. It will generate a message with the type signature determined by the datatype argument. MPI_RECV(buf, 1, datatype, ...) will use the receive buffer defined by the base address buf and the general datatype associated with datatype.

General datatypes can be used in all send and receive operations. We discuss, in Sec. 3.12.5, the case where the second argument count has value > 1.

The basic datatypes presented in section 3.2.2 are particular cases of a general datatype, and are predefined. Thus, MPI_INT is a predefined handle to a datatype with type map ((int,0)), with one entry of type int and displacement zero. The other basic datatypes are similar.

The extent of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype, rounded up to satisfy alignment requirements. That is, if

\[ T_{\text{typemap}} = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}, \]

then

\[
\begin{align*}
&lb(T_{\text{typemap}}) = \min_{0 \leq i < n} disp_i, \\
&ub(T_{\text{typemap}}) = \max_{0 \leq i < n} (disp_i + size[type_i]) + \epsilon, \text{ and} \\
&\text{extent}(T_{\text{typemap}}) = ub(T_{\text{typemap}}) - lb(T_{\text{typemap}}).
\end{align*}
\]

If type[i] requires alignment to a byte address that is a multiple of \( k_i \), then \( \epsilon \) is the least nonnegative increment needed to round \( \text{extent}(T_{\text{typemap}}) \) to the next multiple of \( k_i \). The complete definition of extent is given on page 71.

Example 3.19 Assume that \( T_{\text{typemap}} = \{(\text{double}, 0), (\text{char}, 8)\} \) (a double at displacement zero, followed by a char at displacement eight). Assume, furthermore, that doubles have to be strictly aligned at addresses that are multiples of eight. Then, the extent of this datatype is 16 (9 rounded to the next multiple of 8). A datatype that consists of a character immediately followed by a double will also have an extent of 16.

Rationale. The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in section 3.12.3. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. (End of rationale.)

3.12.1 Datatype Constructors

 CONTIGUOUS The simplest datatype constructor is MPI_TYPE_CONTIGUOUS which allows replication of a datatype into contiguous locations.

3.12.5 DERIVED DATATYPES

Example 3.20 Let oldtype have type map \{(\text{double}, 0), (\text{char}, 8)\}, with extent 16, and let count = 3. The type map of the datatype returned by newtype is \{(\text{double}, 0), (\text{char}, 8), (\text{double}, 16), (\text{double}, 24), (\text{double}, 32), (\text{char}, 40)\};

i.e., alternating double and char elements, with displacements 0, 8, 16, 24, 32, 40.

In general, assume that the type map of oldtype is \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}, with extent ex. Then newtype has a type map with count \( n \) entries defined by:

\[
\begin{align*}
&\{(type_0, disp_0 + (ex \cdot (count - i) + \min(type_i, disp_i)), \ldots, (type_{n-1}, disp_{n-1} + ex \cdot (count - 1) + \min(type_{n-1}, disp_{n-1}))\}.
\end{align*}
\]

Vector The function MPI_TYPE_VECTOR is a more general constructor that allows replication of a datatype into locations that consist of equally spaced blocks. Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.

\[
\text{MPI_TYPE_VECTOR}(\text{count}, \text{blocklength}, \text{stride}, \text{oldtype}, \text{newtype})
\]

IN count replication count (nonnegative integer)
IN oldtype old datatype (handle)
OUT newtype new datatype (handle)

\[
\text{int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype newtype)}
\]

IN oldtype old datatype (handle)
IN newtype new datatype (handle)
OUT datatype (handle)

\[
\text{int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype newtype)}
\]
MPI_TYPE_VECTOR\( (\text{count}, \text{blocklength}, \text{stride}, \text{oldtype}, \text{newtype}, \text{IERROR}) \)
\[
\text{INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR}
\]

**Example 3.21** Assume, again, that \text{oldtype} has type map \{\{double, 0\}, \{char, 8\}\}, with extent 16. A call to MPI_TYPE_VECTOR\( (2, 3, 4, \text{oldtype}, \text{newtype}) \) will create the datatype with type map,
\[
\{\{\text{double}, 0\}, \{\text{char}, 8\}\}, \{\{\text{double}, 16\}, \{\text{char}, 24\}\}, \{\{\text{double}, 32\}, \{\text{char}, 40\}\},
\{\{\text{double}, 64\}, \{\text{char}, 72\}\}, \{\{\text{double}, 80\}, \{\text{char}, 88\}\}, \{\{\text{double}, 96\}, \{\text{char}, 104\}\}\}.
\]

That is, two blocks with three copies each of the old type, with a stride of 4 elements (4\times16 bytes) between the blocks.

**Example 3.22** A call to MPI_TYPE_VECTOR\( (3, 1\times2, \text{oldtype}, \text{newtype}) \) will create the datatype,
\[
\{\{\text{double}, 0\}, \{\text{char}, 8\}\}, \{\{\text{double}, -32\}, \{\text{char}, -24\}\}, \{\{\text{double}, -64\}, \{\text{char}, -56\}\}\}.
\]

In general, assume that \text{oldtype} has type map,
\[
\{(\text{type}0, \text{disp}0), \ldots, (\text{type}_{\text{-}1}, \text{disp}_{\text{-}1})\}
\]
with extent \(ex\). Let \(bl\) be the blocklength. The newly created datatype has a type map with count \(bl\times n\) entries:
\[
\{(\text{type}0, \text{disp}0), \ldots, (\text{type}_{\text{-}1}, \text{disp}_{\text{-}1})\},
\{(\text{type}0, \text{disp}0 + ex), \ldots, (\text{type}_{\text{-}1}, \text{disp}_{\text{-}1} + ex)\}, \ldots,
\{(\text{type}0, \text{disp}0 + \text{stride} \times ex), \ldots, (\text{type}_{\text{-}1}, \text{disp}_{\text{-}1} + \text{stride} \times ex)\}, \ldots,
\{(\text{type}0, \text{disp}0 + \text{stride} \times (\text{count} - 1) \times ex), \ldots,
\{(\text{type}_{\text{-}1}, \text{disp}_{\text{-}1} + \text{stride} \times (\text{count} - 1) \times ex), \ldots,
\{(\text{type}0, \text{disp}0 + \text{stride} \times (\text{count} - 1) + \text{bl} \times ex), \ldots,
\{(\text{type}_{\text{-}1}, \text{disp}_{\text{-}1} + \text{stride} \times (\text{count} - 1) + \text{bl} \times ex)\}.
\]

A call to MPI_TYPE_CONTIGUOUS\( (\text{count}, \text{oldtype}, \text{newtype}) \) is equivalent to a call to MPI_TYPE_VECTOR\( (1, \text{count}, \text{oldtype}, \text{newtype}) \), or to a call to MPI_TYPE_VECTOR\( (\text{count}, \text{oldtype}, \text{newtype}) \), \(n\) arbitrary.

### 3.12 Derived Datatypes

Hvector The function MPI_TYPE_HVECTOR is identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The use for both types of vector constructors is illustrated in Sec. 3.12.7. (H stands for “heterogeneous”).

MPI_TYPE_HVECTOR\( (\text{count}, \text{blocklength}, \text{stride}, \text{oldtype}, \text{newtype}) \)
\[
\begin{align*}
\text{IN} & \quad \text{count} & \quad \text{number of blocks (nonnegative integer)} \\
\text{IN} & \quad \text{blocklength} & \quad \text{number of elements in each block (nonnegative integer)} \\
\text{IN} & \quad \text{stride} & \quad \text{number of bytes between start of each block (integer)} \\
\text{IN} & \quad \text{oldtype} & \quad \text{old datatype (handle)} \\
\text{OUT} & \quad \text{newtype} & \quad \text{new datatype (handle)} \\
\end{align*}
\]

\[
\text{int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)}
\]

**MPI_TYPE_HVECTOR\( (\text{COUNT}, \text{BLOCKLENGTH}, \text{STRIDE}, \text{OLDTYPE}, \text{NEWTYPE}, \text{IERROR}) \)**
\[
\text{INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR}
\]

Assume that \text{oldtype} has type map,
\[
\{(\text{type}0, \text{disp}0), \ldots, (\text{type}_{\text{-}1}, \text{disp}_{\text{-}1})\},
\]
with extent \(ex\). Let \(bl\) be the blocklength. The newly created datatype has a type map with count \(bl\times n\) entries:
\[
\{(\text{type}0, \text{disp}0), \ldots, (\text{type}_{\text{-}1}, \text{disp}_{\text{-}1})\},
\{(\text{type}0, \text{disp}0 + ex), \ldots, (\text{type}_{\text{-}1}, \text{disp}_{\text{-}1} + ex)\}, \ldots,
\{(\text{type}0, \text{disp}0 + \text{stride} \times ex), \ldots, (\text{type}_{\text{-}1}, \text{disp}_{\text{-}1} + \text{stride} \times ex)\}, \ldots,
\{(\text{type}0, \text{disp}0 + \text{stride} \times (\text{count} - 1) \times ex), \ldots,
\{(\text{type}_{\text{-}1}, \text{disp}_{\text{-}1} + \text{stride} \times (\text{count} - 1) \times ex), \ldots,
\{(\text{type}0, \text{disp}0 + \text{stride} \times (\text{count} - 1) + \text{bl} \times ex), \ldots,
\{(\text{type}_{\text{-}1}, \text{disp}_{\text{-}1} + \text{stride} \times (\text{count} - 1) + \text{bl} \times ex)\}.
Indexed The function MPI\_TYPE\_INDEXED allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

MPI\_TYPE\_INDEXED( count, array\_of\_blocklengths, array\_of\_displacements, oldtype, newtype)

IN count number of blocks – also number of entries in array\_of\_displacements and array\_of\_blocklengths (nonnegative integer)

IN array\_of\_blocklengths number of elements per block (array of nonnegative integers)

IN array\_of\_displacements displacement for each block, in multiples of oldtype extent (array of integer)

IN oldtype old datatype (handle)

OUT newtype new datatype (handle)

int MPI\_Type\_indexed(int count, int *array\_of\_blocklengths, int array\_of\_displacements, MPI\_Datatype oldtype, MPI\_Datatype newtype)

MPI\_TYPE\_INDEXED(COUNT, ARRAY\_OF\_BLOCLLENGTHS, ARRAY\_OF\_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)

INTEGER COUNT, ARRAY\_OF\_BLOCLLENGTHS(*), ARRAY\_OF\_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR

Example 3.23 Let oldtype have type map \{(double, 0), (char, 8)\}, with extent 16. Let B = (3, 1) and let D = (4, 0). A call to MPI\_TYPE\_INDEXED(2, B, D, oldtype, newtype) returns a datatype with type map,

\{(double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104),
 (double, 0), (char, 8)\}.

That is, three copies of the old type starting at displacement 64, and one copy starting at displacement 0.

In general, assume that oldtype has type map,

\{(type\_0, disp\_0), ..., (type\_n-1, disp\_n-1)\},

with extent ex. Let B be the array\_of\_blocklength argument and D be the array\_of\_displacements argument. The newly created datatype has \{(\sum_{i=0}^{n-1} B[i])\} entries:

\{(type\_0, disp\_0 + D[0], ex), ..., (type\_n-1, disp\_n-1 + D[0], ex), ...,}
with extent \( ex \). Let \( B \) be the \texttt{array_of_blocklength} argument and \( D \) be the \texttt{array_of_displacements} argument. The newly created datatype has a type map with \( \sum_{0}^{\text{count}-1} B[i] \) entries:

\[
\{(\text{type}_0, \text{disp}_0, 0), \ldots, (\text{type}_{\text{count}-1}, \text{disp}_{\text{count}-1}, 0)\}, \\
(\text{type}_0, \text{disp}_0, 0 + (\text{B}[0] - 1) \cdot \text{ex}), \ldots, \\
(\text{type}_{\text{count}-1}, \text{disp}_{\text{count}-1}, 0 + (\text{B}[\text{count} - 1] - 1) \cdot \text{ex})
\]

Struct \texttt{MPI_TYPE_STRUCT} is the most general type constructor. It further generalizes the previous one in that it allows each block to consist of replications of different datatypes.

\[
\text{MPI_TYPE_STRUCT}(\text{count, array_of_blocklengths, array_of_displacements, array_of_types, newtype})
\]

| IN          | count | number of blocks (integer) – also number of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths |
| IN          | array_of_blocklength | number of elements in each block (array of integer) |
| IN          | array_of_displacements | byte displacement of each block (array of integer) |
| IN          | array_of_types | type of elements in each block (array of handles to datatype objects) |
| OUT         | newtype | new datatype (handle) |

\texttt{int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)}

\texttt{MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)}

\texttt{INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*), ARRAY_OF_TYPES(*), NEWTYPE, IERROR)

Example 3.24 Let \texttt{type1} have type map:

\[
\{(\text{double}, 0), (\text{char}, 8)\}
\]

with extent 16. Let \( B = (2, 1, 3), D = (0, 16, 26) \), and \( T = (\text{MPI_FLOAT}, \text{type1}, \text{MPI_CHAR}) \). Then a call to \texttt{MPI_TYPE_STRUCT(3, B, D, T, newtype)} returns a datatype with type map:

\[
\{(\text{float}, 0), (\text{float}, 4), (\text{double}, 16), (\text{char}, 24), (\text{char}, 26), (\text{char}, 27), (\text{char}, 28)\}
\]

\text{3.12. DERIVED DATATYPES}

That is, two copies of \texttt{MPI_FLOAT} starting at 0, followed by one copy of \texttt{type1} starting at 16, followed by three copies of \texttt{MPI_CHAR} starting at 26. (We assume that a float occupies four bytes.)

In general, let \( T \) be the \texttt{array_of_types} argument, where \( T[i] \) is a handle to,

\[
\text{typemap}_i = \{(\text{type}_0, \text{disp}_0, 0), (\text{type}_{i-1}, \text{disp}_{i-1}, 0)\},
\]

with extent \( ex_i \). Let \( B \) be the \texttt{array_of_blocklength} argument and \( D \) be the \texttt{array_of_displacements} argument. Let \( c \) be the count argument. Then the newly created datatype has a type map with \( \sum_{0}^{c-1} B[i] \) \( n_i \) entries:

\[
\{(\text{type}_0, \text{disp}_0, 0), (\text{type}_{n_0}, \text{disp}_{n_0}, 0 + D[0]), \ldots, \\
(\text{type}_{n_0}, \text{disp}_{n_0}, 0 + (\text{B}[0] - 1) \cdot \text{ex}_0), \ldots, \\
(\text{type}_{n_1}, \text{disp}_{n_1}, 0 + (\text{B}[\text{count} - 1] - 1) \cdot \text{ex}_{c-1})
\]

A call to \texttt{MPI_TYPE_HINDEXED(count, B, D, oldtype, newtype)} is equivalent to a call to \texttt{MPI_TYPE_STRUCT(count, B, D, T, newtype)}, where each entry of \( T \) is equal to \( \text{oldtype} \).

\text{3.12.2 Address and extent functions}

The displacements in a general datatype are relative to some initial buffer address. \texttt{Absolute addresses} can be substituted for these displacements: we treat them as displacements relative to “address zero” the start of the address space. This initial address zero is indicated by the constant \texttt{MPI_BOTTOM}. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the buf argument is passed the value \texttt{MPI_BOTTOM}.

The address of a location in memory can be found by invoking the function \texttt{MPI_ADDRESS}.

\texttt{MPI_ADDRESS(location, address)}

| IN | location | location in caller memory (choice) |
| OUT | address | address of location (integer) |

\texttt{int MPI_Address(void* location, MPI_Aint *address)}

\texttt{MPI_ADDRESS(LOCATION, ADDRESS, IERROR)}

\texttt{<type> LOCATION(\*)}

\texttt{INTEGER ADDRESS, IERROR}

\text{Returns the (byte) address of location.}
Example 3.25 Using \

```fortran
REAL A(100,100)
INTEGER I1, I2, DIFF
CALL MPI_ADDRESS(A(1,1), I1, IERROR)
CALL MPI_ADDRESS(A(10,10), I2, IERROR)
DIFF = I2 - I1
! The value of DIFF is 909*sizeofreal; the values of I1 and I2 are
! implementation dependent.
```

Advice to users. C users may be tempted to avoid the usage of 

```fortran
MPI_ADDRESS and rely on the availability of the address operator &. Note, however, that & cast-
expression is a pointer, not an address. ANSI C does not require that the value of a
pointer (or the pointer cast to it) be the absolute address of the object pointed at—
although this is commonly the case. Furthermore, referencing may not have a unique
definition on machines with a segmented address space. The use of 

```fortran
MPI_ADDRESS to “reference” C variables guarantees portability to such machines as well. (End of
```

Advice to users.)

The following auxiliary functions provide useful information on derived datatypes.

```fortran
MPI_TYPE_EXTENT(datatype, extent)
IN  datatype  datatype (handle)
OUT extent       datatype extent (integer)
```

```fortran
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

```fortran
MPI_TYPE_EXTENT(Datatype, EXTENT, IERROR)
INTEGER DATATYPE, EXTENT, IERROR

Returns the extent of a datatype, where extent is as defined on page 71.
```

```fortran
MPI_TYPE_SIZE(datatype, size)
IN  datatype  datatype (handle)
OUT size       datatype size (integer)
```

```fortran
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

```fortran
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
INTEGER DATATYPE, SIZE, IERROR

MPI_TYPE_SIZE returns the total size, in bytes, of the entries in the type signature
associated with datatype; i.e., the total size of the data in a message that would be created
with this datatype. Entries that occur multiple times in the datatype are counted with
their multiplicity.
```

3.12. DERIVED DATATYPES

3.12.3 Lower-bound and upper-bound markers

It is often convenient to define explicitly the lower bound and upper bound of a type map,
and override the definition given on page 71. This allows one to define a datatype that has
“holes” at its beginning or its end, or a datatype with entries that extend above the upper
bound or below the lower bound. Examples of such usage are provided in Sec. 3.12.7. Also,
the user may want to override the alignment rules that are used to compute upper bounds
and extents. E.g., a C compiler may allow the user to override default alignment rules for
some of the structures within a program. The user has to specify explicitly the bounds of
the datatypes that match these structures.

To achieve this, we add two additional “pseudo-datatypes,” 

```fortran
MPI_LB and MPI UB, that can be used, respectively, to mark the lower bound or the upper bound of a datatype. These
pseudo-datatypes occupy no space (extent(MPI_LB) = extent(MPI UB) = 0). They do not
affect the size or count of a datatype, and do not affect the content of a message created
with this datatype. However, they do affect the definition of the extent of a datatype and,
therefore, affect the outcome of a replication of this datatype by a datatype constructor.
```

Example 3.26 Let D = (-3, 0, 6); T = (MPI LB, MPI LINT, MPI UB), and B = (1, 1, 1).
Then a call to 

```fortran
MPI_TYPE_STRUCT(3, B, D, T, type) 
```
creates a new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer at displacement 0. This is
the datatype defined by the sequence 

```fortran
{(lb, -3), (int, 0), (ub, 6)}
```
If this type is replicated twice by a call to 

```fortran
MPI_TYPE_CONTIGUOUS(2, type1, type2)
```
then the newly created type

```fortran
can be described by the sequence 

```fortran
{(lb, -3), (int, 0), (int, 9), (ub, 15)}
```
(An entry of type ub

can be deleted if there is another entry of type ub with a higher displacement; an entry of
type lb can be deleted if there is another entry of type lb with a lower displacement.)

In general, if

```fortran
Typemap = {(type0, disp0),...(type n, disp n-1)};
```
then the lower bound of Typemap is defined to be

```fortran
lb(Typemap) = \begin{cases} \min_j \{disp_j\} & \text{if no entry has basic type lb} \\
\min_j \{disp_j\} \text{such that type}_j = \text{lb} & \text{otherwise}
\end{cases}
```

Similarly, the upper bound of Typemap is defined to be

```fortran
ub(Typemap) = \begin{cases} \max_j \{disp_j + \text{sizeof(type}_j\} + \epsilon & \text{if no entry has basic type ub} \\
\max_j \{disp_j \text{such that type}_j = \text{ub}\} & \text{otherwise}
\end{cases}
```

Then

```fortran
extent(Typemap) = ub(Typemap) - lb(Typemap)
```
If type requires alignment to a byte address that is a multiple of k, then \( \epsilon \) is the least
nonnegative increment needed to round extent(Typemap) to the next multiple of max(k).

The formal definitions given for the various datatype constructors apply now, with the
amended definition of extent.

The two functions below can be used for finding the lower bound and the upper bound of a datatype.
MPI_TYPE_LB(datatype, displacement)
IN datatype datatype (handle)
OUT displacement displacement of lower bound from origin, in bytes (integer)

int MPI_Type_lb(datatype, AInt* displacement)
MPI_TYPE_LB(DATATYPE, DISPLACEMENT, IERROR)
INTEGER DATATYPE, DISPLACEMENT, IERROR

MPI_TYPE_UB(datatype, displacement)
IN datatype datatype (handle)
OUT displacement displacement of upper bound from origin, in bytes (integer)

int MPI_Type_ub(datatype, AInt* displacement)
MPI_TYPE_UB(DATATYPE, DISPLACEMENT, IERROR)
INTEGER DATATYPE, DISPLACEMENT, IERROR

3.12.4 Commit and free

A datatype object has to be committed before it can be used in a communication. A committed datatype can still be used as an argument in datatype constructors. There is no need to commit basic datatypes. They are "pre-committed."

MPI_TYPE_COMMIT(datatype)
INOUT datatype datatype that is committed (handle)

int MPI_Type_commit(datatype)
MPI_TYPE_COMMIT(DATATYPE, IERROR)
INTEGER DATATYPE, IERROR

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

Advice to implementors. The system may "compile" at commit time an internal representation for the datatype that facilitates communication, e.g., change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (End of advice to implementors.)

MPI_TYPE_FREE(datatype)

int MPI_Type_free(datatype)
MPI_TYPE_FREE(DATATYPE, IERROR)
INTEGER DATATYPE, IERROR

Marks the datatype object associated with datatype for deallocation and sets datatype to MPI_DATATYPE_NULL. Any communication that is currently using this datatype will complete normally. Derived datatypes that were defined from the freed datatype are not affected.

Example 3.27 The following code fragment gives examples of using MPI_TYPE_COMMIT.

Example 3.27

INTEGER type1, type2
CALL MPI_Type_COMMIT(type1, ierr)
! new type object created
CALL MPI_Type_COMMIT(type1, ierr)
! new type can be used for communication
type2 = type1
! type2 can be used for communication
CALL MPI_Type_COMMIT(type1, ierr)
! (it is a handle to same object as type1)
CALL MPI_Type_COMMIT(3, 5, 4, MPI_REAL, type1, ierr)
! new uncommitted type object created
CALL MPI_Type_COMMIT(type1, ierr)
! new type can be used anew for communication

Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value.

Advice to implementors. The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather than copying them. In this case, one needs to keep track of active datatype definition references in order to know when a datatype object can be freed. (End of advice to implementors.)

3.12.5 Use of general datatypes in communication

Handles to derived datatypes can be passed to a communication call wherever a datatype argument is required. A call of the form MPI_SEND(buf, count, datatype, ...), where count > 1, is interpreted as if the call was passed a new datatype which is the concatenation of count copies of datatype. Thus, MPI_SEND(buf, count, datatype, dest, tag, comm) is equivalent to,

MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
MPI_TYPE_COMMIT(newtype)
MPI_SEND(buf, 1, newtype, dest, tag, comm).
Similar statements apply to all other communication functions that have a count and datatype argument.

Suppose that a send operation MPI_SEND(buf, count, datatype, dest, tag, comm) is executed, where datatype has type map,

\{(typeo, disp0), ..., (typen-1, dispn-1)\},

and extent extent. (Empty entries of “pseudo-type” MPI UB and MPI LB are not listed in the type map, but they affect the value of extent.) The send operation sends n-count entries, where entry i: n + j is at location addrij = buf + extent \cdot i + dispj and has type typij, for i = 0, ..., count - 1 and j = 0, ..., n - 1. These entries need not be contiguous, nor distinct; their order can be arbitrary.

The variable stored at address addrij in the calling program should be of a type that matches typij, where type matching is defined as in section 3.3.1. The message sent contains n-count entries, where entry i: n + j has type typij.

Similarly, suppose that a receive operation MPI_RECV(buf, count, datatype, source, tag, comm, status) is executed, where datatype has type map,

\{(typeo, disp0), ..., (typen-1, dispn-1)\},

with extent extent. (Again, empty entries of “pseudo-type” MPI UB and MPI LB are not listed in the type map, but they affect the value of extent.) This receive operation receives n-count entries, where entry i: n + j is at location buf + extent \cdot i + dispj and has type typij. If the incoming message consists of k elements, then we must have k ≤ n-count; the i: n + j-th element of the message should have a type that matches typij.

Type matching is defined according to the type signature of the corresponding datatypes, that is, the sequence of basic type components. Type matching does not depend on some aspects of the datatype definition, such as the displacements (layout in memory) or the intermediate types used.

Example 3.28 This example shows that type matching is defined in terms of the basic types that a derived type consists of.

...  
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, type2, ...)  
CALL MPI_TYPE_CONTIGUOUS(4, MPI_REAL, type4, ...)  
CALL MPI_TYPE_CONTIGUOUS(2, type2, type22, ...)  
...  
CALL MPI_SEND(a, 4, MPI_REAL, ...)  
CALL MPI_SEND(a, 2, type2, ...)  
CALL MPI_SEND(a, 1, type22, ...)  
CALL MPI_SEND(a, 1, type4, ...)  
...  
CALL MPI_RECV(a, 4, MPI_REAL, ...)  
CALL MPI_RECV(a, 2, type2, ...)  
CALL MPI_RECV(a, 1, type22, ...)  
CALL MPI_RECV(a, 1, type4, ...)  
...  
Each of the sends matches any of the receives.

3.12. DERIVED DATATYPES

A datatype may specify overlapping entries. The use of such a datatype in a receive operation is erroneous. (This is erroneous even if the actual message received is short enough not to write any entry more than once.)

A datatype may specify overlapping entries. If such a datatype is used in a receive operation, that is, if some part of the receive buffer is written more than once by the receive operation, then the call is erroneous.

Suppose that MPI_RECV(buf, count, datatype, dest, tag, comm, status) is executed, where datatype has type map,

\{(typeo, disp0), ..., (typen-1, dispn-1)\}.

The received message need not fill all the receive buffer, nor does it need to fill a number of locations which is a multiple of n. Any number, k, of basic elements can be received, where 0 ≤ k ≤ count. The number of basic elements received can be retrieved from status using the query function MPI_GET_ELEMENTS.

MPI_GET_ELEMENTS(stat, datatype, count)

IN status return status of receive operation (Status)
IN datatype datatype used by operation (handle)
OUT count number of received basic elements (integer)
int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)
MPI_GET_ELEMENTS(status, datatype, count, ierr)
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

The previously defined function MPI_GET_COUNTER (Sec. 3.2.5), has a different behavior. It returns the number of “top-level entries” received, i.e. the number of “copies” of type datatype. In the previous example, MPI_GET_COUNTER may return any integer value k, where 0 ≤ k ≤ count. If MPI_GET_COUNTER returns k, then the number of basic elements received (and the value returned by MPI_GET_ELEMENTS) is n · k. If the number of basic elements received is not a multiple of n, that is, if the receive operation has not received an integral number of datatype “copies,” then MPI_GET_COUNTER returns the value MPI_UNDEFINED.

Example 3.29 Usage of MPI_GET_COUNTER and MPI_GET_UNDEFINED.

...  
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
CALL MPI_TYPE_COMMIT(Type2, ierr)
...  
CALL MPI_RECV(a, 4, MPI_REAL, ...)  
CALL MPI_RECV(a, 2, Type2, ...)  
CALL MPI_RECV(a, 1, type22, ...)  
CALL MPI_RECV(a, 1, type4, ...)  
...  
CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
...  
IF(rank.EQ.0) THEN  
CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
ELSE  
END  
CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
CALL MPI_GET_COUNTER(stat, Type2, 1, ierr) ! returns i+1
The function MPI_GET_ELEMENTS can also be used after a probe to find the number of elements in the probed message. Note that the two functions MPI_GET_COUNT and MPI_GET_ELEMENTS return the same values when they are used with basic datatypes.

**Rationale.** The extension given to the definition of MPI_GET_COUNT seems natural: one would expect this function to return the value of the count argument, when the receive buffer is filled. Sometimes datatype represents a basic unit of data one wants to transfer, for example, a record in an array of records (structures). One should be able to find out how many components were received without bothering to divide by the number of elements in each component. However, on other occasions, datatype is used to define a complex layout of data in the receiver memory, and does not represent a basic unit of data for transfers. In such cases, one needs to use the function MPI_GET_ELEMENTS. (End of rationale.)

**Advice to implementors.** The definition implies that a receive cannot change the value of storage outside the entries defined to compose the communication buffer. In particular, the definition implies that packing space in a structure not be modified when such a structure is copied from one process to another. This would prevent the obvious optimization of copying the structure, together with the packing, as one contiguous block. The implementation is free to do this optimization when it does not impact the outcome of the computation. The user can "force" this optimization by explicitly including padding as part of the message. (End of advice to implementors.)

### 3.12.6 Correct use of addresses

Successively declared variables in C or Fortran are not necessarily stored at contiguous locations. Thus, care must be exercised that displacements do not cross from one variable to another. Also, in machines with a segmented address space, addresses are not unique and address arithmetic has some peculiar properties. Thus, the use of addresses, that is, displacements relative to the start address MPI_BOTTOM, has to be restricted.

Variables belong to the same sequential storage if they belong to the same array, to the same COMMON block in Fortran, or to the same structure in C. Valid addresses are defined recursively as follows:

1. The function MPI_ADDRESS returns a valid address, when passed as argument a variable of the calling program.
2. The buf argument of a communication function evaluates to a valid address, when passed as argument a variable of the calling program.
3. If v is a valid address, and i is an integer, then v+i is a valid address, provided v and v+i are in the same sequential storage.
4. If v is a valid address then MPI_BOTTOM + v is a valid address.

### 3.12.7 Examples

The following examples illustrate the use of derived datatypes.

**Example 3.30 Send and receive a section of a 3D array.**

```fortran
REAL a(100,100,100), e(9,9,9)
INTEGER oneslice, twoslice, threeslice, sizeofreal, myrank, ierr
INTEGER status(MPI_STATUS_SIZE)

C extract the section a(1:17:2, 3:11, 2:10)
C and store it in e(:, :, :).
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)
CALL MPI_TYPE_EXTENT(MPI_Real, sizeofreal, ierr)
C create datatype for a 1D section
CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice, ierr)
CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, twoslice, ierr)
CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, threeslice, ierr)
C create data type for a 3D slice
CALL MPI_TYPE_VECTOR( 9, 2, 2, MPI_REAL, e, ierr)
C create array with the desired layout
allocate(dimension(oneslice, twoslice, threeslice))
C send a slice of array a
CALL MPI_SEND(a, 2, 0, 0, 0, comm, status, ierr)
C receive the slice of array a
CALL MPI_RECV(d, 2, 0, 0, 0, comm, status, ierr)
```

A correct program uses only valid addresses to identify the locations of entries in communication buffers. Furthermore, if u and v are two valid addresses, then the (integer) difference u - v can be computed only if both u and v are in the same sequential storage. No other arithmetic operations can be meaningfully executed on addresses.

The rules above impose no constraints on the use of derived datatypes, as long as they are used to define a communication buffer that is wholly contained within the same sequential storage. However, the construction of a communication buffer that contains variables that are not within the same sequential storage must obey certain restrictions. Basically, a communication buffer with variables that are not within the same sequential storage can be used only by specifying in the communication call buf = MPI_BOTTOM, count = 1, and using a datatype argument where all displacements are valid (absolute) addresses.

**Advice to users.** It is not expected that MPI implementations will be able to detect erroneous, "out of bound" displacements — unless those overflow the user address space — since the MPI call may not know the extent of the arrays and records in the host program. (End of advice to users.)

**Advice to implementors.** There is no need to distinguish (absolute) addresses and (relative) displacements on a machine with contiguous address space: MPI_BOTTOM is zero, and both addresses and displacements are integers. On machines where the distinction is required, addresses are recognized as expressions that involve MPI_BOTTOM. (End of advice to implementors.)

Note that in Fortran, Fortran INTEGERs may be too small to contain an address (e.g., 32 bit INTEGERs on a machine with 64 bit pointers). Because of this, in Fortran implementations may restrict the use of absolute addresses to only part of the process memory, and restrict the use of relative displacements to subranges of the process memory where they are constrained by the size of Fortran INTEGERs.
C create datatype for a 2D section
CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr)

C create datatype for the entire section
CALL MPI_TYPE_HVECTOR(9, 1, 100*100*sizeofreal, twoslice, 1, threeslice, ierr)
CALL MPI_TYPE_COMMIT(threeslice, ierr)

CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9, MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)

Example 3.31 Copy the (strictly) lower triangular part of a matrix.

REAL a(100,100), b(100,100)
INTEGER disp(100), blocklen(100), ltype, myrank, ierr
INTEGER status(MPI_STATUS_SIZE)

C copy lower triangular part of array a
onto lower triangular part of array b
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

C compute start and size of each column
DO i=1, 100
  disp(i) = 100*(i-1) + i
  block(i) = 100-i
END DO

C create datatype for lower triangular part
CALL MPI_TYPE_INDEXED(100, block, disp, MPI_REAL, ltype, ierr)
CALL MPI_TYPE_COMMIT(ltype, ierr)

CALL MPI_SENDRECV(a, 1, ltype, myrank, 0, b, 1, ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)

Example 3.32 Transpose a matrix.

REAL a(100,100), b(100,100)
INTEGER row, xpose, sizeofreal, myrank, ierr
INTEGER status(MPI_STATUS_SIZE)

C transpose a matrix a onto b
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)

C create datatype for one row
CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
CALL MPI_TYPE_HVECTOR(100, 1, sizeofreal, row, xpose, ierr)
CALL MPI_TYPE_COMMIT(xpose, ierr)

CALL MPI_SENDRECV(a, 1, xpose, myrank, 0, b, 100*100, MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)

Example 3.33 Another approach to the transpose problem:

REAL a(100,100), b(100,100)
INTEGER disp(2), blocklen(2), type(2), row, row1, sizeofreal
INTEGER myrank, ierr
INTEGER status(MPI_STATUS_SIZE)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

C transpose matrix a onto b
CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)

C create datatype for one row, with the extent of one real number
disp(1) = 0
disp(2) = sizeofreal

blocklen(1) = 1
blocklen(2) = 1

CALL MPI_TYPE_STRUCT(2, blocklen, disp, type, row1, ierr)
CALL MPI_TYPE_COMMIT(row1, ierr)

CALL MPI_SENDRECV(a, 100, row1, myrank, 0, b, 100*100, MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)

Example 3.34 We manipulate an array of structures.

struct Partstruct{
  int class; /* particle class */
  double d[6]; /* particle coordinates */
  char b[7]; /* some additional information */
}
CHAPTER 3. POINT-TO-POINT COMMUNICATION

struct Particle particle[1000];
int i, dest, rank;
MPI_Comm comm;

/* build datatype describing structure */
MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int blocklen[3] = {1, 6, 7};
MPI_Aint disp[3];
int base;

/* compute displacements of structure components */
MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
base = disp[0];
for (i=0; i < 3; i++) disp[i] -= base;

MPI_Type_struct( 3, blocklen, disp, type, &Particletype);

MPI_Datatype type1[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_UB};
int blocklen1[4] = {1, 6, 7, 1};
MPI_Aint disp1[4];
int base;

/* compute displacements of structure components */
MPI_Address( particle, disp1);
MPI_Address( particle[0].d, disp1+1);
MPI_Address( particle[0].b, disp1+2);
MPI_Address( particle+1, disp1+3);
base = disp1[0];
for (i=0; i < 4; i++) disp1[i] -= base;

/* build datatype describing structure */
MPI_Type_struct( 4, blocklen1, disp1, type1, &Particletype);

MPI_Type_commit( &Particletype);
MPI_Send( particle, 1000, Particletype, dest, tag, comm);

MPI_Datatype Zparticles; /* datatype describing all particles with class zero (needs to be recomputed if classes change) */
MPI_Datatype Ztype;
MPI_Aint zdisp[1000];
int zblock[1000], j, k;
int zzblock[2] = {1,1};
MPI_Aint zzdisp[2];
MPI_Datatype zztype[2];

/* compute displacements of class zero particles */
j=0;
for(i=0; i < 1000; i++)
if (particle[i].class==0)
{
  zdisp[j] = i;
  zblock[j] = 1;
  j++;
}

/* create datatype for class zero particles */
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);

/* prepend particle count */
MPI_Address(&j, zzdisp);
MPI_Address(particle, zzdisp+1);

MPI_Type_struct( 2, zzblock, zzdisp, Particletype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);

/* A probably more efficient way of defining Zparticles */
MPI_Datatype zdisps[1000];
int zblocks[1000], j, k;
int zzblocks[2] = {1,1};
MPI_Address( particle, zdisps);
MPI_Address( particle[0].d, zdisps+1);
MPI_Address( particle[0].b, zdisps+2);
MPI_Address( particle[0].d, zdisps+3);
base = disp1[0];
for (i=0; i < 4; i++) disp1[i] -= base;

/* build datatype describing structure */
MPI_Type_struct( 4, blocklen1, disp1, type1, &Particletype);
/* consecutive particles with index zero are handled as one block */

j=0;
for (i=0; i < 1000; i++)
  if (particle[i].index==0)
    for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++)
      zdisp[j] = i;
      zblock[j] = k-i;
      j++;
      i=k;
}

MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);

MPI_Datatype Allpairs; /* datatype for all pairs of coordinates */

MPI_Type_extent( Particletype, &sizeofentry);
  /* sizeofentry can also be computed by subtracting the address 
   of particle[0] from the address of particle[1] */

MPI_Type_hvector( 1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);

MPI_Type_commit( &Allpairs);

MPI_Send( particle[0].d, 1, Allpairs, dest, tag, comm);  

Example 3.35 The same manipulations as in the previous example, but use absolute addresses in datatypes.

struct Partstruct
  { int class;
    double d[6];
    char b[7];
  };

struct Partstruct particle[1000];  
   /* build datatype describing first array entry */

MPI_Datatype Particletype;

MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};

int block[3] = {1, 6, 7};

MPI_Aint disp[3];

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);

MPI_Type_struct( 3, block, disp, type, &Particletype);

/* Particletype describes first array entry -- using absolute addresses */

MPI_Type_commit( &Particletype);

MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);

MPI_Datatype Zparticles, Ztype;

int zdisp[1000]
int zblock[1000], i, j, k;

struct Partstruct particle[1000];
   /* build datatype describing first array entry */

MPI_Datatype Particletype;

MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};

int block[3] = {1, 6, 7};

MPI_Aint disp[3];

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);

MPI_Type_struct( 3, block, disp, type, &Particletype);

/* Particletype describes first array entry -- using absolute addresses */

MPI_Type_commit( &Particletype);

MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);

MPI_Datatype Zparticles, Ztype;

int zdisp[1000]
int zblock[1000], i, j, k;

MPI_Address( particle, zdisp);
MPI_Address( particle[0].d, zdisp+1);
MPI_Address( particle[0].b, zdisp+2);

MPI_Type_struct( 3, block, disp, type, &Particletype);

/* Particletype describes first array entry -- using absolute addresses */

MPI_Type_commit( &Particletype);

MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
if (particle[i].index==0)
{
    for (k=i+1; (k < 1000)&&(particle[k].index = 0) ; k++);
    zdisp[j] = i;
    zblock[j] = k-i;
    j++;
    i = k;
}
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
/* Zparticles describe particles with class zero, using
    their absolute addresses*/
/* prepend particle count */
MPI_Address(&j, zzdisp);
zzdisp[1] = MPI_BOTTOM;
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);
MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);

Example 3.36 Handling of unions.

union {
    int ival;
    float fval;
} u[1000]

int utype;

/* All entries of u have identical type; variable
    utype keeps track of their current type */
MPI_Datatype type[2];
int  blocklen[2] = {1,1};
MPI_Datatype mpi_utype[2];
MPI_Aint i,j;

/* compute an MPI datatype for each possible union type;
    assume values are left-aligned in union storage. */
MPI_Address( u, &i);
MPI_Address( u+1, &j);
disp[0] = 0; disp[1] = j-1;
type[1] = MPI_UB;

3.13. PACK AND UNPACK

Some existing communication libraries provide pack/unpack functions for sending noncon-
tiguous data. In these, the user explicitly packs data into a contiguous buffer before sending
it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are
described in Section 3.12, allow one, in most cases, to avoid explicit packing and unpacking.
The user specifies the layout of the data to be sent or received, and the communication
library directly accesses a noncontiguous buffer. The pack/unpack routines are provided
for compatibility with previous libraries. Also, they provide some functionality that is not
otherwise available in MPI. For instance, a message can be received in several parts, where
the receive operation done on a later part may depend on the content of a former part.
Another use is that outgoing messages may be explicitly buffered in user supplied space,
thus overriding the system buffering policy. Finally, the availability of pack and unpack
operations facilitates the development of additional communication libraries layered on top
of MPI.

MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)

IN inbuf input buffer start (choice)
IN incount number of input data items (integer)
IN datatype datatype of each input data item (handle)
OUT outbuf output buffer start (choice)
OUT outsize output buffer size, in bytes (integer)
INOUT position current position in buffer, in bytes (integer)
IN comm communicator for packed message (handle)

int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
int outsize, int *position, MPI_Comm comm)

MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
<TYPE> INBUF(*), OUTBUF(*)
INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer space specified by outbuf and outcount. The input buffer can be any communication buffer allowed in MPI_SEND. The output buffer is a contiguous storage area containing outsize bytes, starting at the address outbuf (length is counted in bytes, not elements, as if it were a communication buffer for a message of type MPI_PACKED). The input value of position is the first location in the output buffer to be used for packing. position is incremented by the size of the packed message, and the output value of position is the first location in the output buffer following the locations occupied by the packed message. The comm argument is the communicator that will be subsequently used for sending the packed message.

```
MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)
```

<table>
<thead>
<tr>
<th>IN</th>
<th>inbuf</th>
<th>input buffer start (choice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>insize</td>
<td>size of input buffer, in bytes (integer)</td>
</tr>
<tr>
<td>INOUT</td>
<td>position</td>
<td>current position in bytes (integer)</td>
</tr>
<tr>
<td>OUT</td>
<td>outbuf</td>
<td>output buffer start (choice)</td>
</tr>
<tr>
<td>IN</td>
<td>outcount</td>
<td>number of items to be unpacked (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
<td>datatype of each output data item (handle)</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator for packed message (handle)</td>
</tr>
</tbody>
</table>

```
int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from the buffer space specified by inbuf and insize. The output buffer can be any communication buffer allowed in MPI_RECV. The input buffer is a contiguous storage area containing insize bytes, starting at address inbuf. The input value of position is the first location in the input buffer occupied by the packed message. position is incremented by the size of the packed message, so that the output value of position is the first location in the input buffer after the locations occupied by the message that was unpacked. comm is the communicator used to receive the packed message.

```
MPI_RECV(inbuf, incount, position, outbuf, outcount, datatype, comm)
```

<table>
<thead>
<tr>
<th>IN</th>
<th>inbuf</th>
<th>input buffer start (choice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>incount</td>
<td>number of items to be sent (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>inposition</td>
<td>current position in bytes (integer)</td>
</tr>
<tr>
<td>OUT</td>
<td>outbuf</td>
<td>output buffer start (choice)</td>
</tr>
<tr>
<td>IN</td>
<td>outcount</td>
<td>number of items to be received (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
<td>datatype of each input data item (handle)</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator for packed message (handle)</td>
</tr>
</tbody>
</table>

```
int MPI_RECV(void* inbuf, int incount, int *inposition, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

Advice to users. Note the difference between MPI_RECV and MPI_UNPACK. in MPI_RECV, the count argument specifies the maximum number of items that can be received. The actual number of items received is determined by the length of the incoming message. In MPI_UNPACK, the count argument specifies the actual number of items that are unpacked; the “size” of the corresponding message is the increment in position. The reason for this change is that the “incoming message size” is not predetermined since the user decides how much to unpack; nor is it easy to determine the “message size” from the number of items to be unpacked. In fact, in a heterogeneous system, this number may not be determined a priori. (End of advice to users.)

To understand the behavior of pack and unpack, it is convenient to think of the data part of a message as being the sequence obtained by concatenating the successive values sent in that message. The pack operation stores this sequence in the buffer space, as if sending the message to that buffer. The unpack operation retrieves this sequence from buffer space, as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or `sscanf` in C, for a similar function.)

Several messages can be successively packed into one packing unit. This is effected by several successive related calls to MPI_PACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for outbuf, outcount and comm. This packing unit now contains the equivalent information that would have been stored in a message by one send call with a send buffer that is the “concatenation” of the individual send buffers.

A packing unit can be sent using type MPI_PACKED. Any point to point or collective communication function can be used to move the sequence of bytes that forms the packing unit from one process to another. This packing unit can now be received using any receive operation, with any datatype: the type matching rules are relaxed for messages sent with type MPI_PACKED.

A message sent with any type (including MPI_PACKED) can be received using the type MPI_PACKED. Such a message can then be unpacked by calls to MPI_UNPACK.

A packing unit (or a message created by a regular, “typed” send) can be unpacked into several successive messages. This is effected by several successive related calls to MPI_UNPACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for inbuf, incount and comm.

The concatenation of two packing units is not necessarily a packing unit; nor is a substring of a packing unit necessarily a packing unit. Thus, one cannot concatenate two packing units and then unpack the result as one packing unit; nor can one unpack a substring of a packing unit as a separate packing unit. Each packing unit, that was created by a related sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of related unpack calls.

Rationale. The restriction on “atomic” packing and unpacking of packing units allows the implementation to add at the head of packing units additional information, such as a description of the sender architecture (to be used for type conversion, in a heterogeneous environment) (End of rationale.)

The following call allows the user to find out how much space is needed to pack a message and, thus, manage space allocation for buffers.
CHAPTER 3. POINT-TO-POINT COMMUNICATION

**MPI Pack**

**MPI Pack** Size (Incount, Datatype, Comm, Size)

- **IN** incount: count argument to packing call (integer)
- **IN** datatype: datatype argument to packing call (handle)
- **IN** comm: communicator argument to packing call (handle)
- **OUT** size: upper bound on size of packed message, in bytes (integer)

```c
int MPI_Pack(size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)
```

**MPI Pack** Size (Incount, Datatype, Comm, Size, IERROR)

- **IN** incount: upper bound on size of packed message, in bytes (integer)
- **IN** datatype: upper bound on size of packed message, in bytes (integer)
- **IN** comm: communicator argument to packing call (handle)
- **OUT** size: upper bound on size of packed message, in bytes (integer)
- **OUT** IERROR: error status

**Example 3.37** An example using **MPI Pack**.

```c
int position, i, j, a[2];
char buff[1000];

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
    { /* SENDER CODE */
        position = 0;
        MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
        MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
        MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
    }
else /* RECEIVER CODE */
    { /* RECEIVE CODE */
        MPI_Recv(a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
```

**Example 3.38** An elaborate example.

```c
int position, i;
float a[1000];
char buff[1000];

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
    { /* SENDER CODE */
        position = 0;
        MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
        MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
        MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
    }
else /* RECEIVER CODE */
    { /* RECEIVE CODE */
        MPI_Status status;
        MPI_Recv(a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
```

**Rationale.** The call returns an upper bound, rather than an exact bound, since the exact amount of space needed to pack the message may depend on the context (e.g., first message packed in a packing unit may take more space). (End of rationale.)

**Example 3.37** An example using **MPI Pack**.

```c
int position, i, j, a[2];
char buff[1000];

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
    { /* SENDER CODE */
        position = 0;
        MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
        MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
        MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
    }
else /* RECEIVER CODE */
    { /* RECEIVE CODE */
        MPI_Status status;
        MPI_Recv(a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
```

**Example 3.38** An elaborate example.

```c
int position, i, ;
float a[1000];
char buff[1000];

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
    { /* SENDER CODE */
        position = 0;
        MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
        MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
        MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
    }
else /* RECEIVER CODE */
    { /* RECEIVE CODE */
        MPI_Status status;
        MPI_Recv(a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
```
/* Unpack i */
position = 0;
MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);

/* Unpack a[0]...a[i-1] */
MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);

Example 3.39 Each process sends a count, followed by count characters to the root; the root concatenates all characters into one string.

```c
int count, gsize, counts[64], totalcount, k1, k2, k,
displs[64], position, concat_pos;
char chr[100], *lbuf, *rbuf, *cbuf;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

/* allocate local pack buffer */
MPI_Pack_size(1, MPI_INT, comm, &k1);
MPI_Pack_size(count, MPI_CHAR, &k2);
k = k1+k2;
lbuf = (char *)malloc(k);

/* pack count, followed by count characters */
position = 0;
MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
MPI_Pack(chr, count, MPI_CHAR, &lbuf, k, &position, comm);
if (myrank != root)
  /* gather at root sizes of all packed messages */
  MPI_Gather( &position, 1, MPI_INT, NULL, NULL, root, comm);
  /* gather at root packed messages */
  MPI_Gatherv( lbuf, position, MPI_PACKED, NULL, NULL, NULL, NULL, root, comm);
else {
  /* root code */
  /* gather sizes of all packed messages */
  MPI_Gather( &position, 1, MPI_INT, counts, 1, MPI_INT, root, comm);
  /* gather all packed messages */
  displs[0] = 0;
  for (i=1; i < gsize; i++)
    displs[i] = displs[i-1] + counts[i-1];
    ...
...
```

```
3.13. PACK AND UNPACK
```

```c
totalcount = displs[gsize-1] + counts[gsize-1];
rbuf = (char *)malloc(totalcount);
cbuf = (char *)malloc(totalcount);
MPI_Gatherv( lbuf, position, MPI_PACKED, rbuf,
    counts, displs, MPI_PACKED, root, comm);

/* unpack all messages and concatenate strings */
concat_pos = 0;
for (i=0; i < gsize; i++) {
  position = 0;
  MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
          &position, &count, 1, MPI_INT, comm);
  MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
          cbuf+concat_pos, count, MPI_CHAR, comm);
  concat_pos += count;
}
cbuf[concat_pos] = '\0';
```
NAME
rabsonlib – library of some of David Rabson’s functions

SYNOPSIS
cc -c -I/home/5156/rabsonlib ...
c -o ... -L/home/5156/rabsonlib -lrabson -lrabsonmalloc

#include <rabsonlib.h>
#include <rabsonlibmalloc.h>

DESCRIPTION
This library contains some of the subroutines I’ve found most useful over a long time. The most important
are in the manual pages agets(3), ascanf(3), debug(3), f_err(3), pstrcat(3), rabsonlib_malloc(3),
rabsonlib_stralloc(3), and split(3).

Some of the more minor ones I’ve kept in the library partly because my old programs still expect them.
Many of these are prefixed with "rabsonlib_" in an attempt to reduce name conflicts. They are flushin(3),
isaterm(3), rabsonlib_prefix(3), rabsonlib_sleftshift(3), rabsonlib_tolower(3), rabsonlib_toupper(3),
and stripnl(3).

Still to do: update and add to the package my old gmenu package.
NAME
aget – get an arbitrary-length string from a stream, allocating memory

SYNOPSIS
cc -c -I/home/5156/rabsonlib ...
cc -o ... -L/home/5156/rabsonlib -lrabson

#include <stdio.h>
#include <rabsonlib.h>

char *agets(FILE *stream);

DESCRIPTION
This subroutine reads from the given stream until it finds a newline, embedded null, or end-of-file, then returns the string read, allocating sufficient memory with malloc(3). This allocation eliminates the need for the user to provide a fixed-length buffer, as in fgets(3). To avoid memory leakage, the user should call free(3) to free the string returned when it is no longer needed. See notes below.

RETURN VALUES
If the string read ends with a newline, this character is the last character in the string returned. If it ends with an embedded \0, the string returned will not end with a newline, but the \0 will be placed back on the input stream (see ungetc(3).) If the first character read is a \0, an empty (but not NULL) string will be returned. If the file ends without a newline, no newline is appended. If no characters have been read, but the end of the file has been reached, the null pointer is returned.

NOTES ON MALLOC AND FREE
The version of agets(3) in the library -lrabson calls malloc(3) and free(3) as described above. The version in -lrabsonmalloc calls the wrapper functions. If a user program calls the wrappers, it is important to link to -lrabsonmalloc before -lrabson.

SEE ALSO
fgets(3) ungetc(3) malloc(3) free(3)
NAME
ascanf — argument-line scanner similar in syntax to scanf

SYNOPSIS
cc -c -I/home/5156/rabsonlib ...
cc -o ... -L/home/5156/rabsonlib -lrabson

#include <rabsonlib.h>

int ascanf(argc, argv, format_line, address1, ....)
char **argv, *format_line;
TYPE *address1,....;

DESCRIPTION
This is a medium-duty, easy-to-use argument-line parser for C programs: the idea is to avoid a huge and ugly switch/case construct in the main() function. ascanf (3) understands most UNIX argument-line conventions and makes it easy to extract numeric and string arguments and flags.

SUMMARY
A detailed description follows this summary.

%d, %s, etc. Stuff the corresponding pointer with the given object.
#
Return the address of a string; same precedence as %s.
-c|num
If argument -c is specified, OR the return value with num.
-c&num
If argument -c is specified, AND the return value with num.
-cF
Expect "-c value", "-c=value", "-c:value", or "-c:value", where F is a scanf format or #. |num or &num may be combined with this. Multiple formats are allowed. For example, "-c1%d%s" will read arguments "-c 42 foo" or "-c=17 foo" and set the 1 flag.
E%s%s If any errors are encountered in parsing argv, do not print the error messages on stderr; instead, stuff the first corresponding string with a description of the error (which will include a single ‘%s’) and the second string with the offending argument.
+
As FIRST character ONLY: do not complain about arguments in argv that match no formats. Instead, set all elements of argv that were read normally to NULL, and leave only those that matched no formats as they were.
=num
Set the default flag value to num; this is useful in conjunction with & flags, which can unset bits in the flag.

MORE DETAILED DESCRIPTION
argc and argv are the usual arguments of main(); format_line is a string similar to the format string of scanf (3); I describe it more fully below. The remaining arguments are addresses to be stuffed with arguments that have been read.

The format string consists of a sequence of modified scanf formats separated by single spaces. The most common format string is %C where C is one of s, d, c, f, lf, x, or o specifying the type of argument expected in the command line. As with scanf (3), any argument matching the n th such format will be stuffed into the n th user-supplied variable, the addresses of which are specified by address1 , ... above. Example:

ascanf(argc, argv, "%s %d", string, &i);

If a given word in the argument line (specified by argv) could match more than one of the formats in format_line, ascanf chooses the least general: for example, %o is less general than %d, so if both are specified in format_line, a single argument of ’7’ will match %o. %s is the most general format type, so a given argument will match it only if it fails to match anything else. Format modifiers, e.g., %2.2s, are supported. Since %s is not always convenient, ascanf (3) also supports a new format, # (pound sign). Pound
sign matches a string just as does %s, but the corresponding address must be a (char **) to which it points will be stuffed with the address of any matching string in argv.

ascanf (3) provides a very simple way to find out what flags have been specified. Normally, the programmer will associate with each legal flag an arbitrary power of two. In the format_line, he or she will then specify flags in the form "-C|num", where C is the flag expected and num is the associated value. ascanf() will return the bitwise OR of all flags that match. It should be noted that ascanf() returns -1 if there was an error, in which case the flags are lost. Example:

```
#define CFLAG 0x1
#define HFLAG 0x4
sprintf(format, "-c|\d -h|\d", CFLAG, HFLAG);
if((flags=ascanf(argc, argv, format))==1)
  (error)
```

As indicated in the short summary above, it is also possible to AND values when flags are found. One may also specify that a flag has some value:

```
ascanf(argc, argv, ",-g%|4", string);
```

The above example looks for a -g flag with a value, which may be specified in the argument line as "-g value", "-g:value", "-g=value", or "-g=value". The \|4 above is optional, and the |num and value parts of the string format may come in either order.

If there is a minus sign (" - ") sitting by itself in the argument line given by argv, it is not read, but the next argument will not be interpreted as a switch, even if it begins with a minus sign. Note that a number beginning with a minus sign (e.g., -3.14159) will be interpreted as a number, not a switch, so long as there are no numeric switches (e.g., -9) in the format. If there are numeric switches, the user must supply the minus sign (" - ") sitting by itself before a negative number she wishes not to be interpreted as a switch.

The summary section above shows how to override ascanf's normal method of dealing with errors.

**ORIGINALLY WRITTEN 1/88**

**BUGS**

Not all argument lines can be parsed with ascanf, and while it is a big improvement over continually-growing switch/case’s or if/else’s, it is not as neat as the table-run-function approach, which could also give the user more specific and useful error messages. The latter, however, requires more work than most programmers are willing to put into a short program, for which ascanf is more appropriate.

There is currently no provision for a switch that both ands the flag with some number and ors it with some other, e.g., -x&0xffbf|0x1000.
NAME
debug – macros for printing debugging output

SYNOPSIS
cc -c -I/home/5156/rabsonlib ...
cc -o ... -L/home/5156/rabsonlib -lrabson

#include <stdio.h>
#include <rabsonlib.h>

DEBUG((int level, char ∗fmt, <printf(3) arguments>));
DEBUG_((int level, char ∗fmt, <printf(3) arguments>));
void _DEBUG(int level, char ∗fmt, <printf(3) arguments>);
void debugoff(char ∗x);
void debugstop(int i);
void debugoutput(FILE ∗f);

extern int debuglevel=0;

DESCRIPTION
The external int debuglevel controls whether to print debugging output. If level in a call to _DEBUG is greater than or equal to debuglevel, then the remaining arguments are passed to printf(3). The macro DEBUG is convenient, since it may be turned off at compile time with the flag -DDEBUG=NOP. Note the use of double parentheses with DEBUG (see example below) and with DEBUG_, which differs only in not being turned off by the compilation flag -DDEBUG=NOP.

debugoff is used to identify and selectively to turn off debugging messages. With argument x=0, it toggles the printing of a hexadecimal string identifier for each message. If x is non-zero, it is the address of a string as printed when identifiers are printed. Subsequent calls to DEBUG_ will not print.

debugstop with non-zero argument i sets _DEBUG to pause and prompt and wait for a line from stdin every i debugging messages.

debugoutput(f) sends all subsequent output of _DEBUG to file f instead of to stdout.

EXAMPLE
#include <stdlib.h>
#include <stdio.h>
#include <rabsonlib.h>

int
main(int argc, char ∗∗argv)
{
    if(argc>=2)
        debuglevel = atoi(argv[1]);/* set debuglevel */
    DEBUG((4, "This prints only if debuglevel>=4.0"));
    return 0;
}
NAME
f_err – routines for printing error messages

SYNOPSIS
cc -c -I/home/5156/rabsonlib ... 
cc -o ... -L/home/5156/rabsonlib -lrabson

#include <rabsonlib.h>

void {x}{y}_err(char *fmt, <printf(3) arguments);

extern void (*on_f_err)(void);
extern void (*on_w_err)(void);
extern void (*on_done)(void);

DESCRIPTION
The routines f_err(), w_err(), fp_err(), and wp_err() print error messages. If <x> is 'f', the error is considered fatal, and the routine calls exit(3) with argument 1. If <x> is 'w', the routine instead returns. If <y> is 'p', perror(3) is called to print the message.

The hook on_f_err, if set, is called on every call to f_err() or fp_err(). on_w_err, if set, is called on every call to w_err() or wp_err(). on_done, if set, is called on every call to f_err(), fp_err(), w_err(), or wp_err().

EXAMPLE
#include <stdio.h>
#include <rabsonlib.h>

int main(int argc, char **argv)
{
    char name[]="nosuch.file";
    FILE *f = fopen(name, "r");
    if(!f)
        wp_err(name);
    return 0;
}
NAME
    flushin – flush standard input

SYNOPSIS
    cc -c -I/home/5156/rabsonlib ...
    cc -o ... -L/home/5156/rabsonlib -lrabson

    #include <rabsonlib.h>

    int flushin(void);

DESCRIPTION
    This flushes standard input by repeatedly reading from stdin until end of file, then calling tcflush(3), whose return value it passes on.

SEE ALSO
    tcflush(3)
NAME
  isaterm – does file pointer refer to a terminal

SYNOPSIS
  cc -c -I/home/5156/rabsonlib ...
  cc -o ... -L/home/5156/rabsonlib -lrabson

    #include <stdio.h>
    #include <rabsonlib.h>

    int isaterm(FILE *fp);

DESCRIPTION
  Returns 1 if fp points to a terminal, 0 otherwise. This is a wrapper for isatty(3), which takes a file descriptor rather than a file pointer.

SEE ALSO
  isatty(3), fileno(3).
NAME
pstrcat, pstrncat – concatenate strings, returning pointer to end

SYNOPSIS
cc -c -I/home/5156/rabsonlib ...
cc -o ... -L/home/5156/rabsonlib -lrabson

#include <rabsonlib.h>

char *pstrcat(char *dest, char *src);
char *pstrncat(char *dest, char *src, int n);

DESCRIPTION
These functions serve the same purposes as strcat(3) and strncat(3) but return a pointer to the end rather than the beginning of the destination string, dest.

SEE ALSO
strcat(3)  strncat(3)
NAME
rabsonlib_malloc – debugging wrappers for malloc, calloc, realloc, free

SYNOPSIS
cc -Duserabsonlibmalloc -c -I/home/5156/rabsonlib ...  
cc -o ... -L/home/5156/rabsonlib -lrabsonmalloc

#include <rabsonlibmalloc.h>

void *malloc(size_t s);
void *calloc(int n, size_t s);
void *realloc(void *x, size_t s);
void free(void *x);
int rabsonlib_malloc_count(void);
int rabsonlib_malloc_total(void);
int rabsonlib_malloc_freed(void);

DESCRIPTION
This package provides debugging wrappers for the standard memory-management library. If debuglevel
(see debug(3)) is set to 20 or higher, messages will be written to stdout that can be analyzed with a script.
(I have one called debmal; if stdout is inconvenient, see debug(3) for redirecting the debugging messages
to stderr.) Such a script can make sure that every block free’d was previously malloc’ed, and not previ-
ously free’d, (to detect free(3) errors) or that every block malloc’ed is eventually free’d (to detect memory
leaks). Some additional information is printed at debuglevel 21.

There are three ways to to use the wrappers.

1. The include file <rabsonlibmalloc.h>, coupled with the directive -Duserabsonlibmalloc (as in the
synopsis above), replaces malloc(3), calloc(3), realloc(3), and free(3) with the wrapper functions
residing in the library linked with -lrabsonmalloc. The standard library functions are called instead
if -Duserabsonlibmalloc is NOT in place. It is important that a program should not mix calls to the
wrappers and the standard library for allocation and free(3) calls. This can be resolved by recompil-
ing all modules whenever changing the compilation options.

2. If using the GNU version of ld(1), link with the flags
--wrap malloc --wrap free --wrap calloc --wrap realloc
or, if calling ld(3) through gec(1),
-Xlinker --wrap -Xlinker malloc -Xlinker --wrap -Xlinker free -Xlinker --wrap -Xlinker calloc
-Xlinker --wrap -Xlinker realloc

Be sure to link with -lrabsonmalloc before -lrabson so that calls to agets(3) and other rabsonlib func-
tions go to the versions also using the wrappers.

3. With the include file and library of method 1, but without -Duserabsonlibmalloc, one can still call the
wrapper functions explicitly as __wrap_malloc(), __wrap_free(), __wrap_calloc, and
__wrap_realloc().
NAME
rabsonlib_prefix – is a string a prefix of another string

SYNOPSIS
cc -c -I/home/5156/rabsonlib ...
c -o ... -L/home/5156/rabsonlib -lrabson

#include <rabsonlib.h>

char *rabsonlib_prefix(char *name, char *target);

DESCRIPTION
Returns 0 if name is not a prefix to target, 1 if name is identical to target (an exact match), and 2 if name is a prefix but not an exact match.

SEE ALSO
strncmp(3)
NAME
rabsonlib_sleftshift – shift a string to the left

SYNOPSIS
cc -c -I/home/5156/rabsonlib ...
cc -o ... -L/home/5156/rabsonlib -lrabson

#include <rabsonlib.h>

char *rabsonlib_sleftshift(char *s, int n);

DESCRIPTION
Shift the string s to the left by n characters (must be positive). No characters are shifted to bytes before the
beginning of the string; thus, n characters are lost, and the string becomes shorter. However, if n is greater
than the string length, the call has no effect. The beginning of the string s is returned.
NAME
rabsonlib_stralloc – allocate enough room for a string, and copy it there

SYNOPSIS
cc -c -I/home/5156/rabsonlib ...
cc -o ... -L/home/5156/rabsonlib -lrabson

#include <rabsonlib.h>

char *rabsonlib_stralloc(char *s);

DESCRIPTION
Call malloc(3) to allocate enough room for a string s, copy it, and return a pointer to the copied string.

NOTES ON MALLOC AND FREE
The version of rabsonlib_stralloc(3) in the library -lrabson calls malloc(3) and free(3) as described above. The version in -lrabsonmalloc calls the wrapper functions. If a user program calls the wrappers, it is important to link to -lrabsonmalloc before -lrabson.

SEE ALSO
malloc(3), free(3), strlen(3), strcpy(3)
NAME
split, splitdelim – split a string at delimiter characters

SYNOPSIS
cc -c -I/home/5156/rabsonlib ...
cc -o ... -L/home/5156/rabsonlib -lrabson

#include <rabsonlib.h>
char **split(char *string);
char **splitdelim(char *string, enum splitenum (*f)(char));

DESCRIPTION
split() splits a string into words at white-space delimiters, returning an allocated vector of pointers to the words (null-terminated substrings). The pointer may be freed with free(3) when no longer needed. The function is destructive, writing nulls in string and possibly moving other characters around; the vector will point into the space of the original string (except see BUGS below).

A limited subset of the shell’s string-quoting ability is supported: white space may be included inside a word enclosed by a pair of double quotation marks (“...”) or escaped with a backslash (\). Double quotation marks, unless escaped with backslashes, and backslashes, unless escaped with backslashes, are deleted from the string.

splitdelim() is a more general interface to the same subroutine. The user must supply a subroutine f() that returns one of six enum values for each passed character:

- splitenum_nul for the nil character, \0,
- splitenum_ski for a ‘skippable’ delimiter (typically white space--see below),
- splitenum_del for an unskippable delimiter,
- splitenum_esc for the character-escaping character (typically backslash, \),
- splitenum_quo for the quotation character (typically either " or ’), or
- splitenum_ord for all other characters.

Leading, trailing, and intervening skippable delimiters are excluded from the strings returned; multiple intervening skippable delimiters are counted as a single delimiter. This is the usual interpretation of white space, as in split() above. In contrast, each non-skippable delimiter is returned as an individual word.

NOTES ON MALLOC AND FREE
The version of split(3) in the library -lrabson calls malloc(3) and free(3) as described above. The version in -lrabsonmalloc calls the wrapper functions. If a user program calls the wrappers, it is important to link to -lrabsonmalloc before -lrabsonlib.

BUGS
Since the character immediately after a non-skippable delimiter cannot necessarily be replaced by a nil in the source string, a two-byte string must be allocated for each non-skippable delimiter. There are probably many more bugs; I need to re-write the code from scratch instead of re-using the very old base.

SEE ALSO
strtok(3) strtok_r(3) malloc(3) free(3)
NAME
stripnl -- strip a final newline from a string

SYNOPSIS
cc -c -I/home/5156/rabsonlib ...
cc -o ... -L/home/5156/rabsonlib -lrabson

#include <rabsonlib.h>

char *stripnl(char *s);

DESCRIPTION
If the last character of a string is a newline ('\n'), strip it. Return the argument.
Parallel Processing at USF
fall 2006

The first parallel computing cluster at the University of South Florida was set up for this course in 1999. Named “Igor” by Mr. Daniel Majchrzak, then Arts-and-Sciences system manager, it consisted originally of 16 connected 133-MHz Intel Pentium processors, each with 48 MB of RAM. We had salvaged the nodes from parts of different machines that were being thrown out; consequently, the cluster was not very reliable. Igor was replaced in fall 2000 with the College of Arts and Sciences Instructional Automated Computer, or CASIAC, running 20 nodes with more robust, but still out-of-date, hardware.

Having gained this experience, Mr. Majchrzak moved to USF Academic Computing as head of the new research-computing division, where he has set up several clusters for scientific computing. The current general-use cluster, I.R.C.E. (Instructional and Research Computing Environment), replaces the previous Mimir cluster and comprises (as of August 2006) 24 two-CPU 64-bit Opteron nodes running at 2.2 GHz, 12 two-CPU 32-bit Xeon nodes at 2.66 GHz, and several miscellaneous queues. It is controlled from a two-CPU Opteron head node, irce.acomp.usf.edu. Each subsidiary node, running the Linux 2.6 kernel, has 8 GB of RAM; the nodes are connected with a Myrinet optical network and have access to a shared high-speed /scratch filesystem with 1.3 TB of disk space.

The cluster’s fast interconnect, about 100 times faster than the 100/baseT ethernet used at USF to connect office computers, is designed to support the Message-Passing Interface, MPI, in a configuration similar to what runs on supercomputers such as the Cray XT3 and the IBM Blue Gene. Job control is provided by the Sun Grid Engine (SGE), whose user interface is close to that of NQS on Cray systems and to Load Leveler on IBM supercomputers. In addition to supporting hardware, offering private hardware management services, and managing software licensing, the USF research-computing facility includes on staff a scientific consultant, Dr. Martin Ossowski. Additional nodes, software, and consulting support are anticipated over the next few years. For detailed, current information on the facility, read the documentation at 

http://rc.usf.edu

Jobs on IRCE run for up to several days, and there are always a few on queue, as demonstrated with the qstat command (output excerpted to save paper):
The entries in state “qw” are waiting on queue and will run once the requested resources become free. As evident in the “slots” column, most of the jobs are running on multiple processors. The tools of parallel programming can be somewhat tricky to use, susceptible to bugs, finicky, and incompletely documented. In other words, the USF clusters share many of the features of a commercial supercomputer.

Each supercomputer installation has its own way of running jobs. IRCE provides scripts to hide some of the details; the general flavor is very similar to more expensive implementations. There follows an annotated example of compiling and running a trivial MPI program that simply prints out the names of the machines on which it runs.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>

int
main(int argc, char **argv)
{
  int iam, nprocs;
  char buf[128];

  if(MPI_Init(&argc, &argv)
     || MPI_Comm_rank(MPI_COMM_WORLD, &iam) /* Who am I? */
     || MPI_Comm_size(MPI_COMM_WORLD, &nprocs)) { /*# processors*/
    fprintf(stderr, "MPI initialization failed\n"); /* die on err */
    exit(1);
  }

  MPI_Barrier(MPI_COMM_WORLD);
  gethostname(buf,sizeof(buf)-1);
  printf("iam=%d, nprocs=%d, hostname=%s\n",iam, nprocs, buf);
  MPI_Finalize();
  return 0;
}
```

Many copies of this program will run in parallel; MPI calls let the copies communicate. The call to `MPI_Init()` is required to reconstitute the correct values for `argc` and `argv` on all running copies of the program. `MPI_Comm_size()` stuffs `nprocs` with the number of processes; `MPI_Comm_rank()` stuffs `iam` with the process number between 0 and `nprocs`−1. `MPI_Barrier()` does not let any process proceed until all have reached that call. The program then prints out `iam`, `nprocs`, and the name of the computer on which it is running before the required call to `MPI_Finalize()`. The pro-
gramming manual for MPI is available on-line at http://www.mpi-forum.org/docs/docs.html; version 1.1 is more readable than version 2.

The IRCE facility supports three different environments and three different compilers. A program using MPI must be compiled for the appropriate environment and then directed to one of the appropriate queues. Aliases for bash and tcsh users facilitate setting up the many compiler and linker options:

<table>
<thead>
<tr>
<th>environment</th>
<th>compiler-setup command</th>
<th>compiler</th>
<th>-pe flag</th>
<th>-q flag (optional)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit Myrinet</td>
<td>pgic</td>
<td>gcc 3.4.4</td>
<td>intel</td>
<td>mpi.mx</td>
</tr>
<tr>
<td>32-bit Ethernet</td>
<td>pgic.p4</td>
<td>gnu.p4</td>
<td>intel.p4</td>
<td>mpi.p4</td>
</tr>
<tr>
<td>64-bit shared-memory</td>
<td>pgic.shm</td>
<td>gnu.shm</td>
<td>intel.shm</td>
<td>mpi.shm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>amd64.q</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>amd64.q, ia32.q, ia32s.q</td>
</tr>
</tbody>
</table>

The 32-bit Ethernet environment is being phased out; we will be running on the 64-bit Myrinet environment. To use the commercial PGI C compiler, invoke the pgic command:

irce 2% pgic
-- Compiler and MPI Information:

- ARCH: amd64/em64t
- C/C++ Compiler: Portland Group, Ver. 6.0-5
- F77 Compiler: Portland Group, Ver. 6.0-5
- F90 Compiler: Portland Group, Ver. 6.0-5
- BLAS/LAPACK Path: /usr/local/libs/acml/pgi64/lib
  - Link with:
    -L/usr/local/libs/acml/pgi64/lib -lacml
- Best SGE queue: amd64.q with -pe mpi.mx
- MPI uses MX (Myrinet Express) for communication

-- Done.
irce 3%

To compile with MPICH, substitute the wrapper script mpicc for pgcc:

irce 3% mpicc -o ping ping.c

Attempting to run the resulting ./ping executable directly will result in an error. Instead, it is necessary to write a submission script. Lines beginning with #$ will be interpreted as SGE directives rather than comments.

irce 4% cat pingscript
#!/bin/sh
# pingscript
# DAR 8/06
# Batch script for ./ping. Use "qsub pingscript" to submit this to 4 processors. For more options, see the qsub(1) manual page.
#
# Start in the directory from which the script was submitted.
#$ -cwd
# Do not merge stderr with stdout.
#$ -j n
# Limit total CPU time to two minutes. It is important to place
# a limit on processes so that a job entering an infinite loop will
# not monopolize the cluster.
#$ -l h_cpu=00:02:00
# On start, completion, or abortion, send an e-mail.
#$ -M rabson
#$ -m abe
# Give a name to the job.
#$ -N pingscript
#$ -p e mpi.mx 4
# Use the Bourne shell to interpret the rest of this script.
#$ -S /bin/sh

# Run the job in the 64-bit Myrinet environment. The SGE will
# determine the best queue (we could have specified -q amd64.q).
# This is also where we fix the number of processors at 4.
#$ -pe mpi.mx 4
# Run the command (any arguments would follow the executable name).
sge_mpirun ping

Submit the job with the qsub command. While waiting for the job to finish, you may use
qstat to track its progress:
mimir 5% qsub pingscript
Your job 19399 ("pingscript") has been submitted.
mimir 6% qstat | grep rabson
19399 0.75000 pingscript rabson r 08/11/2006 23:43:48 amd64.q@n008 4

SGE qstat displays best in a window with at least 113 columns instead of the usual 80. The
amd64.q queue is currently the only one compatible with the mpi.mx environment

On completion, SGE creates several output files. The “o” file includes the output from all the
nodes (it also includes the full commands used by SGE to start the four processes on the remote
nodes, which I’ve skipped here with the tail command).
irce 7% ls pingscript*19399
pingscript.e19399   pingscript.o19399   pingscript.pe19399  pingscript.po19399

irce 8% tail +6 pingscript.o19399
iam=1, npacs=4, hostname=n008.acomp.usf.edu
iam=0, npacs=4, hostname=n005.acomp.usf.edu
iam=3, npacs=4, hostname=n014.acomp.usf.edu
iam=2, npacs=4, hostname=n009.acomp.usf.edu
irce 9%

Any messages written to standard error would have appeared in pingscript.e19399.

If codes are to be linked with user-supplied libraries, it is important that the libraries should
also have been compiled with the appropriate environment (32-bit or 64-bit); in my own directory
on irce, I keep separate directories, e.g., lib32 and lib64.