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

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

July 30, 2015

UCRL-SM-208112

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 Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed 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 Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

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

Approved for public release; further dissemination unlimited

Contents

List of Tables vii

List of Figures ix

1 Introduction 1
- 1.1 Changes from previous versions . . . 1
- 1.2 Reading this User Guide . . . 4

2 Mathematical Considerations 5
- 2.1 IVP solution . . . 5
- 2.2 Preconditioning . . . 9
- 2.3 Rootfinding . . . 9

3 Code Organization 11
- 3.1 SUNDIALS organization . . . 11
- 3.2 IDA organization . . . 11

4 Using IDA for C Applications 15
- 4.1 Access to library and header files . . . 15
- 4.2 Data types . . . 16
- 4.3 Header files . . . 16
- 4.4 A skeleton of the user's main program . . . 17
- 4.5 User-callable functions . . . 20
  - 4.5.1 IDA initialization and deallocation functions . . . 20
  - 4.5.2 IDA tolerance specification functions . . . 21
  - 4.5.3 Linear solver specification functions . . . 22
  - 4.5.4 Initial condition calculation function . . . 26
  - 4.5.5 Rootfinding initialization function . . . 27
  - 4.5.6 IDA solver function . . . 28
  - 4.5.7 Optional input functions . . . 29
    - 4.5.7.1 Main solver optional input functions . . . 31
    - 4.5.7.2 Dense/band direct linear solvers optional input functions . . . 35
    - 4.5.7.3 Sparse direct linear solvers optional input functions . . . 36
    - 4.5.7.4 Iterative linear solvers optional input functions . . . 38
    - 4.5.7.5 Initial condition calculation optional input functions . . . 41
    - 4.5.7.6 Rootfinding optional input functions . . . 43
  - 4.5.8 Interpolated output function . . . 44
  - 4.5.9 Optional output functions . . . 44
    - 4.5.9.1 Main solver optional output functions . . . 44
    - 4.5.9.2 Initial condition calculation optional output functions . . . 51
    - 4.5.9.3 Rootfinding optional output functions . . . 52
    - 4.5.9.4 Dense/band direct linear solvers optional output functions . . . 52
    - 4.5.9.5 Sparse direct linear solvers optional output functions . . . 54

4.5.9.6 Iterative linear solvers optional output functions . . . 55
4.5.10 IDA reinitialization function . . . 57
4.6 User-supplied functions . . . 58
4.6.1 Residual function . . . 58
4.6.2 Error message handler function . . . 59
4.6.3 Error weight function . . . 59
4.6.4 Rootfinding function . . . 60
4.6.5 Jacobian information (direct method with dense Jacobian) . . . 60
4.6.6 Jacobian information (direct method with banded Jacobian) . . . 61
4.6.7 Jacobian information (direct method with sparse Jacobian) . . . 63
4.6.8 Jacobian information (matrix-vector product) . . . 64
4.6.9 Preconditioning (linear system solution) . . . 64
4.6.10 Preconditioning (Jacobian data) . . . 65
4.7 A parallel band-block-diagonal preconditioner module . . . 66

5 FIDA, an Interface Module for FORTRAN Applications 73
5.1 Important note on portability . . . 73
5.2 Fortran Data Types . . . 73
5.3 FIDA routines . . . 74
5.4 Usage of the FIDA interface module . . . 75
5.5 FIDA optional input and output . . . 84
5.6 Usage of the FIDAROOT interface to rootfinding . . . 87
5.7 Usage of the FIDABBD interface to IDABBDPRE . . . 88

6 Description of the NVECTOR module 91
6.1 The NVECTOR_SERIAL implementation . . . 95
6.2 The NVECTOR_PARALLEL implementation . . . 97
6.3 The NVECTOR_OPENMP implementation . . . 99
6.4 The NVECTOR_PTHREADS implementation . . . 101
6.5 NVECTOR Examples . . . 103
6.6 NVECTOR functions used by IDA . . . 105

7 Providing Alternate Linear Solver Modules 107
7.1 Initialization function . . . 108
7.2 Setup function . . . 108
7.3 Solve function . . . 109
7.4 Performance monitoring function . . . 109
7.5 Memory deallocation function . . . 110

8 General Use Linear Solver Components in SUNDIALS 111
8.1 The DLS modules: DENSE and BAND . . . 112
8.1.1 Type DlsMat . . . 112
8.1.2 Accessor macros for the DLS modules . . . 115
8.1.3 Functions in the DENSE module . . . 115
8.1.4 Functions in the BAND module . . . 118
8.2 The SLS module . . . 119
8.2.1 Type SlsMat . . . 120
8.2.2 Functions in the SLS module . . . 121
8.2.3 The KLU solver . . . 123
8.2.4 The SUPERLUMT solver . . . 123
8.3 The SPILS modules: SPGMR, SPFGMR, SPBCG, and SPTFQMR . . . 123
8.3.1 The SPGMR module . . . 124
8.3.2 The SPFGMR module . . . 125
8.3.3 The SPBCG module . . . 125
8.3.4 The SPTFQMR module . . . 125

A SUNDIALS Package Installation Procedure **127**
A.1 CMake-based installation . . . 128
A.1.1 Configuring, building, and installing on Unix-like systems . . . 128
A.1.2 Configuration options (Unix/Linux) . . . 130
A.1.3 Configuration examples . . . 133
A.1.4 Working with external Libraries . . . 133
A.2 Building and Running Examples . . . 134
A.3 Configuring, building, and installing on Windows . . . 134
A.4 Installed libraries and exported header files . . . 135

B IDA Constants **139**
B.1 IDA input constants . . . 139
B.2 IDA output constants . . . 139

Bibliography **143**

Index **145**

List of Tables

4.1 Optional inputs for IDA, IDADLS, IDASLS, and IDASPILS 30
4.2 Optional outputs from IDA, IDADLS, IDASLS, and IDASPILS 45

5.1 Keys for setting FIDA optional inputs 85
5.2 Description of the FIDA optional output arrays `IOUT` and `ROUT` 86

6.1 Description of the NVECTOR operations 93
6.2 List of vector functions usage by IDA code modules 106

A.1 SUNDIALS libraries and header files 136
A.2 SUNDIALS libraries and header files (cont.) 137

List of Figures

3.1 Organization of the SUNDIALS suite 12
3.2 Overall structure diagram of the IDA package 13

8.1 Diagram of the storage for a banded matrix of type `DlsMat` 114
8.2 Diagram of the storage for a compressed-sparse-column matrix of type `SlsMat` 122

A.1 Initial *ccmake* configuration screen 129
A.2 Changing the *instdir* 130

Chapter 1

Introduction

IDA is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [16]. This suite consists of CVODE, ARKODE, KINSOL, and IDA, and variants of these with sensitivity analysis capabilities.

IDA is a general purpose solver for the initial value problem (IVP) for systems of differential-algebraic equations (DAEs). The name IDA stands for Implicit Differential-Algebraic solver. IDA is based on DASPK [5, 6], but is written in ANSI-standard C rather than FORTRAN77. Its most notable features are that, (1) in the solution of the underlying nonlinear system at each time step, it offers a choice of Newton/direct methods and a choice of Inexact Newton/Krylov (iterative) methods; and (2) it is written in a *data-independent* manner in that it acts on generic vectors without any assumptions on the underlying organization of the data. Thus IDA shares significant modules previously written within CASC at LLNL to support the ordinary differential equation (ODE) solvers CVODE [17, 10] and PVODE [8, 9], and also the nonlinear system solver KINSOL [11].

The Newton/Krylov methods in IDA are: the GMRES (Generalized Minimal RESidual) [21], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [22], and TFQMR (Transpose-Free Quasi-Minimal Residual) linear iterative methods [14]. 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 DAE systems, the Krylov methods are preferable over direct linear solver methods, and are often the only feasible choice. Among the three Krylov methods in IDA, we recommend GMRES as the best overall choice. However, users are encouraged to compare all three, especially if encountering convergence failures with GMRES. Bi-CGFStab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for GMRES depends on the desired Krylov subspace size.

There are several motivations for choosing the C language for IDA. First, a general movement away from FORTRAN and toward C in scientific computing was apparent. Second, the pointer, structure, and dynamic memory allocation features in C are extremely useful in software of this complexity, with the great variety of method options offered. Finally, we prefer C over C++ for IDA 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.

1.1 Changes from previous versions

Changes in v2.8.0

Two major additions were made to the linear system solvers that are available for use with the IDA solver. First, in the serial case, an interface to the sparse direct solver KLU was added. Second, an interface to SuperLU_MT, the multi-threaded version of SuperLU, was added as a thread-parallel

sparse direct solver option, to be used with the serial version of the NVECTOR module. As part of these additions, a sparse matrix (CSC format) structure was added to IDA.

Otherwise, only relatively minor modifications were made to IDA:

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

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

A minor bug was fixed regarding the testing of the input `tstop` on the first call to `IDASolve`.

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

In the FIDA optional input routines `FIDASETIIN`, `FIDASETRIN`, and `FIDASETVIN`, the optional fourth argument `key_length` was removed, with hardcoded key string lengths passed to all `strncmp` tests.

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

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

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

Changes in v2.7.0

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

A large number of minor errors have been fixed. Among these are the following: After the solver memory is created, it is set to zero before being filled. To be consistent with IDAS, IDA uses the function `IDAGetDky` for optional output retrieval. In each linear solver interface function, the linear solver memory is freed on an error return, and the `**Free` function now includes a line setting to NULL the main memory pointer to the linear solver memory. A memory leak was fixed in two of the `IDASp***Free` functions. In the rootfinding functions `IDARcheck1`/`IDARcheck2`, when an exact zero is found, the array `glo` of g values at the left endpoint is adjusted, instead of shifting the t location `tlo` slightly. In the installation files, we modified the treatment of the macro SUNDIALS_USE_GENERIC_MATH, so that the parameter GENERIC_MATH_LIB is either defined (with no value) or not defined.

Changes in v2.6.0

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

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

Changes in v2.5.0

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

A bug was fixed in the internal difference-quotient dense and banded Jacobian approximations, related to the estimation of the perturbation (which could have led to a failure of the linear solver when zero components with sufficiently small absolute tolerances were present).

The user interface to the consistent initial conditions calculations was modified. The `IDACalcIC` arguments `t0`, `yy0`, and `yp0` were removed and a new function, `IDAGetconsistentIC` is provided (see §4.5.4 and §4.5.9.2 for details).

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

Changes in v2.4.0

FIDA, a FORTRAN-C interface module, was added (for details see Chapter 5).

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

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

A user-callable routine was added to access the estimated local error vector.

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 (`ida_` and `sundials_`). When using the default installation procedure, the header files are exported under various subdirectories of the target `include` directory. For more details see Appendix A.

Changes in v2.3.0

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

Changes in v2.2.2

Minor corrections and improvements were made to the build system. A new chapter in the User Guide was added — with constants that appear in the user interface.

Changes in v2.2.1

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

Changes in v2.2.0

The major changes from the previous version involve a redesign of the user interface across the entire SUNDIALS suite. We have eliminated the mechanism of providing optional inputs and extracting optional statistics from the solver through the `iopt` and `ropt` arrays. Instead, IDA now provides a set of routines (with prefix `IDASet`) to change the default values for various quantities controlling the solver and a set of extraction routines (with prefix `IDAGet`) to extract statistics after return from the main solver routine. Similarly, each linear solver module provides its own set of `Set`- and `Get`-type routines. For more details see §4.5.7 and §4.5.9.

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.

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

1.2 Reading this User Guide

The structure of this document is as follows:

- In Chapter 2, we give short descriptions of the numerical methods implemented by IDA for the solution of initial value problems for systems of DAEs, along with short descriptions of preconditioning (§2.2) and rootfinding (§2.3).
- The following chapter describes the structure of the SUNDIALS suite of solvers (§3.1) and the software organization of the IDA solver (§3.2).
- Chapter 4 is the main usage document for IDA for C applications. It includes a complete description of the user interface for the integration of DAE initial value problems.
- In Chapter 5, we describe FIDA, an interface module for the use of IDA with FORTRAN applications.
- Chapter 6 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, as well as details on the NVECTOR implementations provided with SUNDIALS: a serial implementation (§6.1), a distributed memory parallel implementation based on MPI (§6.2), and two thread-parallel implementations based on openMP (§6.3) and Pthreads (§6.4), respectively.
- Chapter 7 describes the interfaces to the linear solver modules, so that a user can provide his/her own such module.
- Chapter 8 describes in detail the generic linear solvers shared by all SUNDIALS solvers.
- Finally, in the appendices, we provide detailed instructions for the installation of IDA, within the structure of SUNDIALS (Appendix A), as well as a list of all the constants used for input to and output from IDA functions (Appendix B).

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

Acknowledgments. We wish to acknowledge the contributions to previous versions of the IDA code and user guide of Allan G. Taylor.

Chapter 2

Mathematical Considerations

IDA solves the initial-value problem (IVP) for a DAE system of the general form

$$F(t, y, \dot{y}) = 0\,, \quad y(t_0) = y_0\,,\ \dot{y}(t_0) = \dot{y}_0\,, \tag{2.1}$$

where y, $\dot{y}$, and F are vectors in $\mathbf{R}^N$, t is the independent variable, $\dot{y} = dy/dt$, and initial values y_0, $\dot{y}_0$ are given. (Often t is time, but it certainly need not be.)

2.1 IVP solution

Prior to integrating a DAE initial-value problem, an important requirement is that the pair of vectors y_0 and $\dot{y}_0$ are both initialized to satisfy the DAE residual $F(t_0, y_0, \dot{y}_0) = 0$. For a class of problems that includes so-called semi-explicit index-one systems, IDA provides a routine that computes consistent initial conditions from a user's initial guess [6]. For this, the user must identify sub-vectors of y (not necessarily contiguous), denoted y_d and y_a, which are its differential and algebraic parts, respectively, such that F depends on $\dot{y}_d$ but not on any components of $\dot{y}_a$. The assumption that the system is "index one" means that for a given t and y_d, the system $F(t, y, \dot{y}) = 0$ defines y_a uniquely. In this case, a solver within IDA computes y_a and $\dot{y}_d$ at $t = t_0$, given y_d and an initial guess for y_a. A second available option with this solver also computes all of $y(t_0)$ given $\dot{y}(t_0)$; this is intended mainly for quasi-steady-state problems, where $\dot{y}(t_0) = 0$ is given. In both cases, IDA solves the system $F(t_0, y_0, \dot{y}_0) = 0$ for the unknown components of y_0 and $\dot{y}_0$, using Newton iteration augmented with a line search global strategy. In doing this, it makes use of the existing machinery that is to be used for solving the linear systems during the integration, in combination with certain tricks involving the step size (which is set artificially for this calculation). For problems that do not fall into either of these categories, the user is responsible for passing consistent values, or risk failure in the numerical integration.

The integration method used in IDA is the variable-order, variable-coefficient BDF (Backward Differentiation Formula), in fixed-leading-coefficient form [3]. The method order ranges from 1 to 5, with the BDF of order q given by the multistep formula

$$\sum_{i=0}^{q} \alpha_{n,i} y_{n-i} = h_n \dot{y}_n\,, \tag{2.2}$$

where y_n and $\dot{y}_n$ are the computed approximations to $y(t_n)$ and $\dot{y}(t_n)$, respectively, and the step size is $h_n = t_n - t_{n-1}$. The coefficients $\alpha_{n,i}$ are uniquely determined by the order q, and the history of the step sizes. The application of the BDF (2.2) to the DAE system (2.1) results in a nonlinear algebraic system to be solved at each step:

$$G(y_n) \equiv F\left(t_n,\, y_n,\, h_n^{-1} \sum_{i=0}^{q} \alpha_{n,i} y_{n-i}\right) = 0\,. \tag{2.3}$$

Regardless of the method options, the solution of the nonlinear system (2.3) is accomplished with some form of Newton iteration. This leads to a linear system for each Newton correction, of the form

$$J[y_{n(m+1)} - y_{n(m)}] = -G(y_{n(m)}) , \tag{2.4}$$

where $y_{n(m)}$ is the m-th approximation to y_n. Here J is some approximation to the system Jacobian

$$J = \frac{\partial G}{\partial y} = \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial \dot{y}} , \tag{2.5}$$

where $\alpha = \alpha_{n,0}/h_n$. The scalar α changes whenever the step size or method order changes.

For the solution of the linear systems within the Newton corrections, IDA provides several choices, including the option of an user-supplied linear solver module. The linear solver modules distributed with SUNDIALS are organized in three families, a *direct* family comprising direct linear solvers for dense or banded matrices, a *sparse* family comprising direct linear solvers for matrices stored in compressed-sparse-column format, and a *spils* family comprising scaled preconditioned iterative (Krylov) linear solvers. The methods offered through these modules are as follows:

- dense direct solvers, using either an internal implementation or a Blas/Lapack implementation (serial or threaded vector modules only),
- band direct solvers, using either an internal implementation or a Blas/Lapack implementation (serial or threaded vector modules only),
- sparse direct solver interfaces, using either the KLU sparse solver library [12, 1], or the thread-enabled SuperLU_MT sparse solver library [20, 13, 2] (serial or threaded vector modules only) [Note that users will need to download and install the KLU or SuperLU_MT packages independent of IDA],
- SPGMR, a scaled preconditioned GMRES (Generalized Minimal Residual method) solver without restarts,
- SPBCG, a scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method) solver, or
- SPTFQMR, a scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method) solver.

For large stiff systems, where direct methods are not feasible, the combination of a BDF integrator and any of the preconditioned Krylov methods (SPGMR, SPBCG, or SPTFQMR) yields a powerful tool because it combines established methods for stiff integration, nonlinear iteration, and Krylov (linear) iteration with a problem-specific treatment of the dominant source of stiffness, in the form of the user-supplied preconditioner matrix [4]. For the *spils* linear solvers, preconditioning is allowed only on the left (see §2.2). Note that the direct linear solvers (dense, band, and sparse) can only be used with serial and threaded vector representations.

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

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

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

In the case of a direct linear solver (dense, band, or sparse), the nonlinear iteration (2.4) is a Modified Newton iteration, in that the Jacobian J is fixed (and usually out of date), with a coefficient $\bar{\alpha}$ in place of α in J. When using one of the Krylov methods SPGMR, SPBCG, or SPTFQMR as the linear solver, the iteration is an Inexact Newton iteration, using the current Jacobian (through matrix-free products Jv), in which the linear residual $J\Delta y + G$ is nonzero but controlled. The Jacobian matrix J (direct cases) or preconditioner matrix P (SPGMR/SPBCG/SPTFQMR case) is updated when:

- starting the problem,
- the value $\bar{\alpha}$ at the last update is such that $\alpha/\bar{\alpha} < 3/5$ or $\alpha/\bar{\alpha} > 5/3$, or
- a non-fatal convergence failure occurred with an out-of-date J or P.

The above strategy balances the high cost of frequent matrix evaluations and preprocessing with the slow convergence due to infrequent updates. To reduce storage costs on an update, Jacobian information is always reevaluated from scratch.

We note that with the sparse direct solvers, the Jacobian *must* be supplied by a user routine in compressed-sparse-column format, as this is not approximated automatically within IDA.

The stopping test for the Newton iteration in IDA ensures that the iteration error $y_n - y_{n(m)}$ is small relative to y itself. For this, we estimate the linear convergence rate at all iterations $m > 1$ as

$$R = \left(\frac{\delta_m}{\delta_1}\right)^{\frac{1}{m-1}} ,$$

where the $\delta_m = y_{n(m)} - y_{n(m-1)}$ is the correction at iteration $m = 1, 2, \ldots$. The Newton iteration is halted if $R > 0.9$. The convergence test at the m-th iteration is then

$$S\|\delta_m\| < 0.33 , \tag{2.7}$$

where $S = R/(R-1)$ whenever $m > 1$ and $R \le 0.9$. The user has the option of changing the constant in the convergence test from its default value of 0.33. The quantity S is set to $S = 20$ initially and whenever J or P is updated, and it is reset to $S = 100$ on a step with $\alpha \neq \bar{\alpha}$. Note that at $m = 1$, the convergence test (2.7) uses an old value for S. Therefore, at the first Newton iteration, we make an additional test and stop the iteration if $\|\delta_1\| < 0.33 \cdot 10^{-4}$ (since such a δ_1 is probably just noise and therefore not appropriate for use in evaluating R). We allow only a small number (default value 4) of Newton iterations. If convergence fails with J or P current, we are forced to reduce the step size h_n, and we replace h_n by $h_n/4$. The integration is halted after a preset number (default value 10) of convergence failures. Both the maximum allowable Newton iterations and the maximum nonlinear convergence failures can be changed by the user from their default values.

When SPGMR, SPBCG, or SPTFQMR is used to solve the linear system, to minimize the effect of linear iteration errors on the nonlinear and local integration error controls, we require the preconditioned linear residual to be small relative to the allowed error in the Newton iteration, i.e., $\|P^{-1}(Jx+G)\| < 0.05 \cdot 0.33$. The safety factor 0.05 can be changed by the user.

In the direct linear solver cases, the Jacobian J defined in (2.5) can be either supplied by the user or have IDA compute one internally by difference quotients. In the latter case, we use the approximation

$$\begin{aligned} J_{ij} &= [F_i(t, y + \sigma_j e_j, \dot{y} + \alpha\sigma_j e_j) - F_i(t, y, \dot{y})]/\sigma_j , \text{ with} \\ \sigma_j &= \sqrt{U} \max\{|y_j|, |h\dot{y}_j|, 1/W_j\} \operatorname{sign}(h\dot{y}_j) , \end{aligned}$$

where U is the unit roundoff, h is the current step size, and W_j is the error weight for the component y_j defined by (2.6). In the SPGMR/SPBCG/SPTFQMR case, if a routine for Jv is not supplied, such products are approximated by

$$Jv = [F(t, y + \sigma v, \dot{y} + \alpha\sigma v) - F(t, y, \dot{y})]/\sigma ,$$

where the increment σ is $1/\|v\|$. As an option, the user can specify a constant factor that is inserted into this expression for σ.

During the course of integrating the system, IDA computes an estimate of the local truncation error, LTE, at the n-th time step, and requires this to satisfy the inequality

$$\|\text{LTE}\|_{\text{WRMS}} \le 1 .$$

Asymptotically, LTE varies as h^{q+1} at step size h and order q, as does the predictor-corrector difference $\Delta_n \equiv y_n - y_{n(0)}$. Thus there is a constant C such that

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

and so the norm of LTE is estimated as $|C| \cdot \|\Delta_n\|$. In addition, IDA requires that the error in the associated polynomial interpolant over the current step be bounded by 1 in norm. The leading term of the norm of this error is bounded by $\bar{C}\|\Delta_n\|$ for another constant $\bar{C}$. Thus the local error test in IDA is

$$\max\{|C|, \bar{C}\}\|\Delta_n\| \leq 1\,. \tag{2.8}$$

A user option is available by which the algebraic components of the error vector are omitted from the test (2.8), if these have been so identified.

In IDA, the local error test is tightly coupled with the logic for selecting the step size and order. First, there is an initial phase that is treated specially; for the first few steps, the step size is doubled and the order raised (from its initial value of 1) on every step, until (a) the local error test (2.8) fails, (b) the order is reduced (by the rules given below), or (c) the order reaches 5 (the maximum). For step and order selection on the general step, IDA uses a different set of local error estimates, based on the asymptotic behavior of the local error in the case of fixed step sizes. At each of the orders q' equal to q, $q-1$ (if $q > 1$), $q-2$ (if $q > 2$), or $q+1$ (if $q < 5$), there are constants $C(q')$ such that the norm of the local truncation error at order q' satisfies

$$\text{LTE}(q') = C(q')\|\phi(q'+1)\| + O(h^{q'+2})\,,$$

where $\phi(k)$ is a modified divided difference of order k that is retained by IDA (and behaves asymptotically as h^k). Thus the local truncation errors are estimated as $\text{ELTE}(q') = C(q')\|\phi(q'+1)\|$ to select step sizes. But the choice of order in IDA is based on the requirement that the scaled derivative norms, $\|h^k y^{(k)}\|$, are monotonically decreasing with k, for k near q. These norms are again estimated using the $\phi(k)$, and in fact

$$\|h^{q'+1} y^{(q'+1)}\| \approx T(q') \equiv (q'+1)\text{ELTE}(q')\,.$$

The step/order selection begins with a test for monotonicity that is made even *before* the local error test is performed. Namely, the order is reset to $q' = q-1$ if (a) $q = 2$ and $T(1) \leq T(2)/2$, or (b) $q > 2$ and $\max\{T(q-1), T(q-2)\} \leq T(q)$; otherwise $q' = q$. Next the local error test (2.8) is performed, and if it fails, the step is redone at order $q \leftarrow q'$ and a new step size h'. The latter is based on the h^{q+1} asymptotic behavior of $\text{ELTE}(q)$, and, with safety factors, is given by

$$\eta = h'/h = 0.9/[2\,\text{ELTE}(q)]^{1/(q+1)}\,.$$

The value of η is adjusted so that $0.25 \leq \eta \leq 0.9$ before setting $h \leftarrow h' = \eta h$. If the local error test fails a second time, IDA uses $\eta = 0.25$, and on the third and subsequent failures it uses $q = 1$ and $\eta = 0.25$. After 10 failures, IDA returns with a give-up message.

As soon as the local error test has passed, the step and order for the next step may be adjusted. No such change is made if $q' = q-1$ from the prior test, if $q = 5$, or if q was increased on the previous step. Otherwise, if the last $q+1$ steps were taken at a constant order $q < 5$ and a constant step size, IDA considers raising the order to $q+1$. The logic is as follows: (a) If $q = 1$, then reset $q = 2$ if $T(2) < T(1)/2$. (b) If $q > 1$ then

- reset $q \leftarrow q-1$ if $T(q-1) \leq \min\{T(q), T(q+1)\}$;
- else reset $q \leftarrow q+1$ if $T(q+1) < T(q)$;
- leave q unchanged otherwise [then $T(q-1) > T(q) \leq T(q+1)$].

In any case, the new step size h' is set much as before:

$$\eta = h'/h = 1/[2\,\text{ELTE}(q)]^{1/(q+1)}\,.$$

The value of η is adjusted such that (a) if $\eta > 2$, η is reset to 2; (b) if $\eta \leq 1$, η is restricted to $0.5 \leq \eta \leq 0.9$; and (c) if $1 < \eta < 2$ we use $\eta = 1$. Finally h is reset to $h' = \eta h$. Thus we do not increase the step size unless it can be doubled. See [3] for details.

IDA permits the user to impose optional inequality constraints on individual components of the solution vector y. Any of the following four constraints can be imposed: $y_i > 0$, $y_i < 0$, $y_i \geq 0$, or $y_i \leq 0$. The constraint satisfaction is tested after a successful nonlinear system solution. If any constraint fails, we declare a convergence failure of the Newton iteration and reduce the step size. Rather than cutting the step size by some arbitrary factor, IDA estimates a new step size h' using a linear approximation of the components in y that failed the constraint test (including a safety factor of 0.9 to cover the strict inequality case). These additional constraints are also imposed during the calculation of consistent initial conditions.

Normally, IDA takes steps until a user-defined output value $t = t_{\text{out}}$ is overtaken, and then 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 IDA not to integrate past a given stopping point $t = t_{\text{stop}}$.

2.2 Preconditioning

When using a Newton method to solve the nonlinear system (2.4), IDA makes repeated use of a linear solver to solve linear systems of the form $J\Delta y = -G$. If this linear system solve is done with one of the scaled preconditioned iterative linear solvers, these solvers are rarely successful if used without preconditioning; it is generally necessary to precondition the system in order to obtain acceptable efficiency. A system $Ax = b$ can be preconditioned on the left, on the right, or on both sides. The Krylov method is then applied to a system with the matrix $P^{-1}A$, or AP^{-1}, or $P_L^{-1}AP_R^{-1}$, instead of A. However, within IDA, preconditioning is allowed *only* on the left, so that the iterative method is applied to systems $(P^{-1}J)\Delta y = -P^{-1}G$. Left preconditioning is required to make the norm of the linear residual in the Newton iteration meaningful; in general, $\|J\Delta y + G\|$ is meaningless, since the weights used in the WRMS-norm correspond to y.

In order to improve the convergence of the Krylov iteration, the preconditioner matrix P should in some sense approximate the system matrix A. Yet at the same time, in order to be cost-effective, the matrix P should be reasonably efficient to evaluate and solve. Finding a good point in this tradeoff between rapid convergence and low cost can be very difficult. Good choices are often problem-dependent (for example, see [4] for an extensive study of preconditioners for reaction-transport systems).

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

2.3 Rootfinding

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

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

The basic scheme used is to check for sign changes of any $g_i(t)$ over each time step taken, and then (when a sign change is found) to home in on the root (or roots) with a modified secant method [15]. In addition, each time g is computed, IDA 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, IDA computes g at $t + \delta$ for a small increment δ, slightly further in the direction of integration, and if any $g_i(t + \delta) = 0$ also, IDA stops and reports an error. This way, each time IDA 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, IDA has an interval $(t_{lo}, t_{hi}]$ in which roots of the $g_i(t)$ are to be sought, such that t_{hi} is further ahead in the direction of integration, and all $g_i(t_{lo}) \neq 0$. The endpoint t_{hi} is either t_n, the end of the time step last taken, or the next requested output time t_{out} if this comes sooner. The endpoint t_{lo} is either t_{n-1}, or the last output time t_{out} (if this occurred within the last step), or the last root location (if a root was just located within this step), possibly adjusted slightly toward t_n if an exact zero was found. The algorithm checks g at t_{hi} for zeros and for sign changes in (t_{lo}, t_{hi}). If no sign changes are found, then either a root is reported (if some $g_i(t_{hi}) = 0$) or we proceed to the next time interval (starting at t_{hi}). If one or more sign changes were found, then a loop is entered to locate the root to within a rather tight tolerance, given by

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

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

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

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

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

Chapter 3

Code Organization

3.1 SUNDIALS organization

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

The various solvers of this family share many subordinate modules. For this reason, it is organized as a family, with a directory structure that exploits that sharing (see Fig. 3.1). The following is a list of the solver packages presently available, and the basic functionality of each:

- CVODE, a solver for stiff and nonstiff ODE systems $dy/dt = f(t, y)$ based on Adams and BDF methods;
- CVODES, a solver for stiff and nonstiff ODE systems with sensitivity analysis capabilities;
- ARKODE, a solver for ODE systems $M dy/dt = f(t, y)$ based on additive Runge-Kutta methods;
- IDA, a solver for differential-algebraic systems $F(t, y, \dot{y}) = 0$ based on BDF methods;
- IDAS, a solver for differential-algebraic systems with sensitivity analysis capabilities;
- KINSOL, a solver for nonlinear algebraic systems $F(u) = 0$.

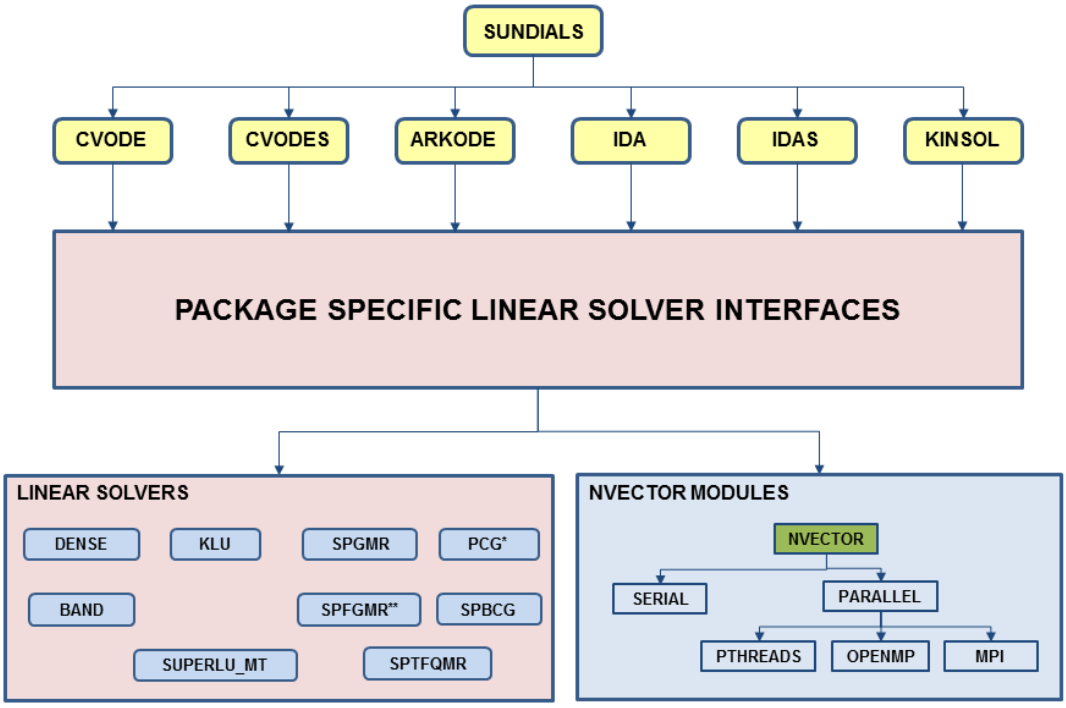
3.2 IDA organization

The IDA 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.

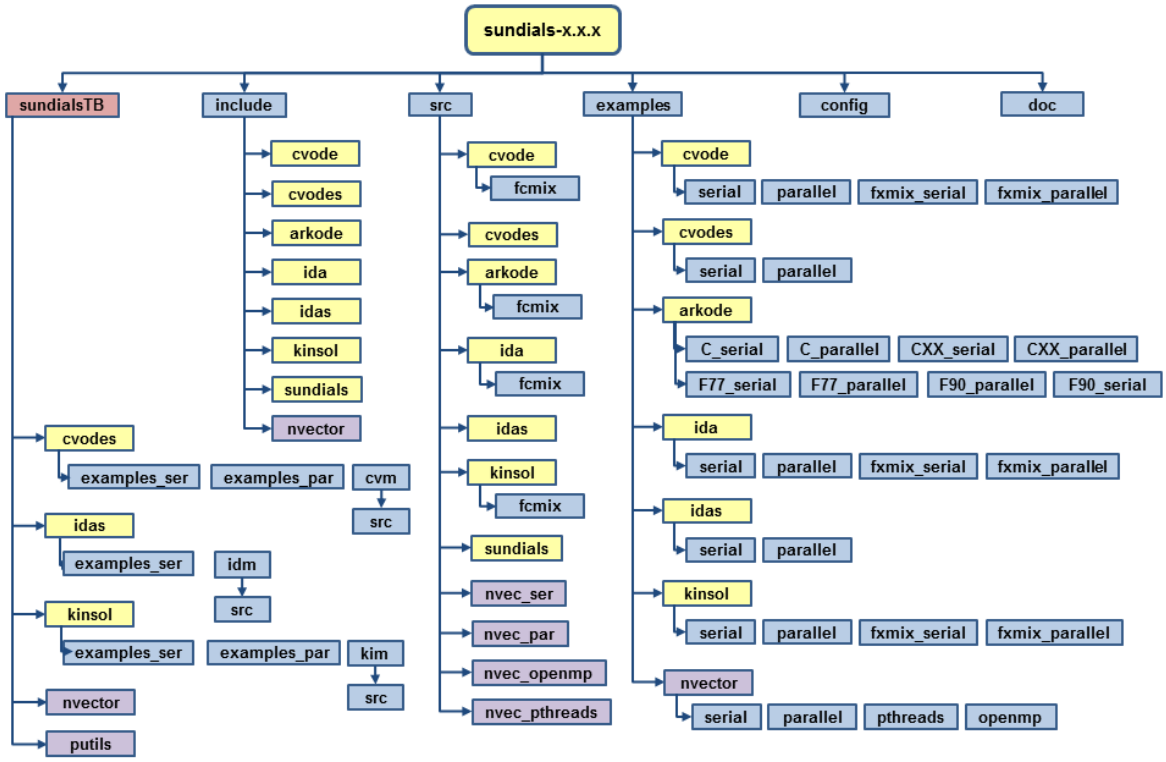
The overall organization of the IDA package is shown in Figure 3.2. The central integration module, implemented in the files `ida.h`, `ida_impl.h`, and `ida.c`, deals with the evaluation of integration coefficients, the Newton iteration process, estimation of local error, selection of stepsize and order, 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 seven IDA linear algebra modules, organized into two families. The *direct* family of linear solvers provides solvers for the direct solution of linear systems with dense, banded, or sparse matrices and includes:

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



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



(b) Directory structure of the source tree

Figure 3.1: Organization of the SUNDIALS suite

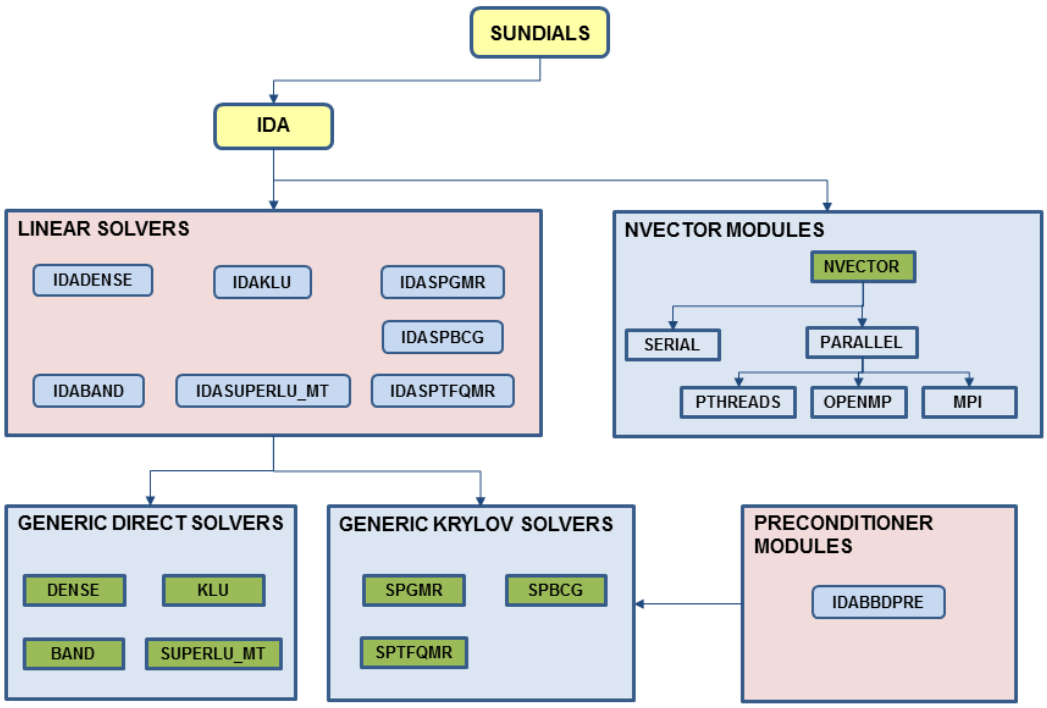


Figure 3.2: Overall structure diagram of the IDA package. Modules specific to IDA are distinguished by rounded boxes, while generic solver and auxiliary modules are in square boxes. Note that the direct linear solvers using Lapack implementations are not explicitly represented. Note also that the KLU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

- IDABAND: LU factorization and backsolving with banded matrices (using either an internal implementation or Blas/Lapack);
- IDAKLU: LU factorization and backsolving with compressed-sparse-column (CSC) matrices using the KLU linear solver library [12, 1] (KLU to be downloaded and compiled by user independent of IDA);
- IDASUPERLUMT: LU factorization and backsolving with compressed-sparse-column (CSC) matrices using the threaded SuperLU_MT linear solver library [20, 13, 2] (SuperLU_MT to be downloaded and compiled by user independent of IDA).

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

- IDASPGMR: scaled preconditioned GMRES method;
- IDASPBCG: scaled preconditioned Bi-CGStab method;
- IDASPTFQMR: scaled preconditioned TFQMR method.

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

In the case of the direct methods IDADENSE and IDABAND, the package includes an algorithm for the approximation of the Jacobian by difference quotients, but the user also has the option of supplying the Jacobian (or an approximation to it) directly. When using the sparse direct linear solvers CVKLU and CVSUPERLUMT, the user must supply a routine for the Jacobian (or an approximation to it) in CSC format, since standard difference quotient approximations do not leverage the inherent sparsity of the problem. In the case of the Krylov iterative methods IDASPGMR, IDASPBCG, and IDASPTFQMR, 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. When using any of the Krylov methods, the user must supply the preconditioning in two phases: a setup phase (preprocessing of Jacobian data) and a solve phase. While there is no default choice of preconditioner analogous to the difference quotient approximation in the direct case, the references [4, 7], together with the example and demonstration programs included with IDA, offer considerable assistance in building preconditioners.

Each IDA linear solver module consists of five routines, devoted to (1) memory allocation and initialization, (2) setup of the matrix data involved, (3) solution of the system, (4) monitoring performance, and (5) 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 IDA 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 (IDADENSE, etc.) consists of an interface built on top of a generic linear system solver (DENSE, etc.). The interface deals with the use of the particular method in the IDA context, whereas the generic solver is independent of the context. While some of the generic linear system solvers (DENSE, BAND, SPGMR, SPBCG, and SPTFQMR) were written with SUNDIALS in mind, they are intended to be usable anywhere as general-purpose solvers. This separation also allows for any generic solver to be replaced by an improved version, with no necessity to revise the IDA package elsewhere.

IDA also provides a preconditioner module, IDABBDPRE, that works in conjunction with NVECTOR_PARALLEL and generates a preconditioner that is a block-diagonal matrix with each block being a band matrix.

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

Chapter 4

Using IDA for C Applications

This chapter is concerned with the use of IDA for the integration of DAEs in a C language setting. The following sections treat the header files, the layout of the user's main program, description of the IDA user-callable functions, and description of user-supplied functions.

The sample programs described in the companion document [18] may also be helpful. Those codes may be used as templates (with the removal of some lines involved in testing), and are included in the IDA package.

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

The user should be aware that not all linear solver modules are compatible with all NVECTOR implementations. For example, NVECTOR_PARALLEL is not compatible with the direct dense, direct band or direct sparse linear solvers, since these linear solver modules need to form the complete system Jacobian. The IDADENSE and IDABAND modules (using either the internal implementation or Lapack) can only be used with NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS. It is not recommended to use a threaded vector module with SuperLU_MT unless it is the NVECTOR_OPENMP module and SuperLU_MT is also compiled with openMP. The preconditioner module IDABBDPRE can only be used with NVECTOR_PARALLEL.

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

4.1 Access to library and header files

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

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

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

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

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

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

4.2 Data types

The sundials_types.h file contains the definition of the type realtype, which is used by the SUNDIALS solvers for all floating-point data. The type realtype can be float, double, or long double, with the default being double. The user can change the precision of the SUNDIALS solvers arithmetic at the configuration stage (see §A.1.2).

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

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

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

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

A user program which uses the type realtype and the RCONST macro to handle floating-point constants is precision-independent except for any calls to precision-specific standard math library functions. (Our example programs use both realtype and RCONST.) Users can, however, use the type double, float, or long double in their code (assuming that this usage is consistent with the typedef for realtype). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use realtype, so long as the SUNDIALS libraries use the correct precision (for details see §A.1.2).

4.3 Header files

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

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

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

The calling program must also include an NVECTOR implementation header file (see Chapter 6 for details). For the NVECTOR implementations that are included in the IDA package, the corresponding header files are:

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

Note that both these files include in turn the header file sundials_nvector.h which defines the abstract N_Vector type.

Finally, a linear solver module header file is required. The header files corresponding to the various linear solver options in IDA are as follows:

- ida_dense.h, which is used with the dense direct linear solver;

- `ida_band.h`, which is used with the band direct linear solver;
- `ida_lapack.h`, which is used with Lapack implementations of dense or band direct linear solvers;
- `ida_klu.h`, which is used with the KLU sparse direct linear solver;
- `ida_superlumt.h`, which is used with the SuperLU_MT threaded sparse direct linear solver;
- `ida_spgmr.h`, which is used with the scaled, preconditioned GMRES Krylov linear solver SPGMR;
- `ida_spbcgs.h`, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver SPBCG;
- `ida_sptfqmr.h`, which is used with the scaled, preconditioned TFQMR Krylov solver SPTFQMR.

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

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

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

Other headers may be needed, according to the choice of preconditioner, etc. For example, in the `idaFoodWeb_kry_p` example (see [18]), preconditioning is done with a block-diagonal matrix. For this, even though the IDASPGMR linear solver is used, the header `sundials_dense.h` is included for access to the underlying generic dense linear solver.

4.4 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) for the integration of a DAE IVP. Some steps are independent of the NVECTOR implementation used; where this is not the case, usage specifications are given for the two implementations provided with IDA: steps marked [**P**] correspond to NVECTOR_PARALLEL, steps marked [**O**] correspond to NVECTOR_OPENMP, steps marked [**T**] correspond to NVECTOR_PTHREADS, while steps marked [**S**] correspond to NVECTOR_SERIAL.

1. **[P] Initialize MPI**

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

2. **Set problem dimensions**

   [**S**], [**O**], [**T**] 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 processors.

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

   Note: The variables `N` and `Nlocal` should be of type `long int`. The variable `num_threads` should be of type `int`.

3. **Set vectors of initial values**

   To set the vectors `y0` and `yp0` to initial values for y and $\dot{y}$, use functions defined by the particular NVECTOR implementation. For the two NVECTOR implementations provided, if a `realtype` array `ydata` already exists, containing the initial values of y, make the calls:

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

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

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

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

   Otherwise, make the calls:

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

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

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

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

   and load initial values into the structure defined by:

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

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

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

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

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

   The initial conditions for $\dot{y}$ are set similarly.

4. **Create IDA object**

   Call `ida_mem = IDACreate()` to create the IDA memory block. `IDACreate` returns a pointer to the IDA memory structure. See §4.5.1 for details. This `void *` pointer must then be passed as the first argument to all subsequent IDA function calls.

5. **Initialize IDA solver**

   Call `IDAInit(...)` to provide required problem specifications (residual function, initial time, and initial conditions), allocate internal memory for IDA, and initialize IDA. `IDAInit` returns an error flag to indicate success or an illegal argument value. See §4.5.1 for details.

6. **Specify integration tolerances**

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

7. **Set optional inputs**

   Optionally, call `IDASet*` functions to change from their default values any optional inputs that control the behavior of IDA. See §4.5.7.1 for details.

8. **Attach linear solver module**

   Initialize the linear solver module with one of the following calls (for details see §4.5.3):

   [**S**], [**O**], [**T**] `flag = IDADense(...);`

[**S**], [**O**], [**T**] `flag = IDABand(...);`

[**S**], [**O**], [**T**] `flag = IDALapackDense(...);`

[**S**], [**O**], [**T**] `flag = IDALapackBand(...);`

[**S**], [**O**], [**T**] `flag = IDAKLU(...);`

[**S**], [**O**], [**T**] `flag = IDASuperLUMT(...);`

`flag = IDASpgmr(...);`

`flag = IDASpbcg(...);`

`flag = IDASptfqmr(...);`

9. **Set linear solver optional inputs**

   Optionally, call `IDA*Set*` functions from the selected linear solver module to change optional inputs specific to that linear solver. See §4.5.7.2 and §4.5.7.4 for details.

10. **Correct initial values**

    Optionally, call `IDACalcIC` to correct the initial values `y0` and `yp0` passed to `IDAInit`. See §4.5.4. Also see §4.5.7.5 for relevant optional input calls.

11. **Specify rootfinding problem**

    Optionally, call `IDARootInit` to initialize a rootfinding problem to be solved during the integration of the DAE system. See §4.5.5 for details, and see §4.5.7.6 for relevant optional input calls.

12. **Advance solution in time**

    For each point at which output is desired, call `flag = IDASolve(ida_mem, tout, &tret, yret, ypret, itask)`. Here `itask` specifies the return mode. The vector `yret` (which can be the same as the vector `y0` above) will contain $y(t)$, while the vector `ypret` will contain $\dot{y}(t)$. See §4.5.6 for details.

13. **Get optional outputs**

    Call `IDA*Get*` functions to obtain optional output. See §4.5.9 for details.

14. **Deallocate memory for solution vectors**

    Upon completion of the integration, deallocate memory for the vectors `yret` and `ypret` by calling the destructor function defined by the NVECTOR implementation:

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

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

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

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

    and similarly for `ypret`.

15. **Free solver memory**

    `IDAFree(&ida_mem)` to free the memory allocated for IDA.

16. **[P] Finalize MPI**

    Call `MPI_Finalize()` to terminate MPI.

4.5 User-callable functions

This section describes the IDA functions that are called by the user to set up and solve a DAE. Some of these are required. However, starting with §4.5.7, the functions listed involve optional inputs/outputs or restarting, and those paragraphs can be skipped for a casual use of IDA. In any case, refer to §4.4 for the correct order of these calls.

On an error, each user-callable function returns a negative value and sends an error message to the error handler routine, which prints the message on `stderr` by default. However, the user can set a file as error output or can provide his own error handler function (see §4.5.7.1).

4.5.1 IDA 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 DAE solution is complete, as it frees the IDA memory block created and allocated by the first two calls.

`IDACreate`

Call `ida_mem = IDACreate();`

Description The function `IDACreate` instantiates an IDA solver object.

Arguments `IDACreate` has no arguments.

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

`IDAInit`

Call `flag = IDAInit(ida_mem, res, t0, y0, yp0);`

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

Arguments
- `ida_mem` (`void *`) pointer to the IDA memory block returned by `IDACreate`.
- `res` (`IDAResFn`) is the C function which computes the residual function F in the DAE. This function has the form `res(t, yy, yp, resval, user_data)`. For full details see §4.6.1.
- `t0` (`realtype`) is the initial value of t.
- `y0` (`N_Vector`) is the initial value of y.
- `yp0` (`N_Vector`) is the initial value of $\dot{y}$.

Return value The return value `flag` (of type `int`) will be one of the following:
- `IDA_SUCCESS` The call to `IDAInit` was successful.
- `IDA_MEM_NULL` The IDA memory block was not initialized through a previous call to `IDACreate`.
- `IDA_MEM_FAIL` A memory allocation request has failed.
- `IDA_ILL_INPUT` An input argument to `IDAInit` has an illegal value.

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

`IDAFree`

Call `IDAFree(&ida_mem);`

Description The function `IDAFree` frees the pointer allocated by a previous call to `IDACreate`.

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

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

4.5.2 IDA tolerance specification functions

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

`IDASStolerances`

Call `flag = IDASStolerances(ida_mem, reltol, abstol);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block returned by `IDACreate`.
`reltol` (`realtype`) is the scalar relative error tolerance.
`abstol` (`realtype`) is the scalar absolute error tolerance.

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

`IDA_SUCCESS`	The call to `IDASStolerances` was successful.
`IDA_MEM_NULL`	The IDA memory block was not initialized through a previous call to `IDACreate`.
`IDA_NO_MALLOC`	The allocation function `IDAInit` has not been called.
`IDA_ILL_INPUT`	One of the input tolerances was negative.

`IDASVtolerances`

Call `flag = IDASVtolerances(ida_mem, reltol, abstol);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block returned by `IDACreate`.
`reltol` (`realtype`) is the scalar relative error tolerance.
`abstol` (`N_Vector`) is the vector of absolute error tolerances.

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

`IDA_SUCCESS`	The call to `IDASVtolerances` was successful.
`IDA_MEM_NULL`	The IDA memory block was not initialized through a previous call to `IDACreate`.
`IDA_NO_MALLOC`	The allocation function `IDAInit` has not been called.
`IDA_ILL_INPUT`	The relative error tolerance was negative or the absolute tolerance had a negative component.

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

`IDAWFtolerances`

Call `flag = IDAWFtolerances(ida_mem, efun);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block returned by `IDACreate`.
`efun` (`IDAEwtFn`) is the C function which defines the `ewt` vector (see §4.6.3).

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

`IDA_SUCCESS`	The call to `IDAWFtolerances` was successful.
`IDA_MEM_NULL`	The IDA memory block was not initialized through a previous call to `IDACreate`.

`IDA_NO_MALLOC` The allocation function `IDAInit` has not been called.

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

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

(2) The absolute tolerances `abstol` (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector `y` may be so small that pure relative error control is meaningless. For example, if `y[i]` starts at some nonzero value, but in time decays to zero, then pure relative error control on `y[i]` makes no sense (and is overly costly) after `y[i]` is below some noise level. Then `abstol` (if scalar) or `abstol[i]` (if a vector) needs to be set to that noise level. If the different components have different noise levels, then `abstol` should be a vector. See the example `idaRoberts_dns` in the IDA package, and the discussion of it in the IDA Examples document [18]. 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 a sort of accumulation of those per-step errors. A good rule of thumb is to reduce the tolerances by a factor of .01 from the actual desired limits on errors. So if you want .01% accuracy (globally), a good choice is `reltol`= 10^{-6}. But in any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

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

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

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

(3) The user's residual routine `res` should never change a negative value in the solution vector `yy` to a non-negative value, as a "solution" to this problem. This can cause instability. If the `res` 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 `yy` vector) for the purposes of computing $F(t, y, \dot{y})$.

(4) IDA provides the option of enforcing positivity or non-negativity on components. Also, such constraints can be enforced by use of the recoverable error return feature in the user-supplied residual function. However, because these options involve some extra overhead cost, they should only be exercised if the use of absolute tolerances to control the computed values is unsuccessful.

4.5.3 Linear solver specification functions

As previously explained, Newton iteration requires the solution of linear systems of the form (2.4). There are seven IDA linear solvers currently available for this task: IDADENSE, IDABAND, IDAKLU, IDASUPERLUMT, IDASPGMR, IDASPBCG, and IDASPTFQMR.

The first two linear solvers are direct and derive their names from the type of approximation

used for the Jacobian $J = \partial F/\partial y + \alpha \partial F/\partial \dot{y}$. IDADENSE and IDABAND work with dense and banded approximations to J, respectively. The SUNDIALS suite includes both internal implementations of these two linear solvers and interfaces to Lapack implementations. Together, these linear solvers are referred to as IDADLS (from Direct Linear Solvers).

The second two linear solvers are sparse direct solvers based on Gaussian elimination, and require user-supplied routines to construct the Jacobian $J = \partial F/\partial y + \alpha \partial F/\partial \dot{y}$ in compressed-sparse-column format. The SUNDIALS suite does not include internal implementations of these solver libraries, instead requiring compilation of SUNDIALS to link with existing installations of these libraries (if either is missing, SUNDIALS will install without the corresponding interface routines). Together, these linear solvers are referred to as IDASLS (from Sparse Linear Solvers).

The remaining three IDA linear solvers, IDASPGMR, IDASPBCG, and IDASPTFQMR, are Krylov iterative solvers. The SPGMR, SPBCG, and SPTFQMR in the names indicate the scaled preconditioned GMRES, scaled preconditioned Bi-CGStab, and scaled preconditioned TFQMR methods, respectively. Together, they are referred to as IDASPILS (from Scaled Preconditioned Iterative Linear Solvers).

When using any of the Krylov linear solvers, preconditioning (on the left) is permitted, and in fact encouraged, for the sake of efficiency. A preconditioner matrix P must approximate the Jacobian J, at least crudely. For the specification of a preconditioner, see §4.5.7.4 and §4.6.

To specify an IDA linear solver, after the call to `IDACreate` but before any calls to `IDASolve`, the user's program must call one of the functions `IDADense`/`IDALapackDense`, `IDABand`/`IDALapackBand`, `IDAKLU`, `IDASuperLUMT`, `IDASpgmr`, `IDASpbcg`, or `IDASptfqmr`, as documented below. The first argument passed to these functions is the IDA memory pointer returned by `IDACreate`. A call to one of these functions links the main IDA integrator to a linear solver and allows the user to specify parameters which are specific to a particular solver, such as the bandwidths in the IDABAND case. The use of each of the linear solvers involves certain constants and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the linear solver, as specified below.

In each case the linear solver module used by IDA is actually built on top of a generic linear system solver, which may be of interest in itself. These generic solvers, denoted DENSE, BAND, KLU, SUPERLUMT, SPGMR, SPBCG, and SPTFQMR, are described separately in Chapter 8.

`IDADense`

Call `flag = IDADense(ida_mem, N);`

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

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

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

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

`IDADLS_SUCCESS`	The IDADENSE initialization was successful.
`IDADLS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDADLS_ILL_INPUT`	The IDADENSE solver is not compatible with the current NVECTOR module.
`IDADLS_MEM_FAIL`	A memory allocation request failed.

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

`IDALapackDense`

Call `flag = IDALapackDense(ida_mem, N);`

Description The function `IDALapackDense` selects the IDADENSE linear solver and indicates the use of Lapack functions.

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`N` (`int`) problem dimension.

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

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

`IDABand`

Call `flag = IDABand(ida_mem, N, mupper, mlower);`

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

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

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

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

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

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

`IDABAND_SUCCESS`	The IDABAND initialization was successful.
`IDABAND_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDABAND_ILL_INPUT`	The IDABAND solver is not compatible with the current NVECTOR module, or one of the Jacobian half-bandwidths is outside its valid range ($0 \ldots$ `N`-1).
`IDABAND_MEM_FAIL`	A memory allocation request failed.

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

`IDALapackBand`

Call `flag = IDALapackBand(ida_mem, N, mupper, mlower);`

Description The function `IDALapackBand` selects the IDABAND linear solver and indicates the use of Lapack functions.

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

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

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

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

IDAKLU

Call `flag = IDAKLU(ida_mem, N, NNZ);`

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

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`N` (`int`) problem dimension.
`NNZ` (`int`) maximum number of nonzero entries in the system Jacobian.

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

`IDASLS_SUCCESS`	The IDAKLU initialization was successful.
`IDASLS_MEM_NULL`	The `idaode_mem` pointer is `NULL`.
`IDASLS_ILL_INPUT`	The IDAKLU solver is not compatible with the current NVECTOR module.
`IDASLS_MEM_FAIL`	A memory allocation request failed.
`IDASLS_PACKAGE_FAIL`	A call to the KLU library returned a failure flag.

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

IDASuperLUMT

Call `flag = IDASuperLUMT(ida_mem, num_threads, N, NNZ);`

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

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`num_threads` (`int`) the number of threads to use when factoring/solving the linear systems. Note that SuperLU_MT is thread-parallel only in the factorization routine.
`N` (`int`) problem dimension.
`NNZ` (`int`) maximum number of nonzero entries in the system Jacobian.

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

`IDASLS_SUCCESS`	The IDASUPERLUMT initialization was successful.
`IDASLS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASLS_ILL_INPUT`	The IDASUPERLUMT solver is not compatible with the current NVECTOR module.
`IDASLS_MEM_FAIL`	A memory allocation request failed.
`IDASLS_PACKAGE_FAIL`	A call to the SuperLU_MT library returned a failure flag.

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

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

`IDASpgmr`

Call `flag = IDASpgmr(ida_mem, maxl);`

Description The function `IDASpgmr` selects the IDASPGMR linear solver.

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

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

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

`IDASPILS_SUCCESS` The IDASPGMR initialization was successful.

`IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDASPILS_MEM_FAIL` A memory allocation request failed.

`IDASpbcg`

Call `flag = IDASpbcg(ida_mem, maxl);`

Description The function `IDASpbcg` selects the IDASPBCG linear solver.

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

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

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

`IDASPILS_SUCCESS` The IDASPBCG initialization was successful.

`IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDASPILS_MEM_FAIL` A memory allocation request failed.

`IDASptfqmr`

Call `flag = IDASptfqmr(ida_mem, maxl);`

Description The function `IDASptfqmr` selects the IDASPTFQMR linear solver.

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

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

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

`IDASPILS_SUCCESS` The IDASPTFQMR initialization was successful.

`IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDASPILS_MEM_FAIL` A memory allocation request failed.

4.5.4 Initial condition calculation function

`IDACalcIC` calculates corrected initial conditions for the DAE system for certain index-one problems including a class of systems of semi-implicit form. (See §2.1 and Ref. [6].) It uses Newton iteration combined with a linesearch algorithm. Calling `IDACalcIC` is optional. It is only necessary when the initial conditions do not satisfy the given system. Thus if `y0` and `yp0` are known to satisfy $F(t_0, y_0, \dot{y}_0) = 0$, then a call to `IDACalcIC` is generally *not* necessary.

A call to the function `IDACalcIC` must be preceded by successful calls to `IDACreate` and `IDAInit` (or `IDAReInit`), and by a successful call to the linear system solver specification function. The call to `IDACalcIC` should precede the call(s) to `IDASolve` for the given problem.

IDACalcIC

Call `flag = IDACalcIC(ida_mem, icopt, tout1);`

Description The function `IDACalcIC` corrects the initial values `y0` and `yp0` at time `t0`.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`icopt` (`int`) is one of the following two options for the initial condition calculation.

`icopt=IDA_YA_YDP_INIT` directs `IDACalcIC` to compute the algebraic components of y and differential components of $\dot{y}$, given the differential components of y. This option requires that the `N_Vector id` was set through `IDASetId`, specifying the differential and algebraic components.

`icopt=IDA_Y_INIT` directs `IDACalcIC` to compute all components of y, given $\dot{y}$. In this case, `id` is not required.

`tout1` (`realtype`) is the first value of t at which a solution will be requested (from `IDASolve`). This value is needed here only to determine the direction of integration and rough scale in the independent variable t.

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

`IDA_SUCCESS`	`IDASolve` succeeded.
`IDA_MEM_NULL`	The argument `ida_mem` was `NULL`.
`IDA_NO_MALLOC`	The allocation function `IDAInit` has not been called.
`IDA_ILL_INPUT`	One of the input arguments was illegal.
`IDA_LSETUP_FAIL`	The linear solver's setup function failed in an unrecoverable manner.
`IDA_LINIT_FAIL`	The linear solver's initialization function failed.
`IDA_LSOLVE_FAIL`	The linear solver's solve function failed in an unrecoverable manner.
`IDA_BAD_EWT`	Some component of the error weight vector is zero (illegal), either for the input value of `y0` or a corrected value.
`IDA_FIRST_RES_FAIL`	The user's residual function returned a recoverable error flag on the first call, but `IDACalcIC` was unable to recover.
`IDA_RES_FAIL`	The user's residual function returned a nonrecoverable error flag.
`IDA_NO_RECOVERY`	The user's residual function, or the linear solver's setup or solve function had a recoverable error, but `IDACalcIC` was unable to recover.
`IDA_CONSTR_FAIL`	`IDACalcIC` was unable to find a solution satisfying the inequality constraints.
`IDA_LINESEARCH_FAIL`	The linesearch algorithm failed to find a solution with a step larger than `steptol` in weighted RMS norm.
`IDA_CONV_FAIL`	`IDACalcIC` failed to get convergence of the Newton iterations.

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

Note that `IDACalcIC` will correct the values of $y(t_0)$ and $\dot{y}(t_0)$ which were specified in the previous call to `IDAInit` or `IDAReInit`. To obtain the corrected values, call `IDAGetconsistentIC` (see §4.5.9.2).

4.5.5 Rootfinding initialization function

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

IDARootInit

Call `flag = IDARootInit(ida_mem, nrtfn, g);`

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

Arguments
- `ida_mem` (`void *`) pointer to the IDA memory block returned by `IDACreate`.
- `nrtfn` (`int`) is the number of root functions g_i.
- `g` (`IDARootFn`) is the C function which defines the `nrtfn` functions $g_i(t, y, \dot{y})$ whose roots are sought. See §4.6.4 for details.

Return value The return value `flag` (of type `int`) is one of
- `IDA_SUCCESS` The call to `IDARootInit` was successful.
- `IDA_MEM_NULL` The `ida_mem` argument was `NULL`.
- `IDA_MEM_FAIL` A memory allocation failed.
- `IDA_ILL_INPUT` The function `g` is `NULL`, but `nrtfn`> 0.

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

4.5.6 IDA solver function

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

IDASolve

Call `flag = IDASolve(ida_mem, tout, &tret, yret, ypret, itask);`

Description The function `IDASolve` integrates the DAE over an interval in t.

Arguments
- `ida_mem` (`void *`) pointer to the IDA memory block.
- `tout` (`realtype`) the next time at which a computed solution is desired.
- `tret` (`realtype`) the time reached by the solver (output).
- `yret` (`N_Vector`) the computed solution vector y.
- `ypret` (`N_Vector`) the computed solution vector $\dot{y}$.
- `itask` (`int`) a flag indicating the job of the solver for the next user step. The `IDA_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 approximate values of y(`tout`) and $\dot{y}$(`tout`). The `IDA_ONE_STEP` option tells the solver to just take one internal step and return the solution at the point reached by that step.

Return value `IDASolve` returns vectors `yret` and `ypret` and a corresponding independent variable value $t =$ `tret`, such that (`yret`, `ypret`) are the computed values of ($y(t)$, $\dot{y}(t)$).

In `IDA_NORMAL` mode with no errors, `tret` will be equal to `tout` and `yret` $= y$(`tout`), `ypret` $= \dot{y}$(`tout`).

The return value `flag` (of type `int`) will be one of the following:
- `IDA_SUCCESS` `IDASolve` succeeded.
- `IDA_TSTOP_RETURN` `IDASolve` succeeded by reaching the stop point specified through the optional input function `IDASetStopTime`.
- `IDA_ROOT_RETURN` `IDASolve` succeeded and found one or more roots. In this case, `tret` is the location of the root. If `nrtfn` > 1, call `IDAGetRootInfo` to see which g_i were found to have a root. See §4.5.9.3 for more information.

IDA_MEM_NULL	The ida_mem argument was NULL.
IDA_ILL_INPUT	One of the inputs to IDASolve was illegal, or some other input to the solver was either illegal or missing. The latter category includes the following situations: (a) The tolerances have not been set. (b) A component of the error weight vector became zero during internal time-stepping. (c) The linear solver initialization function (called by the user after calling IDACreate) failed to set the linear solver-specific lsolve field in ida_mem. (d) A root of one of the root functions was found both at a point t and also very near t. In any case, the user should see the printed error message for details.
IDA_TOO_MUCH_WORK	The solver took mxstep internal steps but could not reach tout. The default value for mxstep is MXSTEP_DEFAULT = 500.
IDA_TOO_MUCH_ACC	The solver could not satisfy the accuracy demanded by the user for some internal step.
IDA_ERR_FAIL	Error test failures occurred too many times (MXNEF = 10) during one internal time step or occurred with $\|h\| = h_{min}$.
IDA_CONV_FAIL	Convergence test failures occurred too many times (MXNCF = 10) during one internal time step or occurred with $\|h\| = h_{min}$.
IDA_LINIT_FAIL	The linear solver's initialization function failed.
IDA_LSETUP_FAIL	The linear solver's setup function failed in an unrecoverable manner.
IDA_LSOLVE_FAIL	The linear solver's solve function failed in an unrecoverable manner.
IDA_CONSTR_FAIL	The inequality constraints were violated and the solver was unable to recover.
IDA_REP_RES_ERR	The user's residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.
IDA_RES_FAIL	The user's residual function returned a nonrecoverable error flag.
IDA_RTFUNC_FAIL	The rootfinding function failed.

Notes

The vector yret can occupy the same space as the vector y0 of initial conditions that was passed to IDAInit, and the vector ypret can occupy the same space as yp0.

In the IDA_ONE_STEP mode, tout is used on the first call only, and 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 IDASolve failures.

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

4.5.7 Optional input functions

There are numerous optional input parameters that control the behavior of the IDA solver. IDA provides functions that can be used to change these optional input parameters from their default values. Table 4.1 lists all optional input functions in IDA which are then described in detail in the remainder of this section. For the most casual use of IDA, the reader can skip to §4.6.

We note that, on an 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.

Table 4.1: Optional inputs for IDA, IDADLS, IDASLS, and IDASPILS

Optional input	Function name	Default
IDA main solver		
Pointer to an error file	`IDASetErrFile`	`stderr`
Error handler function	`IDASetErrHandlerFn`	internal fn.
User data	`IDASetUserData`	`NULL`
Maximum order for BDF method	`IDASetMaxOrd`	5
Maximum no. of internal steps before t_{out}	`IDASetMaxNumSteps`	500
Initial step size	`IDASetInitStep`	estimated
Maximum absolute step size	`IDASetMaxStep`	∞
Value of t_{stop}	`IDASetStopTime`	∞
Maximum no. of error test failures	`IDASetMaxErrTestFails`	10
Maximum no. of nonlinear iterations	`IDASetMaxNonlinIters`	4
Maximum no. of convergence failures	`IDASetMaxConvFails`	10
Maximum no. of error test failures	`IDASetMaxErrTestFails`	7
Coeff. in the nonlinear convergence test	`IDASetNonlinConvCoef`	0.33
Suppress alg. vars. from error test	`IDASetSuppressAlg`	`FALSE`
Variable types (differential/algebraic)	`IDASetId`	`NULL`
Inequality constraints on solution	`IDASetConstraints`	`NULL`
Direction of zero-crossing	`IDASetRootDirection`	both
Disable rootfinding warnings	`IDASetNoInactiveRootWarn`	none
IDA initial conditions calculation		
Coeff. in the nonlinear convergence test	`IDASetNonlinConvCoefIC`	0.0033
Maximum no. of steps	`IDASetMaxNumStepsIC`	5
Maximum no. of Jacobian/precond. evals.	`IDASetMaxNumJacsIC`	4
Maximum no. of Newton iterations	`IDASetMaxNumItersIC`	10
Turn off linesearch	`IDASetLineSearchOffIC`	`FALSE`
Lower bound on Newton step	`IDASetStepToleranceIC`	$\text{uround}^{2/3}$
IDADLS linear solvers		
Dense Jacobian function	`IDADlsSetDenseJacFn`	DQ
Band Jacobian function	`IDADlsSetBandJacFn`	DQ
IDASLS linear solvers		
Sparse Jacobian function	`IDASlsSetSparseJacFn`	none
Sparse matrix ordering algorithm	`IDAKLUSetOrdering`	1 for `COLAMD`
Sparse matrix ordering algorithm	`IDASuperLUMTSetOrdering`	3 for `COLAMD`
IDASPILS linear solvers		
Preconditioner functions	`IDASpilsSetPreconditioner`	NULL, NULL
Jacobian-times-vector function	`IDASpilsSetJacTimesVecFn`	DQ
Factor in linear convergence test	`IDASpilsSetEpsLin`	0.05
Factor in DQ increment calculation	`IDASpilsSetIncrementFactor`	1.0
Maximum no. of restarts (IDASPGMR)	`IDASpilsSetMaxRestarts`	5
Type of Gram-Schmidt orthogonalization [(a)]	`IDASpilsSetGSType`	classical GS
Maximum Krylov subspace size[(b)]	`IDASpilsSetMaxl`	5

[(a)] Only for IDASPGMR

[(b)] Only for IDASPBCG and IDASPTFQMR

4.5.7.1 Main solver optional input functions

The calls listed here can be executed in any order. However, if the user's program calls either IDASetErrFile or IDASetErrHandlerFn, then that call should appear first, in order to take effect for any later error message.

IDASetErrFile

Call `flag = IDASetErrFile(ida_mem, errfp);`

Description The function IDASetErrFile specifies the pointer to the file where all IDA messages should be directed when the default IDA error handler function is used.

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

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

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

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

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

IDASetErrHandlerFn

Call `flag = IDASetErrHandlerFn(ida_mem, ehfun, eh_data);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`ehfun` (`IDAErrHandlerFn`) is the user's C error handler function (see §4.6.2).
`eh_data` (`void *`) pointer to user data passed to `ehfun` every time it is called.

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The function `ehfun` and data pointer `eh_data` have been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

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

IDASetUserData

Call `flag = IDASetUserData(ida_mem, user_data);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`user_data` (`void *`) pointer to the user data.

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

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

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

`IDASetMaxOrd`

Call `flag = IDASetMaxOrd(ida_mem, maxord);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`maxord` (`int`) value of the maximum method order. This must be positive.

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

`IDA_SUCCESS` The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDA_ILL_INPUT` The input value `maxord` is ≤ 0, or larger than its previous value.

Notes The default value is 5. If the input value exceeds 5, the value 5 will be used. Since `maxord` affects the memory requirements for the internal IDA memory block, its value cannot be increased past its previous value.

`IDASetMaxNumSteps`

Call `flag = IDASetMaxNumSteps(ida_mem, mxsteps);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`mxsteps` (`long int`) maximum allowed number of steps.

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

`IDA_SUCCESS` The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

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

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

`IDASetInitStep`

Call `flag = IDASetInitStep(ida_mem, hin);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`hin` (`realtype`) value of the initial step size to be attempted. Pass 0.0 to have IDA use the default value.

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

`IDA_SUCCESS` The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

Notes By default, IDA estimates the initial step as the solution of $\|h\dot{y}\|_{\mathrm{WRMS}} = 1/2$, with an added restriction that $|h| \leq .001|$`tout - t0`$|$.

`IDASetMaxStep`

Call `flag = IDASetMaxStep(ida_mem, hmax);`

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

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

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT` Either `hmax` is not positive or it is smaller than the minimum allowable step.

Notes Pass `hmax`$= 0$ to obtain the default value ∞.

`IDASetStopTime`

Call `flag = IDASetStopTime(ida_mem, tstop);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`tstop` (`realtype`) value of the independent variable past which the solution should not proceed.

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT` The value of `tstop` is not beyond the current t value, t_n.

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

`IDASetMaxErrTestFails`

Call `flag = IDASetMaxErrTestFails(ida_mem, maxnef);`

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

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

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

Notes The default value is 7.

`IDASetMaxNonlinIters`

Call `flag = IDASetMaxNonlinIters(ida_mem, maxcor);`

Description The function `IDASetMaxNonlinIters` specifies the maximum number of nonlinear solver iterations at one step.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`maxcor` (`int`) maximum number of nonlinear solver iterations allowed on one step (> 0).

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

`IDA_SUCCESS` The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

Notes The default value is 3.

`IDASetMaxConvFails`

Call `flag = IDASetMaxConvFails(ida_mem, maxncf);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

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

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

`IDA_SUCCESS` The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

Notes The default value is 10.

`IDASetNonlinConvCoef`

Call `flag = IDASetNonlinConvCoef(ida_mem, nlscoef);`

Description The function `IDASetNonlinConvCoef` specifies the safety factor in the nonlinear convergence test; see Chapter 2, Eq. (2.7).

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

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

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

`IDA_SUCCESS` The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDA_ILL_INPUT` The value of `nlscoef` is $<= 0.0$.

Notes The default value is 0.33.

`IDASetSuppressAlg`

Call `flag = IDASetSuppressAlg(ida_mem, suppressalg);`

Description The function `IDASetSuppressAlg` indicates whether or not to suppress algebraic variables in the local error test.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`suppressalg` (`booleantype`) indicates whether to suppress (`TRUE`) or not (`FALSE`) the algebraic variables in the local error test.

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

`IDA_SUCCESS` The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

Notes The default value is `FALSE`.

If `suppressalg`=`TRUE` is selected, then the `id` vector must be set (through `IDASetId`) to specify the algebraic components.

In general, the use of this option (with `suppressalg = TRUE`) is *discouraged* when solving DAE systems of index 1, whereas it is generally *encouraged* for systems of index 2 or more. See pp. 146-147 of Ref. [3] for more on this issue.

`IDASetId`

Call `flag = IDASetId(ida_mem, id);`

Description The function `IDASetId` specifies algebraic/differential components in the y vector.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`id` (`N_Vector`) state vector. A value of 1.0 indicates a differential variable, while 0.0 indicates an algebraic variable.

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

`IDA_SUCCESS` The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

Notes The vector `id` is required if the algebraic variables are to be suppressed from the local error test (see `IDASetSuppressAlg`) or if `IDACalcIC` is to be called with `icopt = IDA_YA_YDP_INIT` (see §4.5.4).

`IDASetConstraints`

Call `flag = IDASetConstraints(ida_mem, constraints);`

Description The function `IDASetConstraints` specifies a vector defining inequality constraints for each component of the solution vector y.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`constraints` (`N_Vector`) vector of constraint flags. If `constraints[i]` is

- 0.0 then no constraint is imposed on y_i.
- 1.0 then y_i will be constrained to be $y_i \geq 0.0$.
- −1.0 then y_i will be constrained to be $y_i \leq 0.0$.
- 2.0 then y_i will be constrained to be $y_i > 0.0$.
- −2.0 then y_i will be constrained to be $y_i < 0.0$.

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

`IDA_SUCCESS` The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDA_ILL_INPUT` The constraints vector contains illegal values.

Notes The presence of a non-`NULL` constraints vector that is not 0.0 in all components will cause constraint checking to be performed. However, a call with 0.0 in all components of `constraints` will result in an illegal input return.

4.5.7.2 Dense/band direct linear solvers optional input functions

The IDADENSE solver needs a function to compute a dense approximation to the Jacobian matrix $J(t, y, \dot{y})$. This function must be of type `IDADlsDenseJacFn`. The user can supply his/her own dense Jacobian function, or use the default internal difference quotient approximation that comes with the IDADENSE solver. To specify a user-supplied Jacobian function `djac`, IDADENSE provides the function `IDADlsSetDenseJacFn`. The IDADENSE solver passes the pointer `user_data` to the dense Jacobian function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `user_data` may be specified through `IDASetUserData`.

`IDADlsSetDenseJacFn`

Call `flag = IDADlsSetDenseJacFn(ida_mem, djac);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`djac` (`IDADlsDenseJacFn`) user-defined dense Jacobian approximation function.

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

`IDADLS_SUCCESS`	The optional value has been successfully set.
`IDADLS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDADLS_LMEM_NULL`	The IDADENSE linear solver has not been initialized.

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

The function type `IDADlsDenseJacFn` is described in §4.6.5.

The IDABAND solver needs a function to compute a banded approximation to the Jacobian matrix $J(t, y, \dot{y})$. This function must be of type `IDADlsBandJacFn`. The user can supply his/her own banded Jacobian approximation function, or use the default difference quotient function that comes with the IDABAND solver. To specify a user-supplied Jacobian function `bjac`, IDABAND provides the function `IDADlsSetBandJacFn`. The IDABAND solver passes the pointer `user_data` to the banded Jacobian approximation function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `user_data` may be specified through `IDASetUserData`.

`IDADlsSetBandJacFn`

Call `flag = IDADlsSetBandJacFn(ida_mem, bjac);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`bjac` (`IDADlsBandJacFn`) user-defined banded Jacobian approximation function.

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

`IDADLS_SUCCESS`	The optional value has been successfully set.
`IDADLS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDADLS_LMEM_NULL`	The IDABAND linear solver has not been initialized.

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

The function type `IDADlsBandJacFn` is described in §4.6.6.

4.5.7.3 Sparse direct linear solvers optional input functions

The IDAKLU and IDASUPERLUMT solvers require a function to compute a compressed-sparse-column approximation ot the Jacobian matrix $J(t, y, \dot{y})$. This function must be of type `IDASlsSparseJacFn`. The user must supply a custom sparse Jacobian function since a difference quotient approximation would not leverage the underlying sparse matrix structure of the problem. To specify a user-supplied Jacobian function `sjac`, IDAKLU and IDASUPERLUMT provide the function `IDASlsSetSparseJacFn`. The IDAKLU and IDASUPERLUMT solvers pass the pointer `user_data` to the sparse Jacobian function. This mechanism allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `user_data` may be specified through `IDASetUserData`.

`IDASlsSetSparseJacFn`

Call `flag = IDASlsSetSparseJacFn(ida_mem, sjac);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`sjac` (`IDASlsSparseJacFn`) user-defined sparse Jacobian approximation function.

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

`IDASLS_SUCCESS`	The optional value has been successfully set.
`IDASLS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASLS_LMEM_NULL`	The IDAKLU or IDASUPERLUMT linear solver has not been initialized.

Notes The function type `IDASlsSparseJacFn` is described in §4.6.7.

When using a sparse direct solver, there may be instances when the number of state variables does not change, but the number of nonzeroes in the Jacobian does change. In this case, for the IDAKLU solver, we provide the following reinitialization function. This function reinitializes the Jacobian matrix memory for the new number of nonzeroes and sets flags for a new factorization (symbolic and numeric) to be conducted at the next solver setup call. This routine is useful in the cases where the number of nonzeroes has changed, or where the structure of the linear system has changed, requiring a new symbolic (and numeric) factorization.

`IDAKLUReInit`

Call `flag = IDAKLUReInit(ida_mem, n, nnz, reinit_type);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`n` (`int`) number of state variables in the system.
`nnz` (`int`) number of nonzeroes in the Jacobian matrix.
`reinit_type` (`int`) type of reinitialization:

1. The Jacobian matrix will be destroyed and a new one will be allocated based on the `nnz` value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.
2. Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of `nnz` given in the prior call to IDAKLU.

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

`IDASLS_SUCCESS`	The reinitialization succeeded.
`IDASLS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASLS_LMEM_NULL`	The IDAKLU linear solver has not been initialized.
`IDASLS_ILL_INPUT`	The given `reinit_type` has an illegal value.
`IDASLS_MEM_FAIL`	A memory allocation failed.

Notes The default value for `reinit_type` is 2.

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

`IDAKLUSetOrdering`

Call `flag = IDAKLUSetOrdering(ida_mem, ordering_choice);`

Description The function `IDAKLUSetOrdering` specifies the ordering algorithm used by IDAKLU for reducing fill.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`ordering_choice` (`int`) flag denoting algorithm choice:
0 `AMD`
1 `COLAMD`
2 natural ordering

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

`IDASLS_SUCCESS`	The optional value has been successfully set.
`IDASLS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASLS_ILL_INPUT`	The supplied value of `ordering_choice` is illegal.

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

`IDASuperLUMTSetOrdering`

Call `flag = IDASuperLUMTSetOrdering(ida_mem, ordering_choice);`

Description The function `IDASuperLUMTSetOrdering` specifies the ordering algorithm used by IDASUPERLUMT for reducing fill.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`ordering_choice` (`int`) flag denoting algorithm choice:
0 natural ordering
1 minimal degree ordering on $J^T J$
2 minimal degree ordering on $J^T + J$
3 `COLAMD`

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

`IDASLS_SUCCESS`	The optional value has been successfully set.
`IDASLS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASLS_ILL_INPUT`	The supplied value of `ordering_choice` is illegal.

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

4.5.7.4 Iterative linear solvers optional input functions

If preconditioning is to be done with one of the IDASPILS linear solvers, then the user must supply a preconditioner solve function `psolve` and specify its name through a call to `IDASpilsSetPreconditioner`. The evaluation and preprocessing of any Jacobian-related data needed by the user's preconditioner solve function is done in the optional user-supplied function `psetup`. Both of these functions are fully specified in §4.6. If used, the name of the `psetup` function should be specified in the call to `IDASpilsSetPreconditioner`.

The pointer `user_data` received through `IDASetUserData` (or a pointer to `NULL` if `user_data` was not specified) is passed to the preconditioner `psetup` and `psolve` functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program.

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

IDASpilsSetPreconditioner

Call `flag = IDASpilsSetPreconditioner(ida_mem, psetup, psolve);`

Description The function `IDASpilsSetPreconditioner` specifies the preconditioner setup and solve functions.

Arguments
- `ida_mem` (`void *`) pointer to the IDA memory block.
- `psetup` (`IDASpilsPrecSetupFn`) user-defined preconditioner setup function. Pass `NULL` if no setup is to be done.
- `psolve` (`IDASpilsPrecSolveFn`) user-defined preconditioner solve function.

Return value The return value `flag` (of type `int`) is one of
- `IDASPILS_SUCCESS` The optional values have been successfully set.
- `IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`.
- `IDASPILS_LMEM_NULL` The IDASPILS linear solver has not been initialized.

Notes The function type `IDASpilsPrecSolveFn` is described in §4.6.9. The function type `IDASpilsPrecSetupFn` is described in §4.6.10.

IDASpilsSetJacTimesVecFn

Call `flag = IDASpilsSetJacTimesVecFn(ida_mem, jtimes);`

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

Arguments
- `ida_mem` (`void *`) pointer to the IDA memory block.
- `jtimes` (`IDASpilsJacTimesVecFn`) user-defined Jacobian-vector product function.

Return value The return value `flag` (of type `int`) is one of
- `IDASPILS_SUCCESS` The optional value has been successfully set.
- `IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`.
- `IDASPILS_LMEM_NULL` The IDASPILS linear solver has not been initialized.

Notes By default, the IDASPILS solvers use the difference quotient function. If `NULL` is passed to `jtimes`, this default function is used.

The function type `IDASpilsJacTimesVecFn` is described in §4.6.8.

IDASpilsSetGSType

Call `flag = IDASpilsSetGSType(ida_mem, gstype);`

Description The function `IDASpilsSetGSType` specifies the Gram-Schmidt orthogonalization to be used. This must be one of the enumeration constants `MODIFIED_GS` or `CLASSICAL_GS`. These correspond to using modified Gram-Schmidt and classical Gram-Schmidt, respectively.

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

Return value The return value `flag` (of type `int`) is one of
- `IDASPILS_SUCCESS` The optional value has been successfully set.
- `IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`.
- `IDASPILS_LMEM_NULL` The IDASPILS linear solver has not been initialized.
- `IDASPILS_ILL_INPUT` The value of `gstype` is not valid.

Notes The default value is `MODIFIED_GS`.

This option is available only for the IDASPGMR linear solver.

`IDASpilsSetMaxRestarts`

Call `flag = IDASpilsSetMaxRestarts(ida_mem, maxrs);`

Description The function `IDASpilsSetMaxRestarts` specifies the maximum number of restarts to be used in the GMRES algorithm.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`maxrs` (`int`) maximum number of restarts.

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

`IDASPILS_SUCCESS`	The optional value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.
`IDASPILS_ILL_INPUT`	The `maxrs` argument is negative.

Notes The default value is 5. Pass `maxrs` $= 0$ to specify no restarts.

This option is available only for the IDASPGMR linear solver.

`IDASpilsSetEpsLin`

Call `flag = IDASpilsSetEpsLin(ida_mem, eplifac);`

Description The function `IDASpilsSetEpsLin` specifies the factor by which the Krylov linear solver's convergence test constant is reduced from the Newton iteration test constant. (See Chapter 2).

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`eplifac` (`realtype`) linear convergence safety factor (>= 0.0).

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

`IDASPILS_SUCCESS`	The optional value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.
`IDASPILS_ILL_INPUT`	The value of `eplifac` is negative.

Notes The default value is 0.05.

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

`IDASpilsSetIncrementFactor`

Call `flag = IDASpilsSetIncrementFactor(ida_mem, dqincfac);`

Description The function `IDASpilsSetIncrementFactor` specifies a factor in the increments to y used in the difference quotient approximations to the Jacobian-vector products. (See Chapter 2). The increment used to approximate Jv will be $\sigma =$ `dqincfac`$/\|v\|$.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`dqincfac` (`realtype`) difference quotient increment factor.

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

`IDASPILS_SUCCESS`	The optional value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.
`IDASPILS_ILL_INPUT`	The increment factor was non-positive.

Notes The default value is `dqincfac` $= 1.0$.

IDASpilsSetMaxl

Call `flag = IDASpilsSetMaxl(ida_mem, maxl);`

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

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

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

`IDASPILS_SUCCESS`	The optional value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.

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

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

This option is available only for the IDASPBCG and IDASPTFQMR linear solvers.

4.5.7.5 Initial condition calculation optional input functions

The following functions can be called just prior to calling `IDACalcIC` to set optional inputs controlling the initial condition calculation.

IDASetNonlinConvCoefIC

Call `flag = IDASetNonlinConvCoefIC(ida_mem, epiccon);`

Description The function `IDASetNonlinConvCoefIC` specifies the positive constant in the Newton iteration convergence test within the initial condition calculation.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`epiccon` (`realtype`) coefficient in the Newton convergence test (> 0).

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

`IDA_SUCCESS`	The optional value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT`	The `epiccon` factor is <= 0.0.

Notes The default value is $0.01 \cdot 0.33$.

This test uses a weighted RMS norm (with weights defined by the tolerances). For new initial value vectors y and $\dot{y}$ to be accepted, the norm of $J^{-1}F(t_0, y, \dot{y})$ must be $\leq$ `epiccon`, where J is the system Jacobian.

IDASetMaxNumStepsIC

Call `flag = IDASetMaxNumStepsIC(ida_mem, maxnh);`

Description The function `IDASetMaxNumStepsIC` specifies the maximum number of steps allowed when `icopt`=`IDA_YA_YDP_INIT` in `IDACalcIC`, where h appears in the system Jacobian, $J = \partial F/\partial y + (1/h)\partial F/\partial \dot{y}$.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`maxnh` (`int`) maximum allowed number of values for h.

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

`IDA_SUCCESS`	The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDA_ILL_INPUT` `maxnh` is non-positive.

Notes The default value is 5.

`IDASetMaxNumJacsIC`

Call `flag = IDASetMaxNumJacsIC(ida_mem, maxnj);`

Description The function `IDASetMaxNumJacsIC` specifies the maximum number of the approximate Jacobian or preconditioner evaluations allowed when the Newton iteration appears to be slowly converging.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`maxnj` (`int`) maximum allowed number of Jacobian or preconditioner evaluations.

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

`IDA_SUCCESS` The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDA_ILL_INPUT` `maxnj` is non-positive.

Notes The default value is 4.

`IDASetMaxNumItersIC`

Call `flag = IDASetMaxNumItersIC(ida_mem, maxnit);`

Description The function `IDASetMaxNumItersIC` specifies the maximum number of Newton iterations allowed in any one attempt to solve the initial conditions calculation problem.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`maxnit` (`int`) maximum number of Newton iterations.

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

`IDA_SUCCESS` The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDA_ILL_INPUT` `maxnit` is non-positive.

Notes The default value is 10.

`IDASetLineSearchOffIC`

Call `flag = IDASetLineSearchOffIC(ida_mem, lsoff);`

Description The function `IDASetLineSearchOffIC` specifies whether to turn on or off the linesearch algorithm.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`lsoff` (`booleantype`) a flag to turn off (`TRUE`) or keep (`FALSE`) the linesearch algorithm.

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

`IDA_SUCCESS` The optional value has been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

Notes The default value is `FALSE`.

`IDASetStepToleranceIC`

Call `flag = IDASetStepToleranceIC(ida_mem, steptol);`

Description The function `IDASetStepToleranceIC` specifies a positive lower bound on the Newton step.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`steptol` (`int`) Minimum allowed WRMS-norm of the Newton step (> 0.0).

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

`IDA_SUCCESS`	The optional value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT`	The `steptol` tolerance is <= 0.0.

Notes The default value is (unit roundoff)$^{2/3}$.

4.5.7.6 Rootfinding optional input functions

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

`IDASetRootDirection`

Call `flag = IDASetRootDirection(ida_mem, rootdir);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

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

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

`IDA_SUCCESS`	The optional value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT`	rootfinding has not been activated through a call to `IDARootInit`.

Notes The default behavior is to locate both zero-crossing directions.

`IDASetNoInactiveRootWarn`

Call `flag = IDASetNoInactiveRootWarn(ida_mem);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

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

`IDA_SUCCESS`	The optional value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.

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

4.5.8 Interpolated output function

An optional function `IDAGetDky` is available to obtain additional output values. This function must be called after a successful return from `IDASolve` and provides interpolated values of y or its derivatives of order up to the last internal order used for any value of t in the last internal step taken by IDA.

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

`IDAGetDky`

Call `flag = IDAGetDky(ida_mem, t, k, dky);`

Description The function `IDAGetDky` computes the interpolated values of the k^{th} derivative of y for any value of t in the last internal step taken by IDA. The value of k must be non-negative and smaller than the last internal order used. A value of 0 for k means that the y is interpolated. The value of t must satisfy $t_n - h_u \leq t \leq t_n$, where t_n denotes the current internal time reached, and h_u is the last internal step size used successfully.

Arguments
- `ida_mem` (`void *`) pointer to the IDA memory block.
- `t` (`realtype`) time at which to interpolate.
- `k` (`int`) integer specifying the order of the derivative of y wanted.
- `dky` (`N_Vector`) vector containing the interpolated k^{th} derivative of $y(t)$.

Return value The return value `flag` (of type `int`) is one of
- `IDA_SUCCESS` `IDAGetDky` succeeded.
- `IDA_MEM_NULL` The `ida_mem` argument was `NULL`.
- `IDA_BAD_T` `t` is not in the interval $[t_n - h_u, t_n]$.
- `IDA_BAD_K` `k` is not one of $\{0, 1, \ldots, klast\}$.
- `IDA_BAD_DKY` `dky` is `NULL`.

Notes It is only legal to call the function `IDAGetDky` after a successful return from `IDASolve`. Functions `IDAGetCurrentTime`, `IDAGetLastStep` and `IDAGetLastOrder` (see §4.5.9.1) can be used to access t_n, h_u and $klast$.

4.5.9 Optional output functions

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

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

4.5.9.1 Main solver optional output functions

IDA 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

Table 4.2: Optional outputs from IDA, IDADLS, IDASLS, and IDASPILS

Optional output	Function name
IDA main solver	
Size of IDA real and integer workspace	IDAGetWorkSpace
Cumulative number of internal steps	IDAGetNumSteps
No. of calls to residual function	IDAGetNumResEvals
No. of calls to linear solver setup function	IDAGetNumLinSolvSetups
No. of local error test failures that have occurred	IDAGetNumErrTestFails
Order used during the last step	IDAGetLastOrder
Order to be attempted on the next step	IDAGetCurrentOrder
Order reductions due to stability limit detection	IDAGetNumStabLimOrderReds
Actual initial step size used	IDAGetActualInitStep
Step size used for the last step	IDAGetLastStep
Step size to be attempted on the next step	IDAGetCurrentStep
Current internal time reached by the solver	IDAGetCurrentTime
Suggested factor for tolerance scaling	IDAGetTolScaleFactor
Error weight vector for state variables	IDAGetErrWeights
Estimated local errors	IDAGetEstLocalErrors
No. of nonlinear solver iterations	IDAGetNumNonlinSolvIters
No. of nonlinear convergence failures	IDAGetNumNonlinSolvConvFails
Array showing roots found	IDAGetRootInfo
No. of calls to user root function	IDAGetNumGEvals
Name of constant associated with a return flag	IDAGetReturnFlagName
IDA initial conditions calculation	
Number of backtrack operations	IDAGetNumBacktrackops
Corrected initial conditions	IDAGetConsistentIC
IDADLS linear solver	
Size of real and integer workspace	IDADlsGetWorkSpace
No. of Jacobian evaluations	IDADlsGetNumJacEvals
No. of residual calls for finite diff. Jacobian evals.	IDADlsGetNumResEvals
Last return from a linear solver function	IDADlsGetLastFlag
Name of constant associated with a return flag	IDADlsGetReturnFlagName
IDASLS linear solver	
No. of Jacobian evaluations	IDASlsGetNumJacEvals
Last return from a linear solver function	IDASlsGetLastFlag
Name of constant associated with a return flag	IDASlsGetReturnFlagName
IDASPILS linear solvers	
Size of real and integer workspace	IDASpilsGetWorkSpace
No. of linear iterations	IDASpilsGetNumLinIters
No. of linear convergence failures	IDASpilsGetNumConvFails
No. of preconditioner evaluations	IDASpilsGetNumPrecEvals
No. of preconditioner solves	IDASpilsGetNumPrecSolves
No. of Jacobian-vector product evaluations	IDASpilsGetNumJtimesEvals
No. of residual calls for finite diff. Jacobian-vector evals.	IDASpilsGetNumResEvals
Last return from a linear solver function	IDASpilsGetLastFlag
Name of constant associated with a return flag	IDASpilsGetReturnFlagName

as additional data from the IDA 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 IDA nonlinear solver being used. As a convenience, additional extraction functions provide the optional outputs in groups. These optional output functions are described next.

`IDAGetWorkSpace`

Call `flag = IDAGetWorkSpace(ida_mem, &lenrw, &leniw);`

Description The function `IDAGetWorkSpace` returns the IDA real and integer workspace sizes.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`lenrw` (`long int`) number of real values in the IDA workspace.
`leniw` (`long int`) number of integer values in the IDA workspace.

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

`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

Notes In terms of the problem size N, the maximum method order `maxord`, and the number `nrtfn` of root functions (see §4.5.5), the actual size of the real workspace, in `realtype` words, is given by the following:

- base value: `lenrw` $= 55 + (m + 6) * N_r + 3$*`nrtfn`;
- with `IDASVtolerances`: `lenrw` $=$ `lenrw` $+N_r$;
- with constraint checking (see `IDASetConstraints`): `lenrw` $=$ `lenrw` $+N_r$;
- with `id` specified (see `IDASetId`): `lenrw` $=$ `lenrw` $+N_r$;

where $m = \max($`maxord`$, 3)$, and N_r is the number of real words in one `N_Vector` ($\approx N$).

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

- base value: `leniw` $= 38 + (m + 6) * N_i$ + `nrtfn`;
- with `IDASVtolerances`: `leniw` $=$ `leniw` $+N_i$;
- with constraint checking: `lenrw` $=$ `lenrw` $+N_i$;
- with `id` specified: `lenrw` $=$ `lenrw` $+N_i$;

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

For the default value of `maxord`, with no rootfinding, no `id`, no constraints, and with no call to `IDASVtolerances`, these lengths are given roughly by: `lenrw` $= 55 + 11N$, `leniw` $= 49$.

`IDAGetNumSteps`

Call `flag = IDAGetNumSteps(ida_mem, &nsteps);`

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

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

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

`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetNumResEvals`

Call `flag = IDAGetNumResEvals(ida_mem, &nrevals);`

Description The function `IDAGetNumResEvals` returns the number of calls to the user's residual evaluation function.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`nrevals` (`long int`) number of calls to the user's `res` function.

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

Notes The `nrevals` value returned by `IDAGetNumResEvals` does not account for calls made to `res` from a linear solver or preconditioner module.

`IDAGetNumLinSolvSetups`

Call `flag = IDAGetNumLinSolvSetups(ida_mem, &nlinsetups);`

Description The function `IDAGetNumLinSolvSetups` returns the cumulative number of calls made to the linear solver's setup function (total so far).

Arguments `ida_mem` (`void *`) pointer to the IDA 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
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetNumErrTestFails`

Call `flag = IDAGetNumErrTestFails(ida_mem, &netfails);`

Description The function `IDAGetNumErrTestFails` returns the cumulative number of local error test failures that have occurred (total so far).

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

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetLastOrder`

Call `flag = IDAGetLastOrder(ida_mem, &klast);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`klast` (`int`) method order used on the last internal step.

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetCurrentOrder`

Call `flag = IDAGetCurrentOrder(ida_mem, &kcur);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`kcur` (`int`) method order to be used on the next internal step.

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetLastStep`

Call `flag = IDAGetLastStep(ida_mem, &hlast);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`hlast` (`realtype`) step size taken on the last internal step by IDA, or last artificial step size used in `IDACalcIC`, whichever was called last.

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetCurrentStep`

Call `flag = IDAGetCurrentStep(ida_mem, &hcur);`

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

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

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetActualInitStep`

Call `flag = IDAGetActualInitStep(ida_mem, &hinused);`

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

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

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

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

`IDAGetCurrentTime`

Call `flag = IDAGetCurrentTime(ida_mem, &tcur);`

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

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

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetTolScaleFactor`

Call `flag = IDAGetTolScaleFactor(ida_mem, &tolsfac);`

Description The function `IDAGetTolScaleFactor` 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 `ida_mem` (`void *`) pointer to the IDA memory block.
`tolsfac` (`realtype`) suggested scaling factor for user tolerances.

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetErrWeights`

Call `flag = IDAGetErrWeights(ida_mem, eweight);`

Description The function `IDAGetErrWeights` returns the solution error weights at the current time. These are the W_i given by Eq. (2.6) (or by the user's `IDAEwtFn`).

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

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

Notes The user must allocate space for `eweight`.

`IDAGetEstLocalErrors`

Call `flag = IDAGetEstLocalErrors(ida_mem, ele);`

Description The function `IDAGetEstLocalErrors` returns the estimated local errors.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`ele` (`N_Vector`) estimated local errors at the current time.

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

Notes The user must allocate space for `ele`.
The values returned in `ele` are only valid if `IDASolve` returned a non-negative value.

The `ele` vector, togther with the `eweight` vector from `IDAGetErrWeights`, can be used to determine how the various components of the system contributed to the estimated

local error test. Specifically, that error test uses the RMS norm of a vector whose components are the products of the components of these two vectors. Thus, for example, if there were recent error test failures, the components causing the failures are those with largest values for the products, denoted loosely as `eweight[i]*ele[i]`.

`IDAGetIntegratorStats`

Call `flag = IDAGetIntegratorStats(ida_mem, &nsteps, &nrevals, &nlinsetups, &netfails, &klast, &kcur, &hinused, &hlast, &hcur, &tcur);`

Description The function `IDAGetIntegratorStats` returns the IDA integrator statistics as a group.

Arguments

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

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

`IDA_SUCCESS`	the optional output values have been successfully set.
`IDA_MEM_NULL`	the `ida_mem` pointer is `NULL`.

`IDAGetNumNonlinSolvIters`

Call `flag = IDAGetNumNonlinSolvIters(ida_mem, &nniters);`

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

Arguments

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

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

`IDA_SUCCESS`	The optional output value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.

`IDAGetNumNonlinSolvConvFails`

Call `flag = IDAGetNumNonlinSolvConvFails(ida_mem, &nncfails);`

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

Arguments

`ida_mem`	(`void *`) pointer to the IDA memory block.
`nncfails`	(`long int`) number of nonlinear convergence failures.

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

`IDA_SUCCESS`	The optional output value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.

`IDAGetNonlinSolvStats`

Call `flag = IDAGetNonlinSolvStats(ida_mem, &nniters, &nncfails);`

Description The function `IDAGetNonlinSolvStats` returns the IDA nonlinear solver statistics as a group.

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

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetReturnFlagName`

Call `name = IDAGetReturnFlagName(flag);`

Description The function `IDAGetReturnFlagName` returns the name of the IDA constant corresponding to `flag`.

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

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

4.5.9.2 Initial condition calculation optional output functions

`IDAGetNumBcktrackOps`

Call `flag = IDAGetNumBacktrackOps(ida_mem, &nbacktr);`

Description The function `IDAGetNumBacktrackOps` returns the number of backtrack operations done in the linesearch algorithm in `IDACalcIC`.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`nbacktr` (`long int`) the cumulative number of backtrack operations.

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetConsistentIC`

Call `flag = IDAGetConsistentIC(ida_mem, yy0_mod, yp0_mod);`

Description The function `IDAGetConsistentIC` returns the corrected initial conditions calculated by `IDACalcIC`.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`yy0_mod` (`N_Vector`) consistent solution vector.
`yp0_mod` (`N_Vector`) consistent derivative vector.

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfully set.
`IDA_ILL_INPUT` The function was not called before the first call to `IDASolve`.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

Notes If the consistent solution vector or consistent derivative vector is not desired, pass `NULL` for the corresponding argument.

The user must allocate space for `yy0_mod` and `yp0_mod` (if not `NULL`).

4.5.9.3 Rootfinding optional output functions

There are two optional output functions associated with rootfinding.

`IDAGetRootInfo`

Call `flag = IDAGetRootInfo(ida_mem, rootsfound);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

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

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

`IDA_SUCCESS` The optional output values have been successfully set.

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

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

The user must allocate memory for the vector `rootsfound`.

`IDAGetNumGEvals`

Call `flag = IDAGetNumGEvals(ida_mem, &ngevals);`

Description The function `IDAGetNumGEvals` returns the cumulative number of calls to the user root function g.

Arguments `ida_mem` (`void *`) pointer to the IDA 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

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

`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

4.5.9.4 Dense/band direct linear solvers optional output functions

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

`IDADlsGetWorkSpace`

Call `flag = IDADlsGetWorkSpace(ida_mem, &lenrwLS, &leniwLS);`

Description The function `IDADlsGetWorkSpace` returns the sizes of the real and integer workspaces used by an IDADLS linear solver (IDADENSE or IDABAND).

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`lenrwLS` (`long int`) the number of real values in the IDADLS workspace.

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

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

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

IDADLS_MEM_NULL The ida_mem pointer is NULL.

IDADLS_LMEM_NULL The IDADLS linear solver has not been initialized.

Notes For the IDADENSE linear solver, in terms of the problem size N, the actual size of the real workspace is $2N^2$ realtype words, while the actual size of the integer workspace is N integer words. For the IDABAND linear solver, in terms of N and Jacobian half-bandwidths, the actual size of the real workspace is N (2 mupper+3 mlower +2) realtype words, while the actual size of the integer workspace is N integer words.

IDADlsGetNumJacEvals

Call `flag = IDADlsGetNumJacEvals(ida_mem, &njevals);`

Description The function IDADlsGetNumJacEvals returns the cumulative number of calls to the IDADLS (dense or banded) Jacobian approximation function.

Arguments ida_mem (void *) pointer to the IDA memory block.

njevals (long int) the cumulative number of calls to the Jacobian function (total so far).

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

IDADLS_SUCCESS The optional output value has been successfully set.

IDADLS_MEM_NULL The ida_mem pointer is NULL.

IDADLS_LMEM_NULL The IDADENSE linear solver has not been initialized.

IDADlsGetNumResEvals

Call `flag = IDADlsGetNumResEvals(ida_mem, &nrevalsLS);`

Description The function IDADlsGetNumResEvals returns the cumulative number of calls to the user residual function due to the finite difference (dense or band) Jacobian approximation.

Arguments ida_mem (void *) pointer to the IDA memory block.

nrevalsLS (long int) the cumulative number of calls to the user residual function.

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

IDADLS_SUCCESS The optional output value has been successfully set.

IDADLS_MEM_NULL The ida_mem pointer is NULL.

IDADLS_LMEM_NULL The IDADENSE linear solver has not been initialized.

Notes The value nrevalsLS is incremented only if the default internal difference quotient function is used.

IDADlsGetLastFlag

Call `flag = IDADlsGetLastFlag(ida_mem, &lsflag);`

Description The function IDADlsGetLastFlag returns the last return value from an IDADLS routine.

Arguments ida_mem (void *) pointer to the IDA memory block.

lsflag (long int) the value of the last return flag from an IDADLS function.

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

IDADLS_SUCCESS The optional output value has been successfully set.

IDADLS_MEM_NULL The ida_mem pointer is NULL.

IDADLS_LMEM_NULL The IDADENSE linear solver has not been initialized.

Notes If the IDADENSE setup function failed (i.e., IDASolve returned IDA_LSETUP_FAIL), the value lsflag is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or band) Jacobian matrix. For all other failures, the value of lsflag is negative.

`IDADlsGetReturnFlagName`

Call `name = IDADlsGetReturnFlagName(lsflag);`

Description The function `IDADlsGetReturnFlagName` returns the name of the IDADLS constant corresponding to `lsflag`.

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

Return value The return value is a string containing the name of the corresponding constant. If $1 \leq$ `lsflag` $\leq N$ (LU factorization failed), this function returns "NONE".

4.5.9.5 Sparse direct linear solvers optional output functions

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

`IDASlsGetNumJacEvals`

Call `flag = IDASlsGetNumJacEvals(ida_mem, &njevals);`

Description The function `IDASlsGetNumJacEvals` returns the cumulative number of calls to the IDASLS sparse Jacobian approximation function.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`njevals` (`long int`) the cumulative number of calls to the Jacobian function (total so far).

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

`IDASLS_SUCCESS`	The optional output value has been successfully set.
`IDASLS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASLS_LMEM_NULL`	The IDASLS linear solver has not been initialized.

`IDASlsGetLastFlag`

Call `flag = IDASlsGetLastFlag(ida_mem, &lsflag);`

Description The function `IDASlsGetLastFlag` returns the last return value from an IDASLS routine.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`lsflag` (`long int`) the value of the last return flag from an IDASLS function.

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

`IDASLS_SUCCESS`	The optional output value has been successfully set.
`IDASLS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASLS_LMEM_NULL`	The IDASLS linear solver has not been initialized.

Notes

`IDASlsGetReturnFlagName`

Call `name = IDASlsGetReturnFlagName(lsflag);`

Description The function `IDASlsGetReturnFlagName` returns the name of the IDASLS constant corresponding to `lsflag`.

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

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

4.5.9.6 Iterative linear solvers optional output functions

The following optional outputs are available from the IDASPILS modules: workspace requirements, number of linear iterations, number of linear convergence failures, number of calls to the preconditioner setup and solve routines, number of calls to the Jacobian-vector product routine, number of calls to the residual routine for finite-difference Jacobian-vector product approximation, and last return value from a linear solver function. Note that, where the name of an output would otherwise conflict with the name of an optional output from the main solver, a suffix `LS` (for Linear Solver) has been added here (e.g. `lenrwLS`).

`IDASpilsGetWorkSpace`

Call `flag = IDASpilsGetWorkSpace(ida_mem, &lenrwLS, &leniwLS);`

Description The function `IDASpilsGetWorkSpace` returns the global sizes of the IDASPILS real and integer workspaces.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`lenrwLS` (`long int`) global number of real values in the IDASPILS workspace.
`leniwLS` (`long int`) global number of integer values in the IDASPILS workspace.

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

`IDASPILS_SUCCESS`	The optional output value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.

Notes In terms of the problem size N and maximum subspace size `maxl`, the actual size of the real workspace is roughly:
$N *$ (`maxl` $+5)+$ `maxl` $*($ `maxl` $+4) + 1$ `realtype` words for IDASPGMR,
$10 * N$ `realtype` words for IDASPBCG,
and $13 * N$ `realtype` words for IDASPTFQMR.

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

`IDASpilsGetNumLinIters`

Call `flag = IDASpilsGetNumLinIters(ida_mem, &nliters);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`nliters` (`long int`) the current number of linear iterations.

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

`IDASPILS_SUCCESS`	The optional output value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.

`IDASpilsGetNumConvFails`

Call `flag = IDASpilsGetNumConvFails(ida_mem, &nlcfails);`

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

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

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

`IDASPILS_SUCCESS`	The optional output value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.

`IDASpilsGetNumPrecEvals`

Call `flag = IDASpilsGetNumPrecEvals(ida_mem, &npevals);`

Description The function `IDASpilsGetNumPrecEvals` returns the cumulative number of preconditioner evaluations, i.e., the number of calls made to `psetup`.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`npevals` (`long int`) the cumulative number of calls to `psetup`.

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

`IDASPILS_SUCCESS`	The optional output value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.

`IDASpilsGetNumPrecSolves`

Call `flag = IDASpilsGetNumPrecSolves(ida_mem, &npsolves);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`npsolves` (`long int`) the cumulative number of calls to `psolve`.

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

`IDASPILS_SUCCESS`	The optional output value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.

`IDASpilsGetNumJtimesEvals`

Call `flag = IDASpilsGetNumJtimesEvals(ida_mem, &njvevals);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`njvevals` (`long int`) the cumulative number of calls to `jtimes`.

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

`IDASPILS_SUCCESS`	The optional output value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.

`IDASpilsGetNumResEvals`

Call `flag = IDASpilsGetNumResEvals(ida_mem, &nrevalsLS);`

Description The function `IDASpilsGetNumResEvals` returns the cumulative number of calls to the user residual function for finite difference Jacobian-vector product approximation.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`nrevalsLS` (`long int`) the cumulative number of calls to the user residual function.

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

IDASPILS_SUCCESS	The optional output value has been successfully set.
IDASPILS_MEM_NULL	The ida_mem pointer is NULL.
IDASPILS_LMEM_NULL	The IDASPILS linear solver has not been initialized.

Notes The value nrevalsLS is incremented only if the default IDASpilsDQJtimes difference quotient function is used.

IDASpilsGetLastFlag

Call `flag = IDASpilsGetLastFlag(ida_mem, &lsflag);`

Description The function IDASpilsGetLastFlag returns the last return value from an IDASPILS routine.

Arguments ida_mem (void *) pointer to the IDA memory block.

lsflag (long int) the value of the last return flag from an IDASPILS function.

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

IDASPILS_SUCCESS	The optional output value has been successfully set.
IDASPILS_MEM_NULL	The ida_mem pointer is NULL.
IDASPILS_LMEM_NULL	The IDASPILS linear solver has not been initialized.

Notes If the IDASPILS setup function failed (IDASolve returned IDA_LSETUP_FAIL), lsflag will be SPGMR_PSET_FAIL_UNREC, SPBCG_PSET_FAIL_UNREC, or SPTFQMR_PSET_FAIL_UNREC.

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

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

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

IDASpilsGetReturnFlagName

Call `name = IDASpilsGetReturnFlagName(lsflag);`

Description The function IDASpilsGetReturnFlagName returns the name of the IDASPILS constant corresponding to lsflag.

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

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

4.5.10 IDA reinitialization function

The function IDAReInit reinitializes the main IDA solver for the solution of a problem, where a prior call to IDAInit has been made. The new problem must have the same size as the previous

one. IDAReInit performs the same input checking and initializations that IDAInit does, but does no memory allocation, assuming that the existing internal memory is sufficient for the new problem. A call to IDAReInit deletes the solution history that was stored internally during the previous integration.

The use of IDAReInit requires that the maximum method order, maxord, is no larger for the new problem than for the problem specified in the last call to IDAInit. In addition, the same NVECTOR module set for the previous problem will be reused for the new problem.

If there are changes to the linear solver specifications, make the appropriate IDA*** calls, as described in §4.5.3. If there are changes to any optional inputs, make the appropriate IDASet*** calls, as described in §4.5.7.

IDAReInit

Call `flag = IDAReInit(ida_mem, t0, y0, yp0);`

Description The function IDAReInit provides required problem specifications and reinitializes IDA.

Arguments
- `ida_mem` (`void *`) pointer to the IDA memory block.
- `t0` (`realtype`) is the initial value of t.
- `y0` (`N_Vector`) is the initial value of y.
- `yp0` (`N_Vector`) is the initial value of $\dot{y}$.

Return value The return value `flag` (of type `int`) will be one of the following:
- `IDA_SUCCESS` The call to IDAReInit was successful.
- `IDA_MEM_NULL` The IDA memory block was not initialized through a previous call to IDACreate.
- `IDA_NO_MALLOC` Memory space for the IDA memory block was not allocated through a previous call to IDAInit.
- `IDA_ILL_INPUT` An input argument to IDAReInit has an illegal value.

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

4.6 User-supplied functions

The user-supplied functions consist of one function defining the DAE residual, (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 iteration algorithms.

4.6.1 Residual function

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

IDAResFn

Definition
```
typedef int (*IDAResFn)(realtype tt, N_Vector yy, N_Vector yp,
                        N_Vector rr, void *user_data);
```

Purpose This function computes the problem residual for given values of the independent variable t, state vector y, and derivative $\dot{y}$.

Arguments
- `tt` is the current value of the independent variable.
- `yy` is the current value of the dependent variable vector, $y(t)$.
- `yp` is the current value of $\dot{y}(t)$.
- `rr` is the output residual vector $F(t, y, \dot{y})$.

`user_data` is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.

Return value An `IDAResFn` function type should return a value of 0 if successful, a positive value if a recoverable error occurred (e.g. `yy` has an illegal value), or a negative value if a nonrecoverable error occurred. In the last case, the integrator halts. If a recoverable error occurred, the integrator will attempt to correct and retry.

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

For efficiency reasons, the DAE residual function is not evaluated at the converged solution of the nonlinear solver. Therefore, in general, a recoverable error in that converged value cannot be corrected. (It may be detected when the right-hand side function is called the first time during the following integration step, but a successful step cannot be undone.)

Allocation of memory for `yp` is handled within IDA.

4.6.2 Error message handler function

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

`IDAErrHandlerFn`

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

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

Arguments
`error_code` is the error code.
`module` is the name of the IDA 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 `IDASetErrHandlerFn`.

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

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

4.6.3 Error weight function

As an alternative to providing the relative and absolute tolerances, the user may provide a function of type `IDAEwtFn` to compute a vector `ewt` containing the multiplicative weights W_i used in the WRMS norm $\| v\|_{\text{WRMS}} = \sqrt{(1/N)\sum_1^N (W_i \cdot v_i)^2}$. These weights will used in place of those defined by Eq. (2.6). The function type `IDAEwtFn` is defined as follows:

`IDAEwtFn`

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

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

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

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

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

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

4.6.4 Rootfinding function

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

`IDARootFn`

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

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

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

Return value An `IDARootFn` should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and `IDASolve` returns `IDA_RTFUNC_FAIL`).

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

4.6.5 Jacobian information (direct method with dense Jacobian)

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

`IDADlsDenseJacFn`

Definition
```
typedef int (*IDADlsDenseJacFn)(long int Neq, realtype tt, realtype cj,
                                N_Vector yy, N_Vector yp, N_Vector rr,
                                DlsMat Jac, void *user_data,
                                N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
```

Purpose This function computes the dense Jacobian J of the DAE system (or an approximation to it), defined by Eq. (2.5).

Arguments
- `Neq` is the problem size (number of equations).
- `tt` is the current value of the independent variable t.
- `cj` is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.5)).
- `yy` is the current value of the dependent variable vector, $y(t)$.

yp	is the current value of $\dot{y}(t)$.
rr	is the current value of the residual vector $F(t, y, \dot{y})$.
Jac	is the output (approximate) Jacobian matrix, $J = \partial F/\partial y + cj\ \partial F/\partial \dot{y}$.
user_data	is a pointer to user data, the same as the user_data parameter passed to IDASetUserData.
tmp1	
tmp2	
tmp3	are pointers to memory allocated for variables of type N_Vector which can be used by IDADlsDenseJacFn as temporary storage or work space.

Return value An IDADlsDenseJacFn function type should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if a nonrecoverable error occurred.

In the case of a recoverable eror return, the integrator will attempt to recover by reducing the stepsize, and hence changing α in (2.5).

Notes A user-supplied dense Jacobian function must load the Neq × Neq dense matrix Jac with an approximation to the Jacobian matrix $J(t, y, \dot{y})$ at the point (tt, yy, yp). Only nonzero elements need to be loaded into Jac because Jac is set to the zero matrix before the call to the Jacobian function. The type of Jac is DlsMat (described below and in §8.1).

The accessor macros DENSE_ELEM and DENSE_COL allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the DlsMat type. DENSE_ELEM(Jac, i, j) references the (i, j)-th element of the dense matrix Jac (i, j= 0 . . . Neq−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 Neq, the Jacobian element $J_{m,n}$ can be loaded with the statement DENSE_ELEM(Jac, m-1, n-1) = $J_{m,n}$. Alternatively, DENSE_COL(Jac, j) returns a pointer to the storage for the jth column of Jac (j= 0 . . . Neq−1), and the elements of the j-th column are then accessed via ordinary array indexing. Thus $J_{m,n}$ can be loaded with the statements col_n = DENSE_COL(Jac, n-1); col_n[m-1] = $J_{m,n}$. For large problems, it is more efficient to use DENSE_COL than to use DENSE_ELEM. Note that both of these macros number rows and columns starting from 0, not 1.

The DlsMat type and the accessor macros DENSE_ELEM and DENSE_COL are documented in §8.1.

If the user's IDADlsDenseJacFn 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, the user will need to add a pointer to ida_mem to user_data and then use the IDAGet* functions described in §4.5.9.1. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

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

4.6.6 Jacobian information (direct method with banded Jacobian)

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

IDADlsBandJacFn

Definition
```
typedef int (*IDADlsBandJacFn)(long int Neq, long int mupper,
                               long int mlower, realtype tt, realtype cj,
                               N_Vector yy, N_Vector yp, N_Vector rr,
                               DlsMat Jac, void *user_data,
                               N_Vector tmp1, N_Vector tmp2,N_Vector tmp3);
```

Purpose This function computes the banded Jacobian J of the DAE system (or a banded approximation to it), defined by Eq. (2.5).

Arguments

`Neq`	is the problem size.
`mupper`	
`mlower`	are the upper and lower half bandwidth of the Jacobian.
`tt`	is the current value of the independent variable.
`yy`	is the current value of the dependent variable vector, $y(t)$.
`yp`	is the current value of $\dot{y}(t)$.
`rr`	is the current value of the residual vector $F(t, y, \dot{y})$.
`cj`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.5)).
`Jac`	is the output (approximate) Jacobian matrix, $J = \partial F/\partial y + cj\ \partial F/\partial \dot{y}$.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.
`tmp1`	
`tmp2`	
`tmp3`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDADlsBandJacFn` as temporary storage or work space.

Return value A `IDADlsBandJacFn` function type should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if a nonrecoverable error occurred.

In the case of a recoverable eror return, the integrator will attempt to recover by reducing the stepsize, and hence changing α in (2.5).

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

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

The `DlsMat` type and the accessor macros `BAND_ELEM`, `BAND_COL`, and `BAND_COL_ELEM` are documented in §8.1.

If the user's IDADlsBandJacFn 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, the user will need to add a pointer to ida_mem to user_data and then use the IDAGet* functions described in §4.5.9.1. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

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

4.6.7 Jacobian information (direct method with sparse Jacobian)

If the direct linear solver with sparse treatment of the Jacobian is used (i.e. either IDAKLU or IDASuperLUMT is called in Step 8 of §4.4), the user must provide a function of type IDASlsSparseJacFn defined as follows:

IDASlsSparseJacFn

Definition
```
typedef int (*IDASlsSparseJacFn)(realtype t, realtype c_j,
                                 N_Vector y, N_Vector yp, N_Vector r,
                                 SlsMat Jac, void *user_data,
                                 N_Vector tmp1, N_Vector tmp2,N_Vector tmp3);
```

Purpose This function computes the sparse Jacobian J of the DAE system (or an approximation to it), defined by Eq. (2.5).

Arguments

t	is the current value of the independent variable.
y	is the current value of the dependent variable vector, $y(t)$.
yp	is the current value of $\dot{y}(t)$.
r	is the current value of the residual vector $F(t, y, \dot{y})$.
c_j	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.5)).
Jac	is the output (approximate) Jacobian matrix, $J = \partial F/\partial y + cj\ \partial F/\partial \dot{y}$.
user_data	is a pointer to user data, the same as the user_data parameter passed to IDASetUserData.
tmp1	
tmp2	
tmp3	are pointers to memory allocated for variables of type N_Vector which can be used by IDASlsSparseJacFn as temporary storage or work space.

Return value A IDASlsSparseJacFn function type should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if a nonrecoverable error occurred.

In the case of a recoverable error return, the integrator will attempt to recover by reducing the stepsize, and hence changing α in (2.5).

Notes A user-supplied sparse Jacobian function must load the compressed-sparse-column matrix Jac with the elements of the Jacobian $J(t, y, \dot{y})$ at the point (t, y, yp). Storage for Jac already exists on entry to this function, although the user should ensure that sufficient space is allocated in Jac to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and row index arrays as needed. The type of Jac is SlsMat, and the amount of allocated space is available within the SlsMat structure as NNZ. The SlsMat type is further documented in the Section §8.2.

If the user's IDASlsSparseJacFn function uses difference quotient approximations to set the specific nonzero matrix entries, then it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to ida_mem to user_data and then

use the IDAGet* functions described in §4.5.9.1. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

4.6.8 Jacobian information (matrix-vector product)

If one of the Krylov iterative linear solvers SPGMR, SPBCG, or SPTFQMR is selected (IDASp* is called in step 8 of §4.4), the user may provide a function of type IDASpilsJacTimesVecFn, described below, to compute matrix-vector products Jv. If such a function is not supplied, the default is a difference quotient approximation to these products.

IDASpilsJacTimesVecFn

Definition
```
typedef int (*IDASpilsJacTimesVecFn)(realtype tt, N_Vector yy,
                                     N_Vector yp, N_Vector rr,
                                     N_Vector v, N_Vector Jv,
                                     realtype cj, void *user_data,
                                     N_Vector tmp1, N_Vector tmp2);
```

Purpose This function computes the product Jv of the DAE system Jacobian J (or an approximation to it) and a given vector v, where J is defined by Eq. (2.5).

Arguments

tt	is the current value of the independent variable.
yy	is the current value of the dependent variable vector, $y(t)$.
yp	is the current value of $\dot{y}(t)$.
rr	is the current value of the residual vector $F(t, y, \dot{y})$.
v	is the vector by which the Jacobian must be multiplied to the right.
Jv	is the computed output vector.
cj	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.5)).
user_data	is a pointer to user data, the same as the user_data parameter passed to IDASetUserData.
tmp1	
tmp2	are pointers to memory allocated for variables of type N_Vector which can be used by IDASpilsJacTimesVecFn as temporary storage or work space.

Return value The value to be returned by the Jacobian-times-vector function should be 0 if successful. A nonzero value indicates that a nonrecoverable error occurred.

If the user's IDASpilsJacTimesVecFn 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, the user will need to add a pointer to ida_mem to user_data and then use the IDAGet* functions described in §4.5.9.1. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

4.6.9 Preconditioning (linear system solution)

If preconditioning is used, then the user must provide a C function to solve the linear system $Pz = r$ where P is a left preconditioner matrix which approximates (at least crudely) the Jacobian matrix $J = \partial F/\partial y + cj\ \partial F/\partial \dot{y}$. This function must be of type IDASpilsPrecSolveFn, defined as follows:

IDASpilsPrecSolveFn

Definition
```
typedef int (*IDASpilsPrecSolveFn)(realtype tt, N_Vector yy,
                                   N_Vector yp, N_Vector rr,
                                   N_Vector rvec, N_Vector zvec,
                                   realtype cj, realtype delta,
                                   void *user_data, N_Vector tmp);
```

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

Arguments
- `tt` is the current value of the independent variable.
- `yy` is the current value of the dependent variable vector, $y(t)$.
- `yp` is the current value of $\dot{y}(t)$.
- `rr` is the current value of the residual vector $F(t, y, \dot{y})$.
- `rvec` is the right-hand side vector r of the linear system to be solved.
- `zvec` is the computed output vector.
- `cj` is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.5)).
- `delta` is an input tolerance to be used if an iterative method is employed in the solution. In that case, the residual vector $Res = r - Pz$ of the system should be made less than `delta` in weighted l_2 norm, i.e., $\sqrt{\sum_i (Res_i \cdot ewt_i)^2} <$ `delta`. To obtain the `N_Vector ewt`, call `IDAGetErrWeights` (see §4.5.9.1).
- `user_data` is a pointer to user data, the same as the `user_data` parameter passed to the function `IDASetUserData`.
- `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).

4.6.10 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 `IDASpilsPrecSetupFn`, defined as follows:

`IDASpilsPrecSetupFn`

Definition
```
typedef int (*IDASpilsPrecSetupFn)(realtype tt, N_Vector yy,
                                   N_Vector yp, N_Vector rr,
                                   realtype cj, void *user_data,
                                   N_Vector tmp1, N_Vector tmp2,
                                   N_Vector tmp3);
```

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

Arguments The arguments of an `IDASpilsPrecSetupFn` are as follows:
- `tt` is the current value of the independent variable.
- `yy` is the current value of the dependent variable vector, $y(t)$.
- `yp` is the current value of $\dot{y}(t)$.
- `rr` is the current value of the residual vector $F(t, y, \dot{y})$.
- `cj` is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.5)).
- `user_data` is a pointer to user data, the same as the `user_data` parameter passed to the function `IDASetUserData`.
- `tmp1`
- `tmp2`
- `tmp3` are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDASpilsPrecSetupFn` 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.

Each call to the preconditioner setup function is preceded by a call to the `IDAResFn` user function with the same (`tt`, `yy`, `yp`) arguments. Thus the preconditioner setup function can use any auxiliary data that is computed and saved during the evaluation of the DAE residual.

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 `IDASpilsPrecSetupFn` 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, the user will need to add a pointer to `ida_mem` to `user_data` and then use the `IDAGet*` functions described in §4.5.9.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

4.7 A parallel band-block-diagonal preconditioner module

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

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used for several realistic, large-scale problems [19] and is included in a software module within the IDA package. This module works with the parallel vector module NVECTOR_PARALLEL and generates a preconditioner that is a block-diagonal matrix with each block being a band matrix. The blocks need not have the same number of super- and sub-diagonals and these numbers may vary from block to block. This Band-Block-Diagonal Preconditioner module is called IDABBDPRE.

One way to envision these preconditioners is to think of the domain of the computational PDE problem as being subdivided into M non-overlapping sub-domains. Each of these sub-domains is then assigned to one of the M processors to be used to solve the DAE 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 residual function. This requires the definition of a new function $G(t, y, \dot{y})$ which approximates the function $F(t, y, \dot{y})$ in the definition of the DAE system (2.1). However, the user may set $G = F$. Corresponding to the domain decomposition, there is a decomposition of the solution vectors y and $\dot{y}$ into M disjoint blocks y_m and $\dot{y}_m$, and a decomposition of G into blocks G_m. The block G_m depends on y_m and $\dot{y}_m$, and also on components of $y_{m'}$ and $\dot{y}_{m'}$ associated with neighboring sub-domains (so-called ghost-cell data). Let $\bar{y}_m$ and $\bar{\dot{y}}_m$ denote y_m and $\dot{y}_m$ (respectively) augmented with those other components on which G_m depends. Then we have

$$G(t, y, \dot{y}) = [G_1(t, \bar{y}_1, \bar{\dot{y}}_1), G_2(t, \bar{y}_2, \bar{\dot{y}}_2), \ldots, G_M(t, \bar{y}_M, \bar{\dot{y}}_M)]^T \ , \tag{4.1}$$

and each of the blocks $G_m(t, \bar{y}_m, \bar{\dot{y}}_m)$ is uncoupled from the others.

The preconditioner associated with this decomposition has the form

$$P = diag[P_1, P_2, \ldots, P_M] \tag{4.2}$$

where

$$P_m \approx \partial G_m/\partial y_m + \alpha \partial G_m/\partial \dot{y}_m \tag{4.3}$$

This matrix is taken to be banded, with upper and lower half-bandwidths `mudq` and `mldq` defined as the number of non-zero diagonals above and below the main diagonal, respectively. The difference quotient approximation is computed using `mudq` + `mldq` +2 evaluations of G_m, but only a matrix of bandwidth `mukeep` + `mlkeep` +1 is retained.

Neither pair of parameters need be the true half-bandwidths of the Jacobians of the local block of G, if smaller values provide a more efficient preconditioner. Such an efficiency gain may occur if the couplings in the DAE system outside a certain bandwidth are considerably weaker than those within the band. Reducing `mukeep` and `mlkeep` while keeping `mudq` and `mldq` at their true values, discards the elements outside the narrower band. Reducing both pairs has the additional effect of lumping the outer Jacobian elements into the computed elements within the band, and requires more caution and experimentation.

The solution of the complete linear system

$$Px = b \tag{4.4}$$

reduces to solving each of the equations

$$P_m x_m = b_m \tag{4.5}$$

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

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

The IDABBDPRE module calls two user-provided functions to construct P: a required function `Gres` (of type `IDABBDLocalFn`) which approximates the residual function $G(t, y, \dot{y}) \approx F(t, y, \dot{y})$ and which is computed locally, and an optional function `Gcomm` (of type `IDABBDCommFn`) which performs all inter-process communication necessary to evaluate the approximate residual G. These are in addition to the user-supplied residual function `res`. Both functions take as input the same pointer `user_data` as passed by the user to `IDASetUserData` and passed to the user's function `res`. The user is responsible for providing space (presumably within `user_data`) for components of `yy` and `yp` that are communicated by `Gcomm` from the other processors, and that are then used by `Gres`, which should not do any communication.

`IDABBDLocalFn`

Definition
```
typedef int (*IDABBDLocalFn)(long int Nlocal, realtype tt,
                             N_Vector yy, N_Vector yp, N_Vector gval,
                             void *user_data);
```

Purpose This `Gres` function computes $G(t, y, \dot{y})$. It loads the vector `gval` as a function of `tt`, `yy`, and `yp`.

Arguments

`Nlocal`	is the local vector length.
`tt`	is the value of the independent variable.
`yy`	is the dependent variable.
`yp`	is the derivative of the dependent variable.
`gval`	is the output vector.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.

Return value An `IDABBDLocalFn` function type should return 0 to indicate success, 1 for a recoverable error, or -1 for a non-recoverable error.

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

The case where G is mathematically identical to F is allowed.

IDABBDCommFn

Definition
```
typedef int (*IDABBDCommFn)(long int Nlocal, realtype tt,
                            N_Vector yy, N_Vector yp, void *user_data);
```

Purpose This Gcomm function performs all inter-processor communications necessary for the execution of the Gres function above, using the input vectors yy and yp.

Arguments
- Nlocal is the local vector length.
- tt is the value of the independent variable.
- yy is the dependent variable.
- yp is the derivative of the dependent variable.
- user_data is a pointer to user data, the same as the user_data parameter passed to IDASetUserData.

Return value An IDABBDCommFn function type should return 0 to indicate success, 1 for a recoverable error, or -1 for a non-recoverable error.

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

Each call to the Gcomm function is preceded by a call to the residual function res with the same (tt, yy, yp) arguments. Thus Gcomm can omit any communications done by res if relevant to the evaluation of Gres. If all necessary communication was done in res, then Gcomm = NULL can be passed in the call to IDABBDPrecInit (see below).

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

The following is a summary of the usage of this module and describes the sequence of calls in the user main program. Steps that are unchanged from the user main program presented in §4.4 are grayed-out.

1. **Initialize MPI**
2. **Set problem dimensions**
3. **Set vector of initial values**
4. **Create IDA object**
5. **Allocate internal memory**
6. **Set optional inputs**
7. **Attach iterative linear solver, one of:**
   (a) `flag = IDASpgmr(ida_mem, maxl);`
   (b) `flag = IDASpbcg(ida_mem, maxl);`
   (c) `flag = IDASptfqmr(ida_mem, maxl);`
8. **Initialize the IDABBDPRE preconditioner module**

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

   ```
   flag = IDABBDPrecInit(ida_mem, Nlocal, mudq, mldq,
                         mukeep, mlkeep, dq_rel_yy, Gres, Gcomm);
   ```

   to allocate memory and initialize the internal preconditioner data. The last two arguments of IDABBDPrecInit are the two user-supplied functions described above.

9. **Set linear solver optional inputs**

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

10. **Correct initial values**

11. **Specify rootfinding problem**

12. **Advance solution in time**

13. **Get optional outputs**

    Additional optional outputs associated with IDABBDPRE are available by way of two routines described below, `IDABBDPrecGetWorkSpace` and `IDABBDPrecGetNumGfnEvals`.

14. **Deallocate memory for solution vector**

15. **Free solver memory**

16. **Finalize MPI**

The user-callable functions that initialize (step 8 above) or re-initialize the IDABBDPRE preconditioner module are described next.

`IDABBDPrecInit`

Call
```
flag = IDABBDPrecInit(ida_mem, Nlocal, mudq, mldq,
                      mukeep, mlkeep, dq_rel_yy, Gres, Gcomm);
```

Description The function `IDABBDPrecInit` initializes and allocates (internal) memory for the IDABBDPRE preconditioner.

Arguments

`ida_mem`	(`void *`) pointer to the IDA memory block.
`Nlocal`	(`long int`) local vector dimension.
`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.
`dq_rel_yy`	(`realtype`) the relative increment in components of `y` used in the difference quotient approximations. The default is `dq_rel_yy`$= \sqrt{\text{unit roundoff}}$, which can be specified by passing `dq_rel_yy`$= 0.0$.
`Gres`	(`IDABBDLocalFn`) the C function which computes the local residual approximation $G(t, y, \dot{y})$.
`Gcomm`	(`IDABBDCommFn`) the optional C function which performs all inter-process communication required for the computation of $G(t, y, \dot{y})$.

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

`IDASPILS_SUCCESS`	The call to `IDABBDPrecInit` was successful.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer was `NULL`.
`IDASPILS_MEM_FAIL`	A memory allocation request has failed.
`IDASPILS_LMEM_NULL`	An IDASPILS linear solver memory was not attached.
`IDASPILS_ILL_INPUT`	The supplied vector implementation was not compatible with block band preconditioner.

Notes If one of the half-bandwidths `mudq` or `mldq` to be used in the difference-quotient calculation of the approximate Jacobian is negative or exceeds the value `Nlocal`−1, it is replaced by 0 or `Nlocal`−1 accordingly.

The half-bandwidths `mudq` and `mldq` need not be the true half-bandwidths of the Jacobian of the local block of G, when smaller values may provide a greater efficiency.

Also, the half-bandwidths `mukeep` and `mlkeep` of the retained banded approximate Jacobian block may be even smaller, to reduce storage and computation costs further.

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

The IDABBDPRE module also provides a reinitialization function to allow for a sequence of problems of the same size with IDASPGMR/IDABBDPRE, IDASPBCG/IDABBDPRE, or IDASPTFQMR/IDABBDPRE, provided there is no change in `local_N`, `mukeep`, or `mlkeep`. After solving one problem, and after calling `IDAReInit` to re-initialize IDA for a subsequent problem, a call to `IDABBDPrecReInit` 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 `dq_rel_yy`, or one of the user-supplied functions `Gres` and `Gcomm`.

`IDABBDPrecReInit`

Call `flag = IDABBDPrecReInit(ida_mem, mudq, mldq, dq_rel_yy);`

Description The function `IDABBDPrecReInit` reinitializes the IDABBDPRE preconditioner.

Arguments
- `ida_mem` (`void *`) pointer to the IDA memory block.
- `mudq` (`long int`) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mldq` (`long int`) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `dq_rel_yy` (`realtype`) the relative increment in components of `y` used in the difference quotient approximations. The default is `dq_rel_yy` $= \sqrt{\text{unit roundoff}}$, which can be specified by passing `dq_rel_yy` $= 0.0$.

Return value The return value `flag` (of type `int`) is one of
- `IDASPILS_SUCCESS` The call to `IDABBDPrecReInit` was successful.
- `IDASPILS_MEM_NULL` The `ida_mem` pointer was `NULL`.
- `IDASPILS_LMEM_NULL` An IDASPILS linear solver memory was not attached.
- `IDASPILS_PMEM_NULL` The function `IDABBDPrecInit` was not previously called.

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

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

`IDABBDPrecGetWorkSpace`

Call `flag = IDABBDPrecGetWorkSpace(ida_mem, &lenrwBBDP, &leniwBBDP);`

Description The function `IDABBDPrecGetWorkSpace` returns the local sizes of the IDABBDPRE real and integer workspaces.

Arguments
- `ida_mem` (`void *`) pointer to the IDA memory block.
- `lenrwBBDP` (`long int`) local number of real values in the IDABBDPRE workspace.
- `leniwBBDP` (`long int`) local number of integer values in the IDABBDPRE workspace.

Return value The return value `flag` (of type `int`) is one of
- `IDASPILS_SUCCESS` The optional output value has been successfully set.
- `IDASPILS_MEM_NULL` The `ida_mem` pointer was `NULL`.
- `IDASPILS_PMEM_NULL` The IDABBDPRE preconditioner has not been initialized.

Notes In terms of the local vector dimension N_l, and `smu` $= \min(N_l - 1,$ `mukeep` $+$ `mlkeep`$)$, the actual size of the real workspace is $N_l\,(2$ `mlkeep` $+$ `mukeep` $+$ `smu` $+2)$ `realtype` words. The actual size of the integer workspace is N_l integer words.

`IDABBDPrecGetNumGfnEvals`

Call `flag = IDABBDPrecGetNumGfnEvals(ida_mem, &ngevalsBBDP);`

Description The function `IDABBDPrecGetNumGfnEvals` returns the cumulative number of calls to the user `Gres` function due to the finite difference approximation of the Jacobian blocks used within IDABBDPRE's preconditioner setup function.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

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

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

`IDASPILS_SUCCESS`	The optional output value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer was `NULL`.
`IDASPILS_PMEM_NULL`	The IDABBDPRE preconditioner has not been initialized.

In addition to the `ngevalsBBDP` `Gres` evaluations, the costs associated with IDABBDPRE also include `nlinsetups` LU factorizations, `nlinsetups` calls to `Gcomm`, `npsolves` banded backsolve calls, and `nrevalsLS` residual function evaluations, where `nlinsetups` is an optional IDA output (see §4.5.9.1), and `npsolves` and `nrevalsLS` are linear solver optional outputs (see §4.5.9.6).

Chapter 5

FIDA, an Interface Module for FORTRAN Applications

The FIDA interface module is a package of C functions which support the use of the IDA solver, for the solution of DAE systems, in a mixed FORTRAN/C setting. While IDA 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 IDA for all supplied serial and parallel NVECTOR implementations.

5.1 Important note on portability

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

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

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

5.2 Fortran Data Types

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

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

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

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

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

5.3 FIDA routines

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

- Interface to the NVECTOR modules
  - `FNVINITS` (defined by NVECTOR_SERIAL) interfaces to `N_VNewEmpty_Serial`.
  - `FNVINITOMP` (defined by NVECTOR_OPENMP) interfaces to `N_VNewEmpty_OpenMP`.
  - `FNVINITPTS` (defined by NVECTOR_PTHREADS) interfaces to `N_VNewEmpty_Pthreads`.
  - `FNVINITP` (defined by NVECTOR_PARALLEL) interfaces to `N_VNewEmpty_Parallel`.
- Interface to the main IDA module
  - `FIDAMALLOC` interfaces to `IDACreate`, `IDASetUserData`, `IDAInit`, `IDASStolerances`, and `IDASVtolerances`.
  - `FIDAREINIT` interfaces to `IDAReInit` and `IDASStolerances`/`IDASVtolerances`.
  - `FIDASETIIN`, `FIDASETVIN`, and `FIDASETRIN` interface to `IDASet*` functions.
  - `FIDATOLREINIT` interfaces to `IDASStolerances`/`IDASVtolerances`.
  - `FIDACALCIC` interfaces to `IDACalcIC`.
  - `FIDAEWTSET` interfaces to `IDAWFtolerances`.
  - `FIDASOLVE` interfaces to `IDASolve`, `IDAGet*` functions, and to the optional output functions for the selected linear solver module.
  - `FIDAGETDKY` interfaces to `IDAGetDky`.
  - `FIDAGETERRWEIGHTS` interfaces to `IDAGetErrWeights`.
  - `FIDAGETESTLOCALERR` interfaces to `IDAGetEstLocalErrors`.
  - `FIDAFREE` interfaces to `IDAFree`.
- Interface to the linear solver modules
  - `FIDADENSE` interfaces to `IDADense`.
  - `FIDADENSESETJAC` interfaces to `IDADlsSetDenseJacFn`.
  - `FIDALAPACKDENSE` interfaces to `IDALapackDense`.
  - `FIDALAPACKDENSESETJAC` interfaces to `IDADlsSetDenseJacFn`.
  - `FIDABAND` interfaces to `IDABand`.
  - `FIDABANDSETJAC` interfaces to `IDADlsSetBandJacFn`.
  - `FIDALAPACKBAND` interfaces to `IDALapackBand`.
  - `FIDALAPACKBANDSETJAC` interfaces to `IDADlsSetBandJacFn`.
  - `FIDAKLU` interfaces to `IDAKLU`.
  - `FIDAKLUREINIT` interfaces to `IDAKLUReInit`.

– FIDASUPERLUMT interfaces to IDASuperLUMT.
– FIDASPGMR interfaces to IDASpgmr and SPGMR optional input functions.
– FIDASPGMRREINIT interfaces to SPGMR optional input functions.
– FIDASPBCG interfaces to IDASpbcg and SPBCG optional input functions.
– FIDASPBCGREINIT interfaces to SPBCG optional input functions.
– FIDASPTFQMR interfaces to IDASptfqmr and SPTFQMR optional input functions.
– FIDASPTFQMRREINIT interfaces to SPTFQMR optional input functions.
– FIDASPILSSETJAC interfaces to IDASpilsSetJacTimesVecFn.
– FIDASPILSSETPREC interfaces to IDASpilsSetPreconditioner.

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

FIDA routine (FORTRAN, user-supplied)	IDA function (C, interface)	IDA type of interface function
FIDARESFUN	FIDAresfn	IDAResFn
FIDAEWT	FIDAEwtSet	IDAEwtFn
FIDADJAC	FIDADenseJac	IDADlsDenseJacFn
	FIDALapackDenseJac	IDADlsDenseJacFn
FIDABJAC	FIDABandJac	IDADlsBandJacFn
	FIDALapackBandJac	IDADlsBandJacFn
FIDASPJAC	FIDASparseJac	IDASlsSparseJacFn
FIDAPSOL	FIDAPSol	IDASpilsPrecSolveFn
FIDAPSET	FIDAPSet	IDASpilsPrecSetupFn
FIDAJTIMES	FIDAJtimes	IDASpilsJacTimesVecFn

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

5.4 Usage of the FIDA interface module

The usage of FIDA requires calls to five or more 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 IDA 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 FIDA for rootfinding, and usage of FIDA with preconditioner modules, are each described in later sections.

In the instructions below, steps marked [**S**] apply to the NVECTOR module NVECTOR_SERIAL, steps marked [**O**] apply to NVECTOR_OPENMP, steps marked [**T**] apply to NVECTOR_PTHREADS, while steps marked [**P**] apply to NVECTOR_PARALLEL,

1. **Residual function specification**

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

   ```
   SUBROUTINE FIDARESFUN (T, Y, YP, R, IPAR, RPAR, IER)
   DIMENSION Y(*), YP(*), R(*), IPAR(*), RPAR(*)
   ```

   It must set the R array to $F(t, y, \dot{y})$, the residual function of the DAE system, as a function of T $= t$ and the arrays Y $= y$ and YP $= \dot{y}$. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. It should return IER $= 0$ if it was successful, IER $= 1$ if it had a recoverable failure, or IER $=$ -1 if it had a non-recoverable failure.

2. NVECTOR **module initialization**

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

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

where KEY is the solver id (KEY = 2 for IDA), NEQ is the size of vectors, and IER is a return flag, which is set to 0 on success and −1 if a failure occurred.

[**O**] To initialize the NVECTOR_OPENMP NVECTOR module, the user must make the following call:

```
CALL FNVINITOMP(KEY, NEQ, NUMTHREADS, IER)
```

where KEY is the solver id (KEY = 2 for IDA), NEQ is the size of vectors, NUMTHREADS is the number of threads, and IER is a return completion flag which is 0 on success and −1 if a failure occurred.

[**T**] To initialize the NVECTOR_PTHREADS NVECTOR module, the user must make the following call:

```
CALL FNVINITPTS(KEY, NEQ, NUMTHREADS, IER)
```

where KEY is the solver id (KEY = 2 for IDA), NEQ is the size of vectors, NUMTHREADS is the number of threads, and IER is a return completion flag which is 0 on success and −1 if a failure occurred.

[**P**] To initialize the MPI-based distributed memory 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 = 2 for IDA, NLOCAL = the local size of all vectors on this processor, and NGLOBAL = the system size (and the global size of 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.

NOTE: The integers NEQ, NLOCAL, and NGLOBAL should be declared so as to match C type `long int`.

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

3. **Problem specification**

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

FIDAMALLOC

Call
```
CALL FIDAMALLOC(T0, Y0, YP0, IATOL, RTOL, ATOL,
&                IOUT, ROUT, IPAR, RPAR, IER)
```

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

Arguments
- T0 is the initial value of t.
- Y0 is an array of initial conditions for y.
- YP0 is an array of initial conditions for $\dot{y}$.

IATOL specifies the type for absolute tolerance ATOL: 1 for scalar or 2 for array. If IATOL= 3, the arguments RTOL and ATOL are ignored and the user is expected to subsequently call FIDAEWTSET and provide the function FIDAEWT.
RTOL is the relative tolerance (scalar).
ATOL is the absolute tolerance (scalar or array).
IOUT is an integer array of length at least 21 for integer optional outputs.
ROUT is a real array of length at least 6 for real optional outputs.
IPAR is an integer array of user data which will be passed unmodified to all user-provided routines.
RPAR is a real array of user data which will be passed unmodified to all user-provided routines.

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

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

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

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

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

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

It must set the positive components of the error weight vector EWT for the calculation of the WRMS norm of Y. On return, set IER $= 0$ if FIDAEWT 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 FIDAMALLOC.

If the FIDAEWT routine is provided, then, following the call to FIDAMALLOC, the user must make the call:

```
CALL FIDAEWTSET (FLAG, IER)
```

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

4. **Set optional inputs**

Call FIDASETIIN, FIDASETRIN, and/or FIDASETVIN to set desired optional inputs, if any. See §5.5 for details.

5. **Linear solver specification**

The variable-order, variable-coefficient BDF method used by IDA involves the solution of linear systems related to the system Jacobian $J = \partial F/\partial y + \alpha \partial F/\partial \dot{y}$. See Eq. (2.4). The user of FIDA must call a routine with a specific name to make the desired choice of linear solver.

[**S**] **Dense treatment of the linear system**

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

```
CALL FIDADENSE (NEQ, IER)
```

or

```
CALL FIDALAPACKDENSE (NEQ, IER)
```

where NEQ is the size of the DAE system, depending on whether the internal or a Lapack dense linear solver is to be used. 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. In the case of FIDALAPACKDENSE, NEQ must be declared so as to match C type int.

As an option when using the DENSE linear solver, the user may supply a routine that computes a dense approximation of the system Jacobian. If supplied, it must have the following form:

```
 SUBROUTINE FIDADJAC (NEQ, T, Y, YP, R, DJAC, CJ, EWT, H,
&                     IPAR, RPAR, WK1, WK2, WK3, IER)
 DIMENSION Y(*), YP(*), R(*), EWT(*), DJAC(NEQ,*),
&          IPAR(*), RPAR(*), WK1(*), WK2(*), WK3(*)
```

This routine must compute the Jacobian and store it columnwise in DJAC. The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FIDADJAC. The input arguments T, Y, YP, R, and CJ are the current values of t, y, $\dot{y}$, $F(t, y, \dot{y})$, and α, respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. NOTE: The argument NEQ has a type consistent with C type long int even in the case when the Lapack dense solver is to be used.

If the user's FIDADJAC uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. It may also need the unit roundoff, which can be obtained as the optional output ROUT(6), passed from the calling program to this routine using COMMON.

If the FIDADJAC routine is provided, then, following the call to FIDADENSE (or FIDALAPACKDENSE), the user must make the call:

```
CALL FIDADENSESETJAC (FLAG, IER)
```

or

```
CALL FIDALAPACKDENSESETJAC (FLAG, IER)
```

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

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

[S] Band treatment of the linear system

The user must make the call:

```
CALL FIDABAND (NEQ, MU, ML, IER)
```

or

```
CALL FIDALAPACKBAND (NEQ, MU, ML, IER)
```

depending on whether the internal or a Lapack band linear solver is to be used. 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. In the case of FIDALAPACKBAND, the arguments NEQ, MU, and ML must be declared so as to match C type int.

As an option when using the BAND linear solver, the user may supply a routine that computes a band approximation of the system Jacobian. If supplied, it must have the following form:

```
 SUBROUTINE FIDABJAC(NEQ, MU, ML, MDIM, T, Y, YP, R, CJ, BJAC,
&                    EWT, H, IPAR, RPAR, WK1, WK2, WK3, IER)
 DIMENSION Y(*), YP(*), R(*), EWT(*), BJAC(MDIM,*),
&          IPAR(*), RPAR(*), WK1(*), WK2(*), WK3(*)
```

This routine must load the MDIM by NEQ array BJAC with the Jacobian matrix at the current $(t, y, \dot{y})$ in band form. Store in BJAC(k, j) the Jacobian element $J_{i,j}$ with $k = i - j$+MU+1 ($k = 1 \cdots$ ML + MU + 1) and $j = 1 \cdots N$. The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FIDABJAC. The input arguments T, Y, YP, R, and CJ are the current values of t, y, $\dot{y}$, $F(t, y, \dot{y})$, and α, respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. NOTE: The arguments NEQ, MU, ML, and MDIM have a type consistent with C type `long int` even in the case when the Lapack band solver is to be used.

If the user's FIDABJAC uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. It may also need the unit roundoff, which can be obtained as the optional output ROUT(6), passed from the calling program to this routine using COMMON.

If the FIDABJAC routine is provided, then, following the call to FIDABAND (or FIDALAPACKBAND), the user must make the call:

```
CALL FIDABANDSETJAC (FLAG, IER)
```

or

```
CALL FIDALAPACKBANDSETJAC (FLAG, IER)
```

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

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

[S] Sparse direct treatment of the linear system

To use the KLU sparse direct linear solver, the user must make the call:

```
CALL FIDAKLU (NEQ, NNZ, ORDERING, IER)
```

where NEQ is the size of the DAE system, NNZ is the maximum number of nonzeros in the Jacobian matrix, and ORDERING is the matrix ordering desired with possible values from the KLU package (0 = AMD, 1 = COLAMD). The argument IER is an error return flag which is 0 for success or negative for an error.

The IDA KLU solver will reuse much of the factorization information from one nonlinear iteration and time step to the next. If at any time the user wants to force a full refactorization, or if the number of nonzeros in the Jacobian matrix changes, the user should make the call

```
 CALL FIDAKLUREINIT(NEQ, NNZ, REINIT_TYPE)
```

where NEQ is the size of the DAE system, NNZ is the maximum number of nonzeros in the Jacobian matrix, and REINIT_TYPE is 1 or 2. For a value of 1, the matrix will be destroyed and a new one will be allocated with NNZ nonzeros. For a value of 2, only symbolic and numeric factorizations will be completed.

Alternatively, to use the SuperLUMT linear solver, the user must make the call:

```
CALL FIDASUPERLUMT (NEQ, NNZ, ORDERING, IER)
```

where the arguments have the same meanings as for FIDAKLU, except that here possible values for ORDERING derive from the SUPERLUMT package and include: 0 for Natural ordering, 1 for Minimum degree on A^TA, 2 for Minimum degree on $A^T + A$, and 3 for COLAMD.

If the either of the sparse direct interface packages are used, then the user must supply the FIDASPJAC routine that computes a compressed-sparse-column approximation of the system Jacobian $J = \partial F/\partial y + c_j \partial F/\partial \dot{y}$. If supplied, it must have the following form:

```
SUBROUTINE FIDASPJAC(T, CJ, Y, YP, R, N, NNZ, JDATA, JRVALS,
&                    JCPTRS, H, IPAR, RPAR, WK1, WK2, WK3, IER)
```

It must load the N by N compressed sparse column matrix with storage for NNZ nonzeros, stored in the arrays JDATA (nonzero values), JRVALS (row indices for each nonzero), JCOLPTRS (indices for start of each column), with the Jacobian matrix at the current (t, y) in CSC form (see sundials_sparse.h for more information). The arguments are T, the current time; CJ, scalar in the system proportional to the inverse step size; Y, an array containing state variables; YP, an array containing state derivatives; R, an array containing the system nonlinear residual; N, the number of matrix rows/columns in the Jacobian; NNZ, allocated length of nonzero storage; JDATA, nonzero values in the Jacobian (of length NNZ); JRVALS, row indices for each nonzero in Jacobian (of length NNZ); JCPTRS, pointers to each Jacobian column in the two preceding arrays (of length N+1); H, the current step size; IPAR, an array containing integer user data that was passed to FIDAMALLOC; RPAR, an array containing real user data that was passed to FIDAMALLOC; WK*, work arrays containing temporary workspace of same size as Y; and IER, error return code (0 if successful, > 0 if a recoverable error occurred, or < 0 if an unrecoverable error occurred.)

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

[S][P] SPGMR treatment of the linear systems

For the Scaled Preconditioned GMRES solution of the linear systems, the user must make the call

```
CALL FIDASPGMR (MAXL, IGSTYPE, MAXRS, EPLIFAC, DQINCFAC, IER)
```

The arguments are as follows. MAXL is the maximum Krylov subspace dimension. IGSTYPE indicates the Gram-Schmidt process type: 1 for modified, or 2 for classical. MAXRS maximum number of restarts. EPLIFAC is the linear convergence tolerance factor. DQINCFAC is the optional increment factor used in the matrix-vector product Jv. 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 5.2.

For descriptions of the relevant optional user-supplied routines, see **User-supplied routines for SPGMR/SPBCG/SPTFQMR** below.

[S][P] SPBCG treatment of the linear systems

For the Scaled Preconditioned Bi-CGStab solution of the linear systems, the user must make the call

```
CALL FIDASPBCG (MAXL, EPLIFAC, DQINCFAC, IER)
```

The arguments are as follows. MAXL is the maximum Krylov subspace dimension. EPLIFAC is the linear convergence tolerance factor. DQINCFAC is the optional increment factor used in the matrix-vector product Jv. 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 SPBCG case are listed in Table 5.2.

For descriptions of the relevant optional user-supplied routines, see **User-supplied routines for SPGMR/SPBCG/SPTFQMR** below.

[S][P] SPTFQMR treatment of the linear systems

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

```
CALL FIDASPTFQMR (MAXL, EPLIFAC, DQINCFAC, IER)
```

The arguments are as follows. `MAXL` is the maximum Krylov subspace dimension. `EPLIFAC` is the linear convergence tolerance factor. `DQINCFAC` is the optional increment factor used in the matrix-vector product Jv. 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 SPTFQMR case are listed in Table 5.2.

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

[S][P] Functions used by SPGMR/SPBCG/SPTFQMR

An optional user-supplied routine, `FIDAJTIMES`, can be provided for Jacobian-vector products. If it is, then, following the call to `FIDASPGMR`, `FIDASPBCG`, or `FIDASPTFQMR`, the user must make the call:

```
CALL FIDASPILSSETJAC (FLAG, IER)
```

with `FLAG` $\neq 0$. The return flag `IER` is 0 if successful, or negative if a memory error occurred.

If preconditioning is to be done, then the user must call

```
CALL FIDASPILSSETPREC (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 must supply preconditioner routines `FIDAPSET` and `FIDAPSOL`.

[S][P] User-supplied routines for SPGMR/SPBCG/SPTFQMR

With treatment of the linear systems by any of the Krylov iterative solvers, there are three optional user-supplied routines — `FIDAJTIMES`, `FIDAPSOL`, and `FIDAPSET`. The specifications for these routines are given below.

As an option when using any of the Krylov iterative solvers, the user may supply a routine that computes the product of the system Jacobian $J = \partial F/\partial y + \alpha \partial F/\partial \dot{y}$ and a given vector v. If supplied, it must have the following form:

```
 SUBROUTINE FIDAJTIMES(T, Y, YP, R, V, FJV, CJ, EWT, H,
&                      IPAR, RPAR, WK1, WK2, IER)
 DIMENSION Y(*), YP(*), R(*), V(*), FJV(*), EWT(*),
&          IPAR(*), RPAR(*), WK1(*), WK2(*)
```

This routine must compute the product vector Jv, where the vector v is stored in `V`, and store the product in `FJV`. On return, set `IER = 0` if `FIDAJTIMES` was successful, and nonzero otherwise. The vectors `W1K` and `WK2`, of length `NEQ`, are provided as work space for use in `FIDAJTIMES`. The input arguments `T`, `Y`, `YP`, `R`, and `CJ` are the current values of t, y, $\dot{y}$, $F(t, y, \dot{y})$, and α, respectively. The arrays `IPAR` (of integers) and `RPAR` (of reals) contain user data and are the same as those passed to `FIDAMALLOC`.

If the user's `FIDAJTIMES` uses difference quotient approximations, it may need to use the error weight array `EWT` and current stepsize `H` in the calculation of suitable increments. It may also need

the unit roundoff, which can be obtained as the optional output ROUT(6), passed from the calling program to this routine using COMMON.

If preconditioning is to be included, the following routine must be supplied, for solution of the preconditioner linear system:

```
 SUBROUTINE FIDAPSOL(T, Y, YP, R, RV, ZV, CJ, DELTA, EWT,
&                    IPAR, RPAR, WK1, IER)
 DIMENSION Y(*), YP(*), R(*), RV(*), ZV(*), EWT(*),
&          IPAR(*), RPAR(*), WK1(*)
```

It must solve the preconditioner linear system $Pz = r$, where $r =$ RV is input, and store the solution z in ZV. Here P is the left preconditioner. The input arguments T, Y, YP, R, and CJ are the current values of t, y, $\dot{y}$, $F(t, y, \dot{y})$, and α, respectively. On return, set IER = 0 if FIDAPSOL was successful, set IER positive if a recoverable error occurred, and set IER negative if a non-recoverable error occurred.

The arguments EWT and DELTA are input and provide the error weight array and a scalar tolerance, respectively, for use by FIDAPSOL if it uses an iterative method in its solution. In that case, the residual vector $\rho = r - Pz$ of the system should be made less than DELTA in weighted ℓ_2 norm, i.e. $\sqrt{\sum(\rho_i * \texttt{EWT}[i])^2} <$ DELTA. The argument WK1 is a work array of length NEQ for use by this routine. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC.

If the user's preconditioner requires that any Jacobian-related data be evaluated or preprocessed, then the following routine is to be used for the evaluation and preprocessing of the preconditioner:

```
 SUBROUTINE FIDAPSET(T, Y, YP, R, CJ, EWT, H,
&                    IPAR, RPAR, WK1, WK2, WK3, IER)
 DIMENSION Y(*), YP(*), R(*), EWT(*),
&          IPAR(*), RPAR(*), WK1(*), WK2(*), WK3(*)
```

It must perform any evaluation of Jacobian-related data and preprocessing needed for the solution of the preconditioner linear systems by FIDAPSOL. The input arguments T, Y, YP, R, and CJ are the current values of t, y, $\dot{y}$, $F(t, y, \dot{y})$, and α, respectively. On return, set IER = 0 if FIDAPSET 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 FIDAMALLOC.

If the user's FIDAPSET uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. It may also need the unit roundoff, which can be obtained as the optional output ROUT(6), passed from the calling program to this routine using COMMON.

If the user calls FIDASPILSSETPREC, the subroutine FIDAPSET must be provided, even if it is not needed, and it must return IER = 0.

6. **Correct initial values**

Optionally, to correct the initial values y and/or $\dot{y}$, make the call

```
CALL FIDACALCIC (ICOPT, TOUT1, IER)
```

(See §2.1 for details.) The arguments are as follows: ICOPT is 1 for initializing the algebraic components of y and differential components of $\dot{y}$, or 2 for initializing all of y. IER is an error return flag, which is 0 for success, or negative for a failure (see IDACalcIC return values).

7. **Problem solution**

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

   ```
   CALL FIDASOLVE (TOUT, T, Y, YP, ITASK, IER)
   ```

   The arguments are as follows. TOUT specifies the next value of t at which a solution is desired (input). T is the value of t reached by the solver on output. Y is an array containing the computed solution vector y on output. YP is an array containing the computed solution vector $\dot{y}$ on output. ITASK is a task indicator and should be set to 1 for normal mode (overshoot TOUT and interpolate), or to 2 for one-step mode (return after each internal step taken). IER is a completion flag and will be set to a positive value upon successful return or to a negative value if an error occurred. These values correspond to the IDASolve returns (see §4.5.6 and §B.2). The current values of the optional outputs are available in IOUT and ROUT (see Table 5.2).

8. **Additional solution output**

   After a successful return from FIDASOLVE, the routine FIDAGETDKY may be called to get interpolated values of y or any derivative $d^k y/dt^k$ for k not exceeding the current method order, and for any value of t in the last internal step taken by IDA. The call is as follows:

   ```
   CALL FIDAGETDKY (T, K, DKY, IER)
   ```

   where T is the input value of t at which solution derivative is desired, K is the derivative order, and DKY is an array containing the computed vector $y^{(K)}(t)$ on return. The value of T must lie between TCUR - HLAST and TCUR. The value of K must satisfy $0 \leq K \leq$QLAST. (See the optional outputs for TCUR, HLAST, and QLAST.) The return flag IER is set to 0 upon successful return, or to a negative value to indicate an illegal input.

9. **Problem reinitialization**

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

   ```
   CALL FIDAREINIT (T0, Y0, YP0, IATOL, RTOL, ATOL, IER)
   ```

   The arguments have the same names and meanings as those of FIDAMALLOC. FIDAREINIT performs the same initializations as FIDAMALLOC, but does no memory allocation, using instead the existing internal memory created by the previous FIDAMALLOC call.

   Following this call, a call to specify the linear system solver must be made if the choice of linear solver is being changed. Otherwise, a call to reinitialize the linear solver last used may or may not be needed, depending on changes in the inputs to it.

   In the case of the BAND solver, for any change in the half-bandwidth parameters, call FIDABAND (or FIDALAPACKBAND) as described above.

   In the case of SPGMR, for a change of inputs other than MAXL, make the call

   ```
   CALL FIDASPGMRREINIT (IGSTYPE, MAXRS, EPLIFAC, DQINCFAC, IER)
   ```

   which reinitializes SPGMR without reallocating its memory. The arguments have the same names and meanings as those of FIDASPGMR. If MAXL is being changed, then call FIDASPGMR.

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

   ```
   CALL FIDASPBCGREINIT (MAXL, EPLIFAC, DQINCFAC, IER)
   ```

which reinitializes SPBCG without reallocating its memory. The arguments have the same names and meanings as those of FIDASPBCG.

In the case of SPTFQMR, for a change in any inputs, make the call

```
CALL FIDASPTFQMRREINIT (MAXL, EPLIFAC, DQINCFAC, IER)
```

which reinitializes SPTFQMR without reallocating its memory. The arguments have the same names and meanings as those of FIDASPTFQMR.

10. **Memory deallocation**

To free the internal memory created by the call to FIDAMALLOC, make the call

```
CALL FIDAFREE
```

5.5 FIDA optional input and output

In order to keep the number of user-callable FIDA interface routines to a minimum, optional inputs to the IDA solver are passed through only three routines: FIDASETIIN for integer optional inputs, FIDASETRIN for real optional inputs, and FIDASETVIN for real vector (array) optional inputs. These functions should be called as follows:

```
CALL FIDASETIIN(KEY, IVAL, IER)
CALL FIDASETRIN(KEY, RVAL, IER)
CALL FIDASETVIN(KEY, VVAL, IER)
```

where KEY is a quoted string indicating which optoinal input is set (see Table 5.1), IVAL is the input integer value, RVAL is the input real value (scalar), VVAL is the input real array, and IER is an integer return flag which is set to 0 on success and a negative value if a failure occurred. IVAL should be declared so as to match C type long int.

When using FIDASETVIN to specify the variable types (KEY = 'ID_VEC') the components in the array VVAL must be 1.0 to indicate a differential variable, or 0.0 to indicate an algebraic variable. Note that this array is required only if FIDACALCIC is to be called with ICOPT = 1, or if algebraic variables are suppressed from the error test (indicated using FIDASETIIN with KEY = 'SUPPRESS_ALG'). When using FIDASETVIN to specify optional constraints on the solution vector (KEY = 'CONSTR_VEC') the components in the array VVAL should be one of -2.0, -1.0, 0.0, 1.0, or 2.0. See the description of IDASetConstraints (§4.5.7.1) for details.

The optional outputs from the IDA solver are accessed not through individual functions, but rather through a pair of arrays, IOUT (integer type) of dimension at least 21, and ROUT (real type) of dimension at least 6. These arrays are owned (and allocated) by the user and are passed as arguments to FIDAMALLOC. Table 5.2 lists the entries in these two arrays and specifies the optional variable as well as the IDA function which is actually called to extract the optional output.

For more details on the optional inputs and outputs, see §4.5.7 and §4.5.9.

In addition to the optional inputs communicated through FIDASET* calls and the optional outputs extracted from IOUT and ROUT, the following user-callable routines are available:

To reset the tolerances at any time, make the following call:

```
CALL FIDATOLREINIT (IATOL, RTOL, ATOL, IER)
```

The tolerance arguments have the same names and meanings as those of FIDAMALLOC. The error return flag IER is 0 if successful, and negative if there was a memory failure or illegal input.

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

```
CALL FIDAGETERRWEIGHTS (EWT, IER)
```

Table 5.1: Keys for setting FIDA optional inputs

Integer optional inputs (FIDASETIIN)

Key	Optional input	Default value
MAX_ORD	Maximum LMM method order	5
MAX_NSTEPS	Maximum no. of internal steps before t_{out}	500
MAX_ERRFAIL	Maximum no. of error test failures	10
MAX_NITERS	Maximum no. of nonlinear iterations	4
MAX_CONVFAIL	Maximum no. of convergence failures	10
SUPPRESS_ALG	Suppress alg. vars. from error test (1 = TRUE)	0 (= FALSE)
MAX_NSTEPS_IC	Maximum no. of steps for IC calc.	5
MAX_NITERS_IC	Maximum no. of Newton iterations for IC calc.	10
MAX_NJE_IC	Maximum no. of Jac. evals fo IC calc.	4
LS_OFF_IC	Turn off line search (1 = TRUE)	0 (= FALSE)

Real optional inputs (FIDASETRIN)

Key	Optional input	Default value
INIT_STEP	Initial step size	estimated
MAX_STEP	Maximum absolute step size	∞
STOP_TIME	Value of t_{stop}	undefined
NLCONV_COEF	Coeff. in the nonlinear conv. test	0.33
NLCONV_COEF_IC	Coeff. in the nonlinear conv. test for IC calc.	0.0033
STEP_TOL_IC	Lower bound on Newton step for IC calc.	uround$^{2/3}$

Real vector optional inputs (FIDASETVIN)

Key	Optional input	Default value
ID_VEC	Differential/algebraic component types	undefined
CONSTR_VEC	Inequality constraints on solution	undefined

Table 5.2: Description of the FIDA optional output arrays IOUT and ROUT

Integer output array IOUT

Index	Optional output	IDA function
IDA main solver		
1	LENRW	IDAGetWorkSpace
2	LENIW	IDAGetWorkSpace
3	NST	IDAGetNumSteps
4	NRE	IDAGetNumResEvals
5	NETF	IDAGetNumErrTestFails
6	NNCFAILS	IDAGetNonlinSolvConvFails
7	NNI	IDAGetNumNonlinSolvIters
8	NSETUPS	IDAGetNumLinSolvSetups
9	QLAST	IDAGetLastOrder
10	QCUR	IDAGetCurrentOrder
11	NBCKTRKOPS	IDAGetNumBacktrackOps
12	NGE	IDAGetNumGEvals
IDADLS linear solvers		
13	LENRWLS	IDADlsGetWorkSpace
14	LENIWLS	IDADlsGetWorkSpace
15	LS_FLAG	IDADlsGetLastFlag
16	NRELS	IDADlsGetNumResEvals
17	NJE	IDADlsGetNumJacEvals
IDASLS linear solvers		
14	LS_FLAG	IDASlsGetLastFlag
16	NJE	IDASlsGetNumJacEvals
IDASPILS linear solvers		
13	LENRWLS	IDASpilsGetWorkSpace
14	LENIWLS	IDASpilsGetWorkSpace
15	LS_FLAG	IDASpilsGetLastFlag
16	NRELS	IDASpilsGetNumResEvals
17	NJE	IDASpilsGetNumJtimesEvals
18	NPE	IDASpilsGetNumPrecEvals
19	NPS	IDASpilsGetNumPrecSolves
20	NLI	IDASpilsGetNumLinIters
21	NCFL	IDASpilsGetNumConvFails

Real output array ROUT

Index	Optional output	IDA function
1	HO_USED	IDAGetActualInitStep
2	HLAST	IDAGetLastStep
3	HCUR	IDAGetCurrentStep
4	TCUR	IDAGetCurrentTime
5	TOLFACT	IDAGetTolScaleFactor
6	UROUND	unit roundoff

This computes the EWT array, normally defined by Eq. (2.6). The array EWT, of length NEQ or NLOCAL, must already have been declared by the user. The error return flag IER is zero if successful, and negative if there was a memory error.

To obtain the estimated local errors, following a successful call to FIDASOLVE, make the following call:

```
CALL FIDAGETESTLOCALERR (ELE, IER)
```

This computes the ELE array of estimated local errors as of the last step taken. The array ELE must already have been declared by the user. The error return flag IER is zero if successful, and negative if there was a memory error.

5.6 Usage of the FIDAROOT interface to rootfinding

The FIDAROOT interface package allows programs written in FORTRAN to use the rootfinding feature of the IDA solver module. The user-callable functions in FIDAROOT, with the corresponding IDA functions, are as follows:

- FIDAROOTINIT interfaces to IDARootInit.
- FIDAROOTINFO interfaces to IDAGetRootInfo.
- FIDAROOTFREE interfaces to IDARootFree.

Note that, at this time FIDAROOT does not provide support to specify the direction of zero-crossing that is to be monitored. Instead, all roots are considered. However, the actual direction of zero-crossing is reported (through the sign of the non-zero elements in the array INFO returned by FIDAROTINFO).

In order to use the rootfinding feature of IDA, the following call must be made, after calling FIDAMALLOC but prior to calling FIDASOLVE, to allocate and initialize memory for the FIDAROOT module:

```
CALL FIDAROOTINIT (NRTFN, IER)
```

The arguments are as follows: NRTFN is the number of root functions. IER is a return completion flag; its values are 0 for success, -1 if the IDA memory was NULL, and -14 if a memory allocation failed.

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

```
SUBROUTINE FIDAROOTFN (T, Y, YP, G, IPAR, RPAR, IER)
DIMENSION Y(*), YP(*), G(*), IPAR(*), RPAR(*)
```

It must set the G array, of length NRTFN, with components $g_i(t, y, \dot{y})$, as a function of T $= t$ and the arrays Y $= y$ and YP $= \dot{y}$. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. Set IER on 0 if successful, or on a non-zero value if an error occurred.

When making calls to FIDASOLVE to solve the DAE system, the occurrence of a root is flagged by the return value IER $= 2$. In that case, if NRTFN > 1, the functions g_i which were found to have a root can be identified by making the following call:

```
CALL FIDAROOTINFO (NRTFN, INFO, IER)
```

The arguments are as follows: NRTFN is the number of root functions. INFO is an integer array of length NRTFN with root information. IER is a return completion flag; its values are 0 for success, negative if there was a memory failure. The returned values of INFO(i) (i$= 1, \ldots,$ NRTFN) are 0 or ± 1, such that INFO(i) $= +1$ if $g_{\mathtt{i}}$ was found to have a root and $g_{\mathtt{i}}$ is increasing, INFO(i) $= -1$ if $g_{\mathtt{i}}$ was found to have a root and $g_{\mathtt{i}}$ is dereasing, and INFO(i) $= 0$ otherwise.

The total number of calls made to the root function FIDAROOTFN, denoted NGE, can be obtained from IOUT(12). If the FIDA/IDA memory block is reinitialized to solve a different problem via a call to FIDAREINIT, then the counter NGE is reset to zero.

To free the memory resources allocated by a prior call to FIDAROOTINIT, make the following call:

```
CALL FIDAROOTFREE
```

See §4.5.5 for additional information on the rootfinding feature.

5.7 Usage of the FIDABBD interface to IDABBDPRE

The FIDABBD interface sub-module is a package of C functions which, as part of the FIDA interface module, support the use of the IDA solver with the parallel NVECTOR_PARALLEL module, in a combination of any of the Krylov iterative solver modules with the IDABBDPRE preconditioner module (see §4.7).

The user-callable functions in this package, with the corresponding IDA and IDABBDPRE functions, are as follows:

- `FIDABBDINIT` interfaces to `IDABBDPrecAlloc`.
- `FIDABBDREINIT` interfaces to `IDABBDPrecReInit`.
- `FIDABBDOPT` interfaces to IDABBDPRE optional output functions.
- `FIDABBDFREE` interfaces to `IDABBDPrecFree`.

In addition to the FORTRAN residual function `FIDARESFUN`, the user-supplied functions used by this package, are listed below, each with the corresponding interface function which calls it (and its type within IDABBDPRE or IDA):

FIDABBD routine (FORTRAN)	IDA function (C)	IDA function type
`FIDAGLOCFN`	`FIDAgloc`	`IDABBDLocalFn`
`FIDACOMMFN`	`FIDAcfn`	`IDABBDCommFn`
`FIDAJTIMES`	`FIDAJtimes`	`IDASpilsJacTimesVecFn`

As with the rest of the FIDA routines, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program. Additionally, based on flags discussed above in §5.3, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `fidabbd.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §5.4 are grayed-out.

1. **Residual function specification**
2. **NVECTOR module initialization**
3. **Problem specification**
4. **Set optional inputs**
5. **Iterative linear solver specification**
6. **BBD preconditioner initialization**

   To initialize the IDABBDPRE preconditioner, make the following call:

   ```
   CALL FIDABBDINIT (NLOCAL, MUDQ, MLDQ, MU, ML, DQRELY, IER)
   ```

   The arguments are as follows. `NLOCAL` is the local size of vectors on this processor. `MUDQ` and `MLDQ` are the upper and lower half-bandwidths to be used in the computation of the local Jacobian blocks by difference quotients. These may be smaller than the true half-bandwidths of the Jacobian of the local block of G, when smaller values may provide greater efficiency. `MU` and `ML` are the upper and lower half-bandwidths of the band matrix that is retained as an approximation of the local Jacobian block. These may be smaller than `MUDQ` and `MLDQ`. `DQRELY` is the relative increment factor in y for difference quotients (optional). A value of 0.0 indicates the default, $\sqrt{\text{unit roundoff}}$. `IER` is a return completion flag. A value of 0 indicates success, while a value of -1 indicates that a memory failure occurred or that an input had an illegal value.

7. **Problem solution**

8. IDABBDPRE **Optional outputs**

   Optional outputs specific to the SPGMR, SPBCG, or SPTFQMR solver are listed in Table 5.2. To obtain the optional outputs associated with the IDABBDPRE module, make the following call:

   ```
   CALL FIDABBDOPT (LENRWBBD, LENIWBBD, NGEBBD)
   ```

   The arguments should be consistent with C type `long int`. Their returned values are as follows: `LENRWBBD` is the length of real preconditioner work space, in `realtype` words. `LENIWBBD` is the length of integer preconditioner work space, in integer words. Both of these sizes are local to the current processor. `NGEBBD` is the number of $G(t, y, \dot{y})$ evaluations (calls to `FIDALOCFN`) so far.

9. **Problem reinitialization**

   If a sequence of problems of the same size is being solved using the SPGMR, SPBCG, or SPTFQMR linear solver in combination with the IDABBDPRE preconditioner, then the IDA package can be reinitialized for the second and subsequent problems by calling `FIDAREINIT`, following which a call to `FIDABBDINIT` may or may not be needed. If the input arguments are the same, no `FIDABBDINIT` call is needed. If there is a change in input arguments other than `MU`, `ML`, or `MAXL`, then the user program should make the call

   ```
   CALL FIDABBDREINIT (NLOCAL, MUDQ, MLDQ, DQRELY, IER)
   ```

   This reinitializes the IDABBDPRE preconditioner, but without reallocating its memory. The arguments of the `FIDABBDREINIT` routine have the same names and meanings as those of `FIDABBDINIT`. If the value of `MU` or `ML` is being changed, then a call to `FIDABBDINIT` must be made. Finally, if `MAXL` is being changed, then a call to `FIDASPGMR`, `FIDASPBCG`, or `FIDASPTFQMR` must be made; in this case the linear solver memory is reallocated.

10. **Memory deallocation**

    (The memory allocated for the FIDABBD module is deallocated automatically by `FIDAFREE`.)

11. **User-supplied routines**

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

    ```
    SUBROUTINE FIDAGLOCFN (NLOC, T, YLOC, YPLOC, GLOC, IPAR, RPAR, IER)
    DIMENSION YLOC(*), YPLOC(*), GLOC(*), IPAR(*), RPAR(*)
    ```

    This routine is to evaluate the function $G(t, y, \dot{y})$ approximating F (possibly identical to F), in terms of `T` $= t$, and the arrays `YLOC` and `YPLOC` (of length `NLOC`), which are the sub-vectors of y and $\dot{y}$ local to this processor. The resulting (local) sub-vector is to be stored in the array `GLOC`. `IER` is a return flag that should be set to 0 if successful, to 1 (for a recoverable error), or to -1 (for a non-recoverable error). The arrays `IPAR` (of integers) and `RPAR` (of reals) contain user data and are the same as those passed to `FIDAMALLOC`.

    ```
    SUBROUTINE FIDACOMMFN (NLOC, T, YLOC, YPLOC, IPAR, RPAR, IER)
    DIMENSION YLOC(*), YPLOC(*), IPAR(*), RPAR(*)
    ```

    This routine is to perform the inter-processor communication necessary for the `FIDAGLOCFN` routine. Each call to `FIDACOMMFN` is preceded by a call to the residual routine `FIDARESFUN` with the same arguments `T`, `YLOC`, and `YPLOC`. Thus `FIDACOMMFN` can omit any communications done by `FIDARESFUN` if relevant to the evaluation of `GLOC`. The arrays `IPAR` (of integers) and `RPAR` (of reals)

contain user data and are the same as those passed to `FIDAMALLOC`. `IER` is a return flag that should be set to 0 if successful, to 1 (for a recoverable error), or to -1 (for a non-recoverable error).

The subroutine `FIDACOMMFN` must be supplied even if it is empty, and it must return `IER = 0`.

Optionally, the user can supply a routine `FIDAJTIMES` for the evaluation of Jacobian-vector products, as described above in step 5 in §5.4.

Chapter 6

Description of the NVECTOR module

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type `N_Vector`) through a set of operations defined by the particular NVECTOR implementation. Users can provide their own specific implementation of the NVECTOR module, or use one of four provided within SUNDIALS – a serial implementation and three parallel implementations. The generic operations are described below. In the sections following, the implementations provided with SUNDIALS are described.

The generic `N_Vector` type is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the vector, and an *ops* field pointing to a structure with generic vector operations. The type `N_Vector` is defined as

```
typedef struct _generic_N_Vector *N_Vector;

struct _generic_N_Vector {
    void *content;
    struct _generic_N_Vector_Ops *ops;
};
```

The `_generic_N_Vector_Ops` structure is essentially a list of pointers to the various actual vector operations, and is defined as

```
struct _generic_N_Vector_Ops {
  N_Vector    (*nvclone)(N_Vector);
  N_Vector    (*nvcloneempty)(N_Vector);
  void        (*nvdestroy)(N_Vector);
  void        (*nvspace)(N_Vector, long int *, long int *);
  realtype*   (*nvgetarraypointer)(N_Vector);
  void        (*nvsetarraypointer)(realtype *, N_Vector);
  void        (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
  void        (*nvconst)(realtype, N_Vector);
  void        (*nvprod)(N_Vector, N_Vector, N_Vector);
  void        (*nvdiv)(N_Vector, N_Vector, N_Vector);
  void        (*nvscale)(realtype, N_Vector, N_Vector);
  void        (*nvabs)(N_Vector, N_Vector);
  void        (*nvinv)(N_Vector, N_Vector);
  void        (*nvaddconst)(N_Vector, realtype, N_Vector);
  realtype    (*nvdotprod)(N_Vector, N_Vector);
  realtype    (*nvmaxnorm)(N_Vector);
  realtype    (*nvwrmsnorm)(N_Vector, N_Vector);
```

```
  realtype    (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvmin)(N_Vector);
  realtype    (*nvwl2norm)(N_Vector, N_Vector);
  realtype    (*nvl1norm)(N_Vector);
  void        (*nvcompare)(realtype, N_Vector, N_Vector);
  booleantype (*nvinvtest)(N_Vector, N_Vector);
  booleantype (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvminquotient)(N_Vector, N_Vector);
};
```

The generic NVECTOR module defines and implements the vector operations acting on `N_Vector`. These routines are nothing but wrappers for the vector operations defined by a particular NVECTOR implementation, which are accessed through the *ops* field of the `N_Vector` structure. To illustrate this point we show below the implementation of a typical vector operation from the generic NVECTOR module, namely `N_VScale`, which performs the scaling of a vector `x` by a scalar `c`:

```
void N_VScale(realtype c, N_Vector x, N_Vector z)
{
   z->ops->nvscale(c, x, z);
}
```

Table 6.1 contains a complete list of all vector operations defined by the generic NVECTOR module.

Finally, note that the generic NVECTOR module defines the functions `N_VCloneVectorArray` and `N_VCloneEmptyVectorArray`. Both functions create (by cloning) an array of `count` variables of type `N_Vector`, each of the same type as an existing `N_Vector`. Their prototypes are

```
N_Vector *N_VCloneVectorArray(int count, N_Vector w);
N_Vector *N_VCloneEmptyVectorArray(int count, N_Vector w);
```

and their definitions are based on the implementation-specific `N_VClone` and `N_VCloneEmpty` operations, respectively.

An array of variables of type `N_Vector` can be destroyed by calling `N_VDestroyVectorArray`, whose prototype is

```
void N_VDestroyVectorArray(N_Vector *vs, int count);
```

and whose definition is based on the implementation-specific `N_VDestroy` operation.

A particular implementation of the NVECTOR module must:

- Specify the *content* field of `N_Vector`.
- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different `N_Vector` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free an `N_Vector` with the new *content* field and with *ops* pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `N_Vector` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the *content* field of the newly defined `N_Vector`.

Table 6.1: Description of the NVECTOR operations

Name	Usage and Description
N_VClone	`v = N_VClone(w);` Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not copy the vector, but rather allocates storage for the new vector.
N_VCloneEmpty	`v = N_VCloneEmpty(w);` Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not allocate storage for data.
N_VDestroy	`N_VDestroy(v);` Destroys the `N_Vector v` and frees memory allocated for its internal data.
N_VSpace	`N_VSpace(nvSpec, &lrw, &liw);` Returns storage requirements for one `N_Vector`. `lrw` contains the number of realtype words and `liw` contains the number of integer words. This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied NVECTOR module if that information is not of interest.
N_VGetArrayPointer	`vdata = N_VGetArrayPointer(v);` Returns a pointer to a `realtype` array from the `N_Vector v`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the solver-specific interfaces to the dense and banded (serial) linear solvers, the sparse linear solvers (serial and threaded), and in the interfaces to the banded (serial) and band-block-diagonal (parallel) preconditioner modules provided with SUNDIALS.
N_VSetArrayPointer	`N_VSetArrayPointer(vdata, v);` Overwrites the data in an `N_Vector` with a given array of `realtype`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVECTOR module for a parallel environment.
N_VLinearSum	`N_VLinearSum(a, x, b, y, z);` Performs the operation $z = ax + by$, where a and b are `realtype` scalars and x and y are of type `N_Vector`: $z_i = ax_i + by_i$, $i = 0, \ldots, n-1$.
N_VConst	`N_VConst(c, z);` Sets all components of the `N_Vector z` to `realtype c`: $z_i = c$, $i = 0, \ldots, n-1$.

continued on next page

continued from last page

Name	Usage and Description
N_VProd	`N_VProd(x, y, z);` Sets the `N_Vector z` to be the component-wise product of the `N_Vector` inputs `x` and `y`: $z_i = x_i y_i$, $i = 0, \ldots, n-1$.
N_VDiv	`N_VDiv(x, y, z);` Sets the `N_Vector z` to be the component-wise ratio of the `N_Vector` inputs `x` and `y`: $z_i = x_i/y_i$, $i = 0, \ldots, n-1$. The y_i may not be tested for 0 values. It should only be called with a `y` that is guaranteed to have all nonzero components.
N_VScale	`N_VScale(c, x, z);` Scales the `N_Vector x` by the `realtype` scalar `c` and returns the result in `z`: $z_i = cx_i$, $i = 0, \ldots, n-1$.
N_VAbs	`N_VAbs(x, z);` Sets the components of the `N_Vector z` to be the absolute values of the components of the `N_Vector x`: $y_i = \lvert x_i \rvert$, $i = 0, \ldots, n-1$.
N_VInv	`N_VInv(x, z);` Sets the components of the `N_Vector z` to be the inverses of the components of the `N_Vector x`: $z_i = 1.0/x_i$, $i = 0, \ldots, n-1$. This routine may not check for division by 0. It should be called only with an `x` which is guaranteed to have all nonzero components.
N_VAddConst	`N_VAddConst(x, b, z);` Adds the `realtype` scalar `b` to all components of `x` and returns the result in the `N_Vector z`: $z_i = x_i + b$, $i = 0, \ldots, n-1$.
N_VDotProd	`d = N_VDotProd(x, y);` Returns the value of the ordinary dot product of `x` and `y`: $d = \sum_{i=0}^{n-1} x_i y_i$.
N_VMaxNorm	`m = N_VMaxNorm(x);` Returns the maximum norm of the `N_Vector x`: $m = \max_i \lvert x_i \rvert$.
N_VWrmsNorm	`m = N_VWrmsNorm(x, w)` Returns the weighted root-mean-square norm of the `N_Vector x` with `realtype` weight vector `w`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i)^2\right)/n}$.
N_VWrmsNormMask	`m = N_VWrmsNormMask(x, w, id);` Returns the weighted root mean square norm of the `N_Vector x` with `realtype` weight vector `w` built using only the elements of `x` corresponding to nonzero elements of the `N_Vector id`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i \mathrm{sign}(id_i))^2\right)/n}$.
N_VMin	`m = N_VMin(x);` Returns the smallest element of the `N_Vector x`: $m = \min_i x_i$.

continued on next page

continued from last page

Name	Usage and Description
N_VWL2Norm	`m = N_VWL2Norm(x, w);` Returns the weighted Euclidean ℓ_2 norm of the `N_Vector x` with `realtype` weight vector `w`: $m = \sqrt{\sum_{i=0}^{n-1}(x_i w_i)^2}$.
N_VL1Norm	`m = N_VL1Norm(x);` Returns the ℓ_1 norm of the `N_Vector x`: $m = \sum_{i=0}^{n-1} \|x_i\|$.
N_VCompare	`N_VCompare(c, x, z);` Compares the components of the `N_Vector x` to the `realtype` scalar `c` and returns an `N_Vector z` such that: $z_i = 1.0$ if $\|x_i\| \geq c$ and $z_i = 0.0$ otherwise.
N_VInvTest	`t = N_VInvTest(x, z);` Sets the components of the `N_Vector z` to be the inverses of the components of the `N_Vector x`, with prior testing for zero values: $z_i = 1.0/x_i$, $i = 0, \ldots, n-1$. This routine returns a boolean assigned to `TRUE` if all components of `x` are nonzero (successful inversion) and returns `FALSE` otherwise.
N_VConstrMask	`t = N_VConstrMask(c, x, m);` Performs the following constraint tests: $x_i > 0$ if $c_i = 2$, $x_i \geq 0$ if $c_i = 1$, $x_i \leq 0$ if $c_i = -1$, $x_i < 0$ if $c_i = -2$. There is no constraint on x_i if $c_i = 0$. This routine returns a boolean assigned to `FALSE` if any element failed the constraint test and assigned to `TRUE` if all passed. It also sets a mask vector `m`, with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.
N_VMinQuotient	`minq = N_VMinQuotient(num, denom);` This routine returns the minimum of the quotients obtained by termwise dividing $\texttt{num}_i$ by $\texttt{denom}_i$. A zero element in `denom` will be skipped. If no such quotients are found, then the large value `BIG_REAL` (defined in the header file `sundials_types.h`) is returned.

6.1 The NVECTOR_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of *data*.

```
struct _N_VectorContent_Serial {
  long int length;
  booleantype own_data;
  realtype *data;
};
```

The following five macros are provided to access the content of an NVECTOR_SERIAL vector. The suffix `_S` in the names denotes the serial version.

- `NV_CONTENT_S`

  This routine gives access to the contents of the serial vector `N_Vector`.

The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_S(v) ( (N_VectorContent_Serial)(v->content) )
```

- `NV_OWN_DATA_S`, `NV_DATA_S`, `NV_LENGTH_S`

  These macros give individual access to the parts of the content of a serial `N_Vector`.

  The assignment `v_data = NV_DATA_S(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector v`. The assignment `NV_DATA_S(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

  The assignment `v_len = NV_LENGTH_S(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_S(v) = len_v` sets the length of `v` to be `len_v`.

  Implementation:

  ```
  #define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
  #define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
  #define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
  ```

- `NV_Ith_S`

  This macro gives access to the individual components of the data array of an `N_Vector`.

  The assignment `r = NV_Ith_S(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_S(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
  ```

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in Table 6.1. Their names are obtained from those in Table 6.1 by appending the suffix `_Serial`. The module NVECTOR_SERIAL provides the following additional user-callable routines:

- `N_VNew_Serial`

  This function creates and allocates memory for a serial `N_Vector`. Its only argument is the vector length.

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

- `N_VNewEmpty_Serial`

  This function creates a new serial `N_Vector` with an empty (`NULL`) data array.

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

- `N_VMake_Serial`

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

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

- `N_VCloneVectorArray_Serial`

  This function creates (by cloning) an array of `count` serial vectors.

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

- `N_VCloneEmptyVectorArray_Serial`

  This function creates (by cloning) an array of `count` serial vectors, each with an empty (`NULL`) data array.

  ```
  N_Vector *N_VCloneEmptyVectorArray_Serial(int count, N_Vector w);
  ```

- N_VDestroyVectorArray_Serial

  This function frees memory allocated for the array of count variables of type N_Vector created with N_VCloneVectorArray_Serial or with N_VCloneEmptyVectorArray_Serial.

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

- N_VPrint_Serial

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

  ```
  void N_VPrint_Serial(N_Vector v);
  ```

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the component array via v_data = NV_DATA_S(v) and then access v_data[i] within the loop than it is to use NV_Ith_S(v,i) within the loop.

- N_VNewEmpty_Serial, N_VMake_Serial, and N_VCloneEmptyVectorArray_Serial set the field *own_data* = FALSE. N_VDestroy_Serial and N_VDestroyVectorArray_Serial will not attempt to free the pointer *data* for any N_Vector with *own_data* set to FALSE. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- To maximize efficiency, vector operations in the NVECTOR_SERIAL implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

6.2 The NVECTOR_PARALLEL implementation

The NVECTOR_PARALLEL implementation of the NVECTOR module provided with SUNDIALS is based on MPI. It defines the *content* field of N_Vector to be a structure containing the global and local lengths of the vector, a pointer to the beginning of a contiguous local data array, an MPI communicator, an a boolean flag *own_data* indicating ownership of the data array *data*.

```
struct _N_VectorContent_Parallel {
  long int local_length;
  long int global_length;
  booleantype own_data;
  realtype *data;
  MPI_Comm comm;
};
```

The following seven macros are provided to access the content of a NVECTOR_PARALLEL vector. The suffix _P in the names denotes the distributed memory parallel version.

- NV_CONTENT_P

  This macro gives access to the contents of the parallel vector N_Vector.

  The assignment v_cont = NV_CONTENT_P(v) sets v_cont to be a pointer to the N_Vector content structure of type struct _N_VectorParallelContent.

  Implementation:

  ```
  #define NV_CONTENT_P(v) ( (N_VectorContent_Parallel)(v->content) )
  ```

- NV_OWN_DATA_P, NV_DATA_P, NV_LOCLENGTH_P, NV_GLOBLENGTH_P

  These macros give individual access to the parts of the content of a parallel N_Vector.

The assignment `v_data = NV_DATA_P(v)` sets `v_data` to be a pointer to the first component of the local data for the `N_Vector` `v`. The assignment `NV_DATA_P(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_llen = NV_LOCLENGTH_P(v)` sets `v_llen` to be the length of the local part of `v`. The call `NV_LENGTH_P(v) = llen_v` sets the local length of `v` to be `llen_v`.

The assignment `v_glen = NV_GLOBLENGTH_P(v)` sets `v_glen` to be the global length of the vector `v`. The call `NV_GLOBLENGTH_P(v) = glen_v` sets the global length of `v` to be `glen_v`.

Implementation:

```
#define NV_OWN_DATA_P(v)    ( NV_CONTENT_P(v)->own_data )
#define NV_DATA_P(v)        ( NV_CONTENT_P(v)->data )
#define NV_LOCLENGTH_P(v)   ( NV_CONTENT_P(v)->local_length )
#define NV_GLOBLENGTH_P(v)  ( NV_CONTENT_P(v)->global_length )
```

- `NV_COMM_P`

  This macro provides access to the MPI communicator used by the NVECTOR_PARALLEL vectors.

  Implementation:

  ```
  #define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
  ```

- `NV_Ith_P`

  This macro gives access to the individual components of the local data array of an `N_Vector`.

  The assignment `r = NV_Ith_P(v,i)` sets `r` to be the value of the `i`-th component of the local part of `v`. The assignment `NV_Ith_P(v,i) = r` sets the value of the `i`-th component of the local part of `v` to be `r`.

  Here i ranges from 0 to $n-1$, where n is the local length.

  Implementation:

  ```
  #define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
  ```

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Table 6.1 Their names are obtained from those in Table 6.1 by appending the suffix `_Parallel`. The module NVECTOR_PARALLEL provides the following additional user-callable routines:

- `N_VNew_Parallel`

  This function creates and allocates memory for a parallel vector.

  ```
  N_Vector N_VNew_Parallel(MPI_Comm comm,
                           long int local_length,
                           long int global_length);
  ```

- `N_VNewEmpty_Parallel`

  This function creates a new parallel `N_Vector` with an empty (`NULL`) data array.

  ```
  N_Vector N_VNewEmpty_Parallel(MPI_Comm comm,
                                long int local_length,
                                long int global_length);
  ```

- `N_VMake_Parallel`

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

```
N_Vector N_VMake_Parallel(MPI_Comm comm,
                          long int local_length,
                          long int global_length,
                          realtype *v_data);
```

- `N_VCloneVectorArray_Parallel`

  This function creates (by cloning) an array of `count` parallel vectors.

  ```
  N_Vector *N_VCloneVectorArray_Parallel(int count, N_Vector w);
  ```

- `N_VCloneEmptyVectorArray_Parallel`

  This function creates (by cloning) an array of `count` parallel vectors, each with an empty (`NULL`) data array.

  ```
  N_Vector *N_VCloneEmptyVectorArray_Parallel(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Parallel`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Parallel` or with `N_VCloneEmptyVectorArray_Parallel`.

  ```
  void N_VDestroyVectorArray_Parallel(N_Vector *vs, int count);
  ```

- `N_VPrint_Parallel`

  This function prints the content of a parallel vector to stdout.

  ```
  void N_VPrint_Parallel(N_Vector v);
  ```

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the local component array via `v_data = NV_DATA_P(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_P(v,i)` within the loop.
- `N_VNewEmpty_Parallel`, `N_VMake_Parallel`, and `N_VCloneEmptyVectorArray_Parallel` set the field *own_data* = `FALSE`. `N_VDestroy_Parallel` and `N_VDestroyVectorArray_Parallel` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- To maximize efficiency, vector operations in the NVECTOR_PARALLEL implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.3 The NVECTOR_OPENMP implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least $100,000$ before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The OpenMP NVECTOR implementation provided with SUNDIALS, NVECTOR_OPENMP, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using OpenMP.

```
struct _N_VectorContent_OpenMP {
  long int length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The following six macros are provided to access the content of an NVECTOR_OPENMP vector. The suffix _OMP in the names denotes the OpenMP version.

- NV_CONTENT_OMP

  This routine gives access to the contents of the OpenMP vector N_Vector.

  The assignment v_cont = NV_CONTENT_OMP(v) sets v_cont to be a pointer to the OpenMP N_Vector content structure.

  Implementation:

  ```
  #define NV_CONTENT_OMP(v) ( (N_VectorContent_OpenMP)(v->content) )
  ```

- NV_OWN_DATA_OMP, NV_DATA_OMP, NV_LENGTH_OMP, NV_NUM_THREADS_OMP

  These macros give individual access to the parts of the content of a OpenMP N_Vector.

  The assignment v_data = NV_DATA_OMP(v) sets v_data to be a pointer to the first component of the data for the N_Vector v. The assignment NV_DATA_OMP(v) = v_data sets the component array of v to be v_data by storing the pointer v_data.

  The assignment v_len = NV_LENGTH_OMP(v) sets v_len to be the length of v. On the other hand, the call NV_LENGTH_OMP(v) = len_v sets the length of v to be len_v.

  The assignment v_num_threads = NV_NUM_THREADS_OMP(v) sets v_num_threads to be the number of threads from v. On the other hand, the call NV_NUM_THREADS_OMP(v) = num_threads_v sets the number of threads for v to be num_threads_v.

  Implementation:

  ```
  #define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )
  #define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )
  #define NV_LENGTH_OMP(v) ( NV_CONTENT_OMP(v)->length )
  #define NV_NUM_THREADS_OMP(v) ( NV_CONTENT_OMP(v)->num_threads )
  ```

- NV_Ith_OMP

  This macro gives access to the individual components of the data array of an N_Vector.

  The assignment r = NV_Ith_OMP(v,i) sets r to be the value of the i-th component of v. The assignment NV_Ith_OMP(v,i) = r sets the value of the i-th component of v to be r.

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_OMP(v,i) ( NV_DATA_OMP(v)[i] )
  ```

The NVECTOR_OPENMP module defines OpenMP implementations of all vector operations listed in Table 6.1. Their names are obtained from those in Table 6.1 by appending the suffix _OpenMP. The module NVECTOR_OPENMP provides the following additional user-callable routines:

- N_VNew_OpenMP

  This function creates and allocates memory for a OpenMP N_Vector. Arguments are the vector length and number of threads.

  ```
  N_Vector N_VNew_OpenMP(long int vec_length, int num_threads);
  ```

- N_VNewEmpty_OpenMP

  This function creates a new OpenMP N_Vector with an empty (NULL) data array.

  ```
  N_Vector N_VNewEmpty_OpenMP(long int vec_length, int num_threads);
  ```

- N_VMake_OpenMP

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

  ```
  N_Vector N_VMake_OpenMP(long int vec_length, realtype *v_data, int num_threads);
  ```

- N_VCloneVectorArray_OpenMP

  This function creates (by cloning) an array of count OpenMP vectors.

  ```
  N_Vector *N_VCloneVectorArray_OpenMP(int count, N_Vector w);
  ```

- N_VCloneEmptyVectorArray_OpenMP

  This function creates (by cloning) an array of count OpenMP vectors, each with an empty (NULL) data array.

  ```
  N_Vector *N_VCloneEmptyVectorArray_OpenMP(int count, N_Vector w);
  ```

- N_VDestroyVectorArray_OpenMP

  This function frees memory allocated for the array of count variables of type N_Vector created with N_VCloneVectorArray_OpenMP or with N_VCloneEmptyVectorArray_OpenMP.

  ```
  void N_VDestroyVectorArray_OpenMP(N_Vector *vs, int count);
  ```

- N_VPrint_OpenMP

  This function prints the content of a OpenMP vector to stdout.

  ```
  void N_VPrint_OpenMP(N_Vector v);
  ```

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the component array via v_data = NV_DATA_OMP(v) and then access v_data[i] within the loop than it is to use NV_Ith_OMP(v,i) within the loop.

- N_VNewEmpty_OpenMP, N_VMake_OpenMP, and N_VCloneEmptyVectorArray_OpenMP set the field *own_data* = FALSE. N_VDestroy_OpenMP and N_VDestroyVectorArray_OpenMP will not attempt to free the pointer *data* for any N_Vector with *own_data* set to FALSE. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- To maximize efficiency, vector operations in the NVECTOR_OPENMP implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

6.4 The NVECTOR_PTHREADS implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The Pthreads NVECTOR implementation provided with SUNDIALS, NVECTOR_PTHREADS, defines the *content* field of N_Vector to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using POSIX threads (Pthreads).

```
struct _N_VectorContent_Pthreads {
  long int length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The following six macros are provided to access the content of an NVECTOR_PTHREADS vector. The suffix _PT in the names denotes the Pthreads version.

- NV_CONTENT_PT

  This routine gives access to the contents of the Pthreads vector N_Vector.

  The assignment v_cont = NV_CONTENT_PT(v) sets v_cont to be a pointer to the Pthreads N_Vector content structure.

  Implementation:

  ```
  #define NV_CONTENT_PT(v) ( (N_VectorContent_Pthreads)(v->content) )
  ```

- NV_OWN_DATA_PT, NV_DATA_PT, NV_LENGTH_PT, NV_NUM_THREADS_PT

  These macros give individual access to the parts of the content of a Pthreads N_Vector.

  The assignment v_data = NV_DATA_PT(v) sets v_data to be a pointer to the first component of the data for the N_Vector v. The assignment NV_DATA_PT(v) = v_data sets the component array of v to be v_data by storing the pointer v_data.

  The assignment v_len = NV_LENGTH_PT(v) sets v_len to be the length of v. On the other hand, the call NV_LENGTH_PT(v) = len_v sets the length of v to be len_v.

  The assignment v_num_threads = NV_NUM_THREADS_PT(v) sets v_num_threads to be the number of threads from v. On the other hand, the call NV_NUM_THREADS_PT(v) = num_threads_v sets the number of threads for v to be num_threads_v.

  Implementation:

  ```
  #define NV_OWN_DATA_PT(v) ( NV_CONTENT_PT(v)->own_data )
  #define NV_DATA_PT(v) ( NV_CONTENT_PT(v)->data )
  #define NV_LENGTH_PT(v) ( NV_CONTENT_PT(v)->length )
  #define NV_NUM_THREADS_PT(v) ( NV_CONTENT_PT(v)->num_threads )
  ```

- NV_Ith_PT

  This macro gives access to the individual components of the data array of an N_Vector.

  The assignment r = NV_Ith_PT(v,i) sets r to be the value of the i-th component of v. The assignment NV_Ith_PT(v,i) = r sets the value of the i-th component of v to be r.

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_PT(v,i) ( NV_DATA_PT(v)[i] )
  ```

The NVECTOR_PTHREADS module defines Pthreads implementations of all vector operations listed in Table 6.1. Their names are obtained from those in Table 6.1 by appending the suffix _Pthreads. The module NVECTOR_PTHREADS provides the following additional user-callable routines:

- N_VNew_Pthreads

  This function creates and allocates memory for a Pthreads N_Vector. Arguments are the vector length and number of threads.

  ```
  N_Vector N_VNew_Pthreads(long int vec_length, int num_threads);
  ```

- N_VNewEmpty_Pthreads

  This function creates a new Pthreads N_Vector with an empty (NULL) data array.

  ```
  N_Vector N_VNewEmpty_Pthreads(long int vec_length, int num_threads);
  ```

- N_VMake_Pthreads

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

  ```
  N_Vector N_VMake_Pthreads(long int vec_length, realtype *v_data, int num_threads);
  ```

- N_VCloneVectorArray_Pthreads

  This function creates (by cloning) an array of count Pthreads vectors.

  ```
  N_Vector *N_VCloneVectorArray_Pthreads(int count, N_Vector w);
  ```

- N_VCloneEmptyVectorArray_Pthreads

  This function creates (by cloning) an array of count Pthreads vectors, each with an empty (NULL) data array.

  ```
  N_Vector *N_VCloneEmptyVectorArray_Pthreads(int count, N_Vector w);
  ```

- N_VDestroyVectorArray_Pthreads

  This function frees memory allocated for the array of count variables of type N_Vector created with N_VCloneVectorArray_Pthreads or with N_VCloneEmptyVectorArray_Pthreads.

  ```
  void N_VDestroyVectorArray_Pthreads(N_Vector *vs, int count);
  ```

- N_VPrint_Pthreads

  This function prints the content of a Pthreads vector to stdout.

  ```
  void N_VPrint_Pthreads(N_Vector v);
  ```

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the component array via v_data = NV_DATA_PT(v) and then access v_data[i] within the loop than it is to use NV_Ith_PT(v,i) within the loop.

- N_VNewEmpty_Pthreads, N_VMake_Pthreads, and N_VCloneEmptyVectorArray_Pthreads set the field *own_data* = FALSE. N_VDestroy_Pthreads and N_VDestroyVectorArray_Pthreads will not attempt to free the pointer *data* for any N_Vector with *own_data* set to FALSE. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- To maximize efficiency, vector operations in the NVECTOR_PTHREADS implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

6.5 NVECTOR Examples

There are NVector examples that may be installed for each implementation: serial, parallel, OpenMP, and Pthreads. Each implementation makes use of the functions in test_nvector.c. These example functions show simple usage of the NVector family of functions. The input to the examples are the vector length, number of threads (if threaded implementation), and a print timing flag.
The following is a list of the example functions in test_nvector.c:

- Test_N_VClone: Creates clone of vector and checks validity of clone.
- Test_N_VCloneEmpty: Creates clone of empty vector and checks validity of clone.

- `Test_N_VCloneVectorArray`: Creates clone of vector array and checks validity of cloned array.
- `Test_N_VCloneVectorArray`: Creates clone of empty vector array and checks validity of cloned array.
- `Test_N_VGetArrayPointer`: Get array pointer.
- `Test_N_VSetArrayPointer`: Allocate new vector, set pointer to new vector array, and check values.
- `Test_N_VLinearSum` Case 1a: Test y = x + y
- `Test_N_VLinearSum` Case 1b: Test y = -x + y
- `Test_N_VLinearSum` Case 1c: Test y = ax + y
- `Test_N_VLinearSum` Case 2a: Test x = x + y
- `Test_N_VLinearSum` Case 2b: Test x = x - y
- `Test_N_VLinearSum` Case 2c: Test x = x + by
- `Test_N_VLinearSum` Case 3: Test z = x + y
- `Test_N_VLinearSum` Case 4a: Test z = x - y
- `Test_N_VLinearSum` Case 4b: Test z = -x + y
- `Test_N_VLinearSum` Case 5a: Test z = x + by
- `Test_N_VLinearSum` Case 5b: Test z = ax + y
- `Test_N_VLinearSum` Case 6a: Test z = -x + by
- `Test_N_VLinearSum` Case 6b: Test z = ax - y
- `Test_N_VLinearSum` Case 7: Test z = a(x + y)
- `Test_N_VLinearSum` Case 8: Test z = a(x - y)
- `Test_N_VLinearSum` Case 9: Test z = ax + by
- `Test_N_VConst`: Fill vector with constant and check result.
- `Test_N_VProd`: Test vector multiply: z = x * y
- `Test_N_VDiv`: Test vector division: z = x / y
- `Test_N_VScale`: Case 1: scale: x = cx
- `Test_N_VScale`: Case 2: copy: z = x
- `Test_N_VScale`: Case 3: negate: z = -x
- `Test_N_VScale`: Case 4: combination: z = cx
- `Test_N_VAbs`: Create absolute value of vector.
- `Test_N_VAddConst`: add constant vector: z = c + x
- `Test_N_VDotProd`: Calculate dot product of two vectors.
- `Test_N_VMaxNorm`: Create vector with known values, find and validate max norm.
- `Test_N_VWrmsNorm`: Create vector of known values, find and validate weighted root mean square.

- `Test_N_VWrmsNormMask`: Case 1: Create vector of known values, find and validate weighted root mean square using all elements.
- `Test_N_VWrmsNormMask`: Case 2: Create vector of known values, find and validate weighted root mean square using no elements.
- `Test_N_VMin`: Create vector, find and validate the min.
- `Test_N_VWL2Norm`: Create vector, find and validate the weighted Euclidean L2 norm.
- `Test_N_VL1Norm`: Create vector, find and validate the L1 norm.
- `Test_N_VCompare`: Compare vector with constant returning and validating comparison vector.
- `Test_N_VInvTest`: Test z[i] = 1 / x[i]
- `Test_N_VConstrMask`: Test mask of vector x with vector c.
- `Test_N_VMinQuotient`: Fill two vectors with known values. Calculate and validate minimum quotient.

6.6 NVECTOR functions used by IDA

In Table 6.2 below, we list the vector functions used in the NVECTOR module used by the IDA package. The table also shows, for each function, which of the code modules uses the function. The IDA column shows function usage within the main integrator module, while the remaining five columns show function usage within each of the seven IDA linear solvers, the IDABBDPRE preconditioner module, and the FIDA module. Here IDADLS stands for IDADENSE and IDABAND; IDASPILS stands for IDASPGMR, IDASPBCG, and IDASPTFQMR; and IDASLS stands for IDAKLU and IDASUPERLUMT.

There is one subtlety in the IDASPILS column hidden by the table, explained here for the case of the IDASPGMR module. The `N_VDotProd` function is called both within the interface file `ida_spgmr.c` and within the implementation files `sundials_spgmr.c` and `sundials_iterative.c` for the generic SPGMR solver upon which the IDASPGMR solver is built. Also, although `N_VDiv` and `N_VProd` are not called within the interface file `ida_spgmr.c`, they are called within the implementation file `sundials_spgmr.c`, and so are required by the IDASPGMR solver module. Analogous statements apply to the IDASPBCG and IDASPTFQMR modules, except that they do not use `sundials_iterative.c`. This issue does not arise for the direct IDA linear solvers because the generic DENSE and BAND solvers (used in the implementation of IDADENSE and IDABAND) do not make calls to any vector functions.

Of the functions listed in Table 6.1, `N_VWL2Norm`, `N_VL1Norm`, and `N_VInvTest` are *not* used by IDA. Therefore a user-supplied NVECTOR module for IDA could omit these three functions.

Table 6.2: List of vector functions usage by IDA code modules

	IDA	IDADLS	IDASPILS	IDASLS	IDABBDPRE	FIDA
N_VClone	✓		✓		✓	✓
N_VCloneEmpty						✓
N_VDestroy	✓		✓		✓	✓
N_VSpace	✓					
N_VGetArrayPointer		✓		✓	✓	✓
N_VSetArrayPointer		✓				✓
N_VLinearSum	✓	✓	✓			
N_VConst	✓		✓			
N_VProd	✓		✓			
N_VDiv	✓		✓			
N_VScale	✓	✓	✓	✓	✓	
N_VAbs	✓					
N_VInv	✓					
N_VAddConst	✓					
N_VDotProd			✓			
N_VMaxNorm	✓					
N_VWrmsNorm	✓		✓			
N_VMin	✓					
N_VMinQuotient	✓					
N_VConstrMask	✓					
N_VWrmsNormMask	✓					
N_VCompare	✓					

Chapter 7

Providing Alternate Linear Solver Modules

The central IDA module interfaces with a linear solver module by way of calls to five functions. These are denoted here by `linit`, `lsetup`, `lsolve`, `lperf`, and `lfree`. Briefly, their purposes are as follows:

- `linit`: initialize memory specific to the linear solver;
- `lsetup`: evaluate and preprocess the Jacobian or preconditioner;
- `lsolve`: solve the linear system;
- `lperf`: monitor performance and issue warnings;
- `lfree`: free the linear solver memory.

A linear solver module must also provide a user-callable **specification function** (like those described in §4.5.3) which will attach the above five functions to the main IDA memory block. The IDA memory block is a structure defined in the header file `ida_impl.h`. A pointer to such a structure is defined as the type `IDAMem`. The five fields in an `IDAMem` structure that must point to the linear solver's functions are `ida_linit`, `ida_lsetup`, `ida_lsolve`, `ida_lperf`, and `ida_lfree`, respectively. Note that of the five interface functions, only `lsolve` is required. The `lfree` function must be provided only if the solver specification function makes any memory allocation. For any of the functions that are *not* provided, the corresponding field should be set to `NULL`. The linear solver specification function must also set the value of the field `ida_setupNonNull` in the IDA memory block — to `TRUE` if `lsetup` is used, or `FALSE` otherwise.

Typically, the linear solver will require a block of memory specific to the solver, and a principal function of the specification function is to allocate that memory block, and initialize it. Then the field `ida_lmem` in the IDA memory block is available to attach a pointer to that linear solver memory. This block can then be used to facilitate the exchange of data between the five interface functions.

If the linear solver involves adjustable parameters, the specification function should set the default values of those. User-callable functions may be defined that could, optionally, override the default parameter values.

We encourage the use of performance counters in connection with the various operations involved with the linear solver. Such counters would be members of the linear solver memory block, would be initialized in the `linit` function, and would be incremented by the `lsetup` and `lsolve` functions. Then, user-callable functions would be needed to obtain the values of these counters.

For consistency with the existing IDA linear solver modules, we recommend that the return value of the specification function be 0 for a successful return, and a negative value if an error occurs. Possible error conditions include: the pointer to the main IDA memory block is `NULL`, an input is illegal, the NVECTOR implementation is not compatible, or a memory allocation fails.

These five functions, which interface between IDA and the linear solver module, necessarily have fixed call sequences. Thus a user wishing to implement another linear solver within the IDA package must adhere to this set of interfaces. The following is a complete description of the call list for each of these functions. Note that the call list of each function includes a pointer to the main IDA memory block, by which the function can access various data related to the IDA solution. The contents of this memory block are given in the file `ida_impl.h` (but not reproduced here, for the sake of space).

7.1 Initialization function

The type definition of `linit` is

`linit`

Definition `int (*linit)(IDAMem IDA_mem);`

Purpose The purpose of `linit` is to complete initializations for the specific linear solver, such as counters and statistics. It should also set pointers to data blocks that will later be passed to functions associated with the linear solver. The `linit` function is called once only, at the start of the problem, during the first call to `IDASolve`.

Arguments `IDA_mem` is the IDA memory pointer of type `IDAMem`.

Return value An `linit` function should return 0 if it has successfully initialized the IDA linear solver and a negative value otherwise.

7.2 Setup function

The type definition of `lsetup` is

`lsetup`

Definition
```
int (*lsetup)(IDAMem IDA_mem, N_Vector yyp, N_Vector ypp, N_Vector resp,
              N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);
```

Purpose The job of `lsetup` is to prepare the linear solver for subsequent calls to `lsolve`, in the solution of systems $Mx = b$, where M is some approximation to the system Jacobian, $J = \partial F/\partial y + cj\ \partial F/\partial \dot{y}$. (See Eqn. (2.5), in which $\alpha = cj$). Here cj is available as `IDA_mem->ida_cj`.

The `lsetup` function may call a user-supplied function, or a function within the linear solver module, to compute Jacobian-related data that is required by the linear solver. It may also preprocess that data as needed for `lsolve`, which may involve calling a generic function (such as for LU factorization). This data may be intended either for direct use (in a direct linear solver) or for use in a preconditioner (in a preconditioned iterative linear solver).

The `lsetup` function is not called at every time step, but only as frequently as the solver determines that it is appropriate to perform the setup task. In this way, Jacobian-related data generated by `lsetup` is expected to be used over a number of time steps.

Arguments `IDA_mem` is the IDA memory pointer of type `IDAMem`.

`yyp` is the predicted y vector for the current IDA internal step.

`ypp` is the predicted $\dot{y}$ vector for the current IDA internal step.

`resp` is the value of the residual function at `yyp` and `ypp`, i.e. $F(t_n, y_{pred}, \dot{y}_{pred})$.

`vtemp1`

`vtemp2`

`vtemp3` are temporary variables of type `N_Vector` provided for use by `lsetup`.

Return value The `lsetup` function should return 0 if successful, a positive value for a recoverable error, and a negative value for an unrecoverable error. On a recoverable error return, the solver will attempt to recover by reducing the step size.

7.3 Solve function

The type definition of `lsolve` is

`lsolve`

Definition
```
int (*lsolve)(IDAMem IDA_mem, N_Vector b, N_Vector weight,
              N_Vector ycur, N_Vector ypcur, N_Vector rescur);
```

Purpose The `lsolve` function must solve the linear system, $Mx = b$, where M is some approximation to the system Jacobian, $J = \partial F/\partial y + cj\, \partial F/\partial \dot{y}$ (see Eqn. (2.5), in which $\alpha = cj$), and the right-hand side vector, b, is input. Here cj is available as `IDA_mem->ida_cj`.

`lsolve` is called once per Newton iteration, hence possibly several times per time step.

If there is an `lsetup` function, this `lsolve` function should make use of any Jacobian data that was computed and preprocessed by `lsetup`, either for direct use, or for use in a preconditioner.

Arguments
- `IDA_mem` is the IDA memory pointer of type `IDAMem`.
- `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 (2.6). This weight vector is included here to enable the computation of weighted norms needed to test for the convergence of iterative methods (if any) within the linear solver.
- `ycur` is a vector that contains the solver's current approximation to $y(t_n)$.
- `ypcur` is a vector that contains the solver's current approximation to $\dot{y}(t_n)$.
- `rescur` is a vector that contains the current residual, $F(t_n$, `ycur`, `ypcur`$)$.

Return value The `lsolve` function should return a positive value for a recoverable error and a negative value for an unrecoverable error. Success is indicated by a 0 return value. On a recoverable error return, the solver will attempt to recover, such as by calling the `lsetup` function with current arguments.

7.4 Performance monitoring function

The type definition of `lperf` is

`lperf`

Definition
```
int (*lperf)(IDAMem IDA_mem, int perftask);
```

Purpose The `lperf` function is to monitor the performance of the linear solver. It can be used to compute performance metrics related to the linear solver and issue warnings if these indicate poor performance of the linear solver. The `lperf` function is called with `perftask` = 0 at the start of each call to `IDASolve`, and then is called with `perftask` = 1 just before each internal time step.

Arguments
- `IDA_mem` is the IDA memory pointer of type `IDAMem`.
- `perftask` is a task flag. `perftask` = 0 means initialize needed counters. `perftask` = 1 means evaluate performance and issue warnings if needed. Counters that are used to compute performance metrics (e.g. counts of iterations within the `lsolve` function) should be initialized here when `perftask` = 0, and used for the calculation of metrics when `perftask` = 1.

Return value The `lperf` return value is ignored.

7.5 Memory deallocation function

The type definition of `lfree` is

`lfree`

Definition `void (*lfree)(IDAMem IDA_mem);`

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

Arguments The argument `IDA_mem` is the IDA memory pointer of type `IDAMem`.

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

Notes This function is called once a problem has been completed and the linear solver is no longer needed.

Chapter 8

General Use Linear Solver Components in SUNDIALS

In this chapter, we describe linear solver code components that are included in SUNDIALS, but which are of potential use as generic packages in themselves, either in conjunction with the use of SUNDIALS or separately.

These generic modules in SUNDIALS are organized in three families, the *dls* family, which includes direct linear solvers appropriate for sequential computations; the *sls* family, which includes sparse matrix solvers; and the *spils* family, which includes scaled preconditioned iterative (Krylov) linear solvers. The solvers in each family share common data structures and functions.

The *dls* family contains the following two generic linear solvers:

- The DENSE package, a linear solver for dense matrices either specified through a matrix type (defined below) or as simple arrays.
- The BAND package, a linear solver for banded matrices either specified through a matrix type (defined below) or as simple arrays.

Note that this family also includes the Blas/Lapack linear solvers (dense and band) available to the SUNDIALS solvers, but these are not discussed here.

The *sls* family contains a sparse matrix package and interfaces between it and two sparse direct solver packages:

- The KLU package, a linear solver for compressed-sparse-column matrices, [1, 12].
- The SUPERLUMT package, a threaded linear solver for compressed-sparse-column matrices, [2, 20, 13].

The *spils* family contains the following generic linear solvers:

- The SPGMR package, a solver for the scaled preconditioned GMRES method.
- The SPFGMR package, a solver for the scaled preconditioned Flexible GMRES method.
- The SPBCG package, a solver for the scaled preconditioned Bi-CGStab method.
- The SPTFQMR package, a solver for the scaled preconditioned TFQMR method.

For reasons related to installation, the names of the files involved in these packages begin with the prefix `sundials_`. But despite this, each of the *dls* and *spils* solvers is in fact generic, in that it is usable completely independently of SUNDIALS.

For the sake of space, the functions for the `dense` and `band` modules that work with a matrix type, and the functions in the SPGMR, SPFGMR, SPBCG, and SPTFQMR modules are only summarized briefly, since they are less likely to be of direct use in connection with a SUNDIALS solver. However, the

functions for dense matrices treated as simple arrays and sparse matrices are fully described, because we expect that they will be useful in the implementation of preconditioners used with the combination of one of the SUNDIALS solvers and one of the *spils* linear solvers.

8.1 The DLS modules: DENSE and BAND

The files comprising the DENSE generic linear solver, and their locations in the SUNDIALS *srcdir*, are as follows:

- header files (located in *srcdir*/include/sundials)
  sundials_direct.h, sundials_dense.h,
  sundials_types.h, sundials_math.h, sundials_config.h
- source files (located in *srcdir*/src/sundials)
  sundials_direct.c, sundials_dense.c, sundials_math.c

The files comprising the BAND generic linear solver are as follows:

- header files (located in *srcdir*/include/sundials)
  sundials_direct.h, sundials_band.h,
  sundials_types.h, sundials_math.h, sundials_config.h
- source files (located in *srcdir*/src/sundials)
  sundials_direct.c, sundials_band.c, sundials_math.c

Only two of the preprocessing directives in the header file sundials_config.h are relevant to the DENSE and BAND packages by themselves.

- (required) definition of the precision of the SUNDIALS type realtype. One of the following lines must be present:
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
- (optional) use of generic math functions: #define SUNDIALS_USE_GENERIC_MATH 1

The sundials_types.h header file defines the SUNDIALS realtype and booleantype types and the macro RCONST, while the sundials_math.h header file is needed for the macros SUNMIN and SUNMAX, and the function SUNRabs.

The files listed above for either module can be extracted from the SUNDIALS *srcdir* and compiled by themselves into a separate library or into a larger user code.

8.1.1 Type DlsMat

The type DlsMat, defined in sundials_direct.h is a pointer to a structure defining a generic matrix, and is used with all linear solvers in the *dls* family:

```
typedef struct _DlsMat {
  int type;
  long int M;
  long int N;
  long int ldim;
  long int mu;
  long int ml;
  long int s_mu;
  realtype *data;
  long int ldata;
  realtype **cols;
} *DlsMat;
```

For the DENSE module, the relevant fields of this structure are as follows. Note that a dense matrix of type `DlsMat` need not be square.

type - `SUNDIALS_DENSE` (=1)

M - number of rows

N - number of columns

ldim - leading dimension (`ldim` $\geq$ `M`)

data - pointer to a contiguous block of `realtype` variables

ldata - length of the data array (= `ldim`·`N`). The (`i`,`j`)-th element of a dense matrix `A` of type `DlsMat` (with 0 $\leq$ `i` $<$ `M` and 0 $\leq$ `j` $<$ `N`) is given by the expression `(A->data)[0][j*M+i]`

cols - array of pointers. `cols[j]` points to the first element of the j-th column of the matrix in the array data. The (`i`,`j`)-th element of a dense matrix `A` of type `DlsMat` (with 0 $\leq$ `i` $<$ `M` and 0 $\leq$ `j` $<$ `N`) is given by the expression `(A->cols)[j][i]`

For the BAND module, the relevant fields of this structure are as follows (see Figure 8.1 for a diagram of the underlying data representation in a banded matrix of type `DlsMat`). Note that only square band matrices are allowed.

type - `SUNDIALS_BAND` (=2)

M - number of rows

N - number of columns (`N` = `M`)

mu - upper half-bandwidth, 0 $\leq$ `mu` $<$ min(`M`,`N`)

ml - lower half-bandwidth, 0 $\leq$ `ml` $<$ min(`M`,`N`)

s_mu - storage upper bandwidth, `mu` $\leq$ `s_mu` $<$ `N`. The LU decomposition routine writes the LU factors into the storage for A. The upper triangular factor U, however, may have an upper bandwidth as big as min(`N`-1,`mu`+`ml`) because of partial pivoting. The `s_mu` field holds the upper half-bandwidth allocated for A.

ldim - leading dimension (`ldim` $\geq$ `s_mu`)

data - pointer to a contiguous block of `realtype` variables. The elements of a banded matrix of type `DlsMat` are stored columnwise (i.e. columns are stored one on top of the other in memory). Only elements within the specified half-bandwidths are stored. `data` is a pointer to `ldata` contiguous locations which hold the elements within the band of A.

ldata - length of the data array (= `ldim`·(`s_mu`+`ml`+1)

cols - array of pointers. `cols[j]` is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from `s_mu`−`mu` (to access the uppermost element within the band in the j-th column) to `s_mu`+`ml` (to access the lowest element within the band in the j-th column). Indices from 0 to `s_mu`−`mu`−1 give access to extra storage elements required by the LU decomposition function. Finally, `cols[j][i-j+s_mu]` is the (i, j)-th element, j−`mu` $\leq i \leq j$+`ml`.

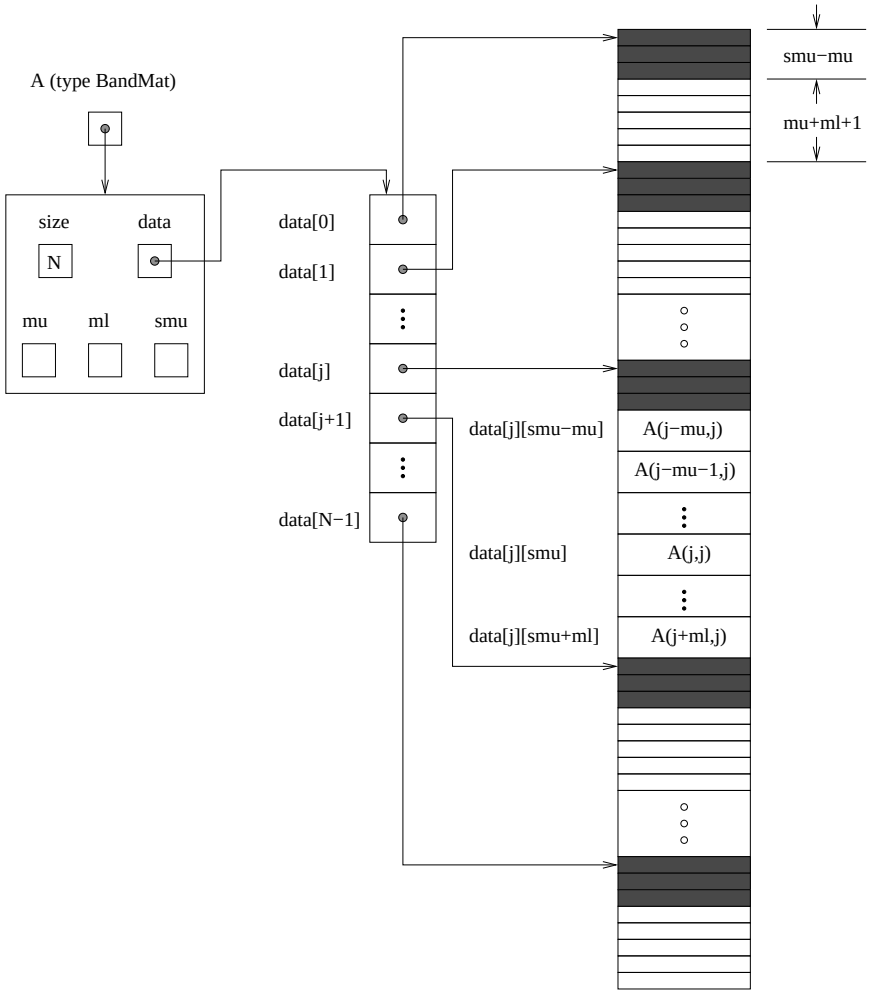


Figure 8.1: Diagram of the storage for a banded matrix of type `DlsMat`. Here `A` is an $N \times N$ band matrix of type `DlsMat` with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to $N-1$ and the (i,j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the `BandGBTRF` and `BandGBTRS` routines.

8.1.2 Accessor macros for the DLS modules

The macros below allow a user to efficiently access individual matrix elements without writing out explicit data structure references and without knowing too much about the underlying element storage. The only storage assumption needed is that elements are stored columnwise and that a pointer to the j-th column of elements can be obtained via the DENSE_COL or BAND_COL macros. Users should use these macros whenever possible.

The following two macros are defined by the DENSE module to provide access to data in the DlsMat type:

- DENSE_ELEM

  Usage : DENSE_ELEM(A,i,j) = a_ij; or a_ij = DENSE_ELEM(A,i,j);

  DENSE_ELEM references the (i,j)-th element of the $M \times N$ DlsMat A, $0 \le$ i $< M$, $0 \le$ j $< N$.

- DENSE_COL

  Usage : col_j = DENSE_COL(A,j);

  DENSE_COL references the j-th column of the $M \times N$ DlsMat A, $0 \le$ j $< N$. The type of the expression DENSE_COL(A,j) is realtype * . After the assignment in the usage above, col_j may be treated as an array indexed from 0 to $M - 1$. The (i, j)-th element of A is referenced by col_j[i].

The following three macros are defined by the BAND module to provide access to data in the DlsMat type:

- BAND_ELEM

  Usage : BAND_ELEM(A,i,j) = a_ij; or a_ij = BAND_ELEM(A,i,j);

  BAND_ELEM references the (i,j)-th element of the $N \times N$ band matrix A, where $0 \le$ i, j $\le N-1$. The location (i,j) should further satisfy j−(A->mu) $\le$ i $\le$ j+(A->ml).

- BAND_COL

  Usage : col_j = BAND_COL(A,j);

  BAND_COL references the diagonal element of the j-th column of the $N \times N$ band matrix A, $0 \le$ j $\le N-1$. The type of the expression BAND_COL(A,j) is realtype *. The pointer returned by the call BAND_COL(A,j) can be treated as an array which is indexed from −(A->mu) to (A->ml).

- BAND_COL_ELEM

  Usage : BAND_COL_ELEM(col_j,i,j) = a_ij; or a_ij = BAND_COL_ELEM(col_j,i,j);

  This macro references the (i,j)-th entry of the band matrix A when used in conjunction with BAND_COL to reference the j-th column through col_j. The index (i,j) should satisfy j−(A->mu) $\le$ i $\le$ j+(A->ml).

8.1.3 Functions in the DENSE module

The DENSE module defines two sets of functions with corresponding names. The first set contains functions (with names starting with a capital letter) that act on dense matrices of type DlsMat. The second set contains functions (with names starting with a lower case letter) that act on matrices represented as simple arrays.

The following functions for DlsMat dense matrices are available in the DENSE package. For full details, see the header files sundials_direct.h and sundials_dense.h.

- NewDenseMat: allocation of a DlsMat dense matrix;

- DestroyMat: free memory for a DlsMat matrix;

- `PrintMat`: print a `DlsMat` matrix to standard output.
- `NewLintArray`: allocation of an array of `long int` integers for use as pivots with `DenseGETRF` and `DenseGETRS`;
- `NewIntArray`: allocation of an array of `int` integers for use as pivots with the Lapack dense solvers;
- `NewRealArray`: allocation of an array of `realtype` for use as right-hand side with `DenseGETRS`;
- `DestroyArray`: free memory for an array;
- `SetToZero`: load a matrix with zeros;
- `AddIdentity`: increment a square matrix by the identity matrix;
- `DenseCopy`: copy one matrix to another;
- `DenseScale`: scale a matrix by a scalar;
- `DenseGETRF`: LU factorization with partial pivoting;
- `DenseGETRS`: solution of $Ax = b$ using LU factorization (for square matrices A);
- `DensePOTRF`: Cholesky factorization of a real symmetric positive matrix;
- `DensePOTRS`: solution of $Ax = b$ using the Cholesky factorization of A;
- `DenseGEQRF`: QR factorization of an $m \times n$ matrix, with $m \geq n$;
- `DenseORMQR`: compute the product $w = Qv$, with Q calculated using `DenseGEQRF`;
- `DenseMatvec`: compute the product $y = Ax$, for an M by N matrix A;

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

- `newDenseMat`

  `newDenseMat(m,n)` allocates storage for an `m` by `n` dense matrix. It returns a pointer to the newly allocated storage if successful. If the memory request cannot be satisfied, then `newDenseMat` returns `NULL`. The underlying type of the dense matrix returned is `realtype**`. If we allocate a dense matrix `realtype** a` by `a = newDenseMat(m,n)`, then `a[j][i]` references the (`i`,`j`)-th element of the matrix `a`, $0 \leq$ `i` $<$ `m`, $0 \leq$ `j` $< n$, and `a[j]` is a pointer to the first element in the `j`-th column of `a`. The location `a[0]` contains a pointer to `m` $\times$ `n` contiguous locations which contain the elements of `a`.

- `destroyMat`

  `destroyMat(a)` frees the dense matrix `a` allocated by `newDenseMat`;

- `newLintArray`

  `newLintArray(n)` allocates an array of `n` integers, all `long int`. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `newIntArray`

  `newIntArray(n)` allocates an array of `n` integers, all `int`. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `newRealArray`

  `newRealArray(n)` allocates an array of `n realtype` values. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `destroyArray`

  `destroyArray(p)` frees the array `p` allocated by `newLintArray`, `newIntArray`, or `newRealArray`;

- `denseCopy`

  `denseCopy(a,b,m,n)` copies the `m` by `n` dense matrix `a` into the `m` by `n` dense matrix `b`;

- `denseScale`

  `denseScale(c,a,m,n)` scales every element in the `m` by `n` dense matrix `a` by the scalar `c`;

- `denseAddIdentity`

  `denseAddIdentity(a,n)` increments the *square* `n` by `n` dense matrix `a` by the identity matrix I_n;

- `denseGETRF`

  `denseGETRF(a,m,n,p)` factors the `m` by `n` dense matrix `a`, using Gaussian elimination with row pivoting. It overwrites the elements of `a` with its LU factors and keeps track of the pivot rows chosen in the pivot array `p`.

  A successful LU factorization leaves the matrix `a` and the pivot array `p` with the following information:

  1. `p[k]` contains the row number of the pivot element chosen at the beginning of elimination step `k`, `k` $= 0, 1, ...,$`n`-1.
  2. If the unique LU factorization of `a` is given by $Pa = LU$, where P is a permutation matrix, L is an `m` by `n` lower trapezoidal matrix with all diagonal elements equal to 1, and U is an `n` by `n` upper triangular matrix, then the upper triangular part of `a` (including its diagonal) contains U and the strictly lower trapezoidal part of `a` contains the multipliers, $I - L$. If `a` is square, L is a unit lower triangular matrix.

     `denseGETRF` returns 0 if successful. Otherwise it encountered a zero diagonal element during the factorization, indicating that the matrix `a` does not have full column rank. In this case it returns the column index (numbered from one) at which it encountered the zero.

- `denseGETRS`

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

- `densePOTRF`

  `densePOTRF(a,m)` calculates the Cholesky decomposition of the `m` by `m` dense matrix `a`, assumed to be symmetric positive definite. Only the lower triangle of `a` is accessed and overwritten with the Cholesky factor.

- `densePOTRS`

  `densePOTRS(a,m,b)` solves the `m` by `m` linear system $ax = b$. It assumes that the Cholesky factorization of `a` has been calculated in the lower triangular part of `a` by a successful call to `densePOTRF(a,m)`.

- `denseGEQRF`

  `denseGEQRF(a,m,n,beta,wrk)` calculates the QR decomposition of the `m` by `n` matrix `a` (`m` $\geq$ `n`) using Householder reflections. On exit, the elements on and above the diagonal of `a` contain the `n` by `n` upper triangular matrix `R`; the elements below the diagonal, with the array `beta`, represent the orthogonal matrix `Q` as a product of elementary reflectors. The real array `wrk`, of length `m`, must be provided as temporary workspace.

- `denseORMQR`

  `denseORMQR(a,m,n,beta,v,w,wrk)` calculates the product $w = Qv$ for a given vector `v` of length `n`, where the orthogonal matrix Q is encoded in the `m` by `n` matrix `a` and the vector `beta` of length `n`, after a successful call to `denseGEQRF(a,m,n,beta,wrk)`. The real array `wrk`, of length `m`, must be provided as temporary workspace.

- `denseMatvec`

  `denseMatvec(a,x,y,m,n)` calculates the product $y = ax$ for a given vector `x` of length `n`, and `m` by `n` matrix `a`.

8.1.4 Functions in the BAND module

The BAND module defines two sets of functions with corresponding names. The first set contains functions (with names starting with a capital letter) that act on band matrices of type `DlsMat`. The second set contains functions (with names starting with a lower case letter) that act on matrices represented as simple arrays.

The following functions for `DlsMat` banded matrices are available in the BAND package. For full details, see the header files `sundials_direct.h` and `sundials_band.h`.

- `NewBandMat`: allocation of a `DlsMat` band matrix;
- `DestroyMat`: free memory for a `DlsMat` matrix;
- `PrintMat`: print a `DlsMat` matrix to standard output.
- `NewLintArray`: allocation of an array of `int` integers for use as pivots with `BandGBRF` and `BandGBRS`;
- `NewIntArray`: allocation of an array of `int` integers for use as pivots with the Lapack band solvers;
- `NewRealArray`: allocation of an array of `realtype` for use as right-hand side with `BandGBRS`;
- `DestroyArray`: free memory for an array;
- `SetToZero`: load a matrix with zeros;
- `AddIdentity`: increment a square matrix by the identity matrix;
- `BandCopy`: copy one matrix to another;
- `BandScale`: scale a matrix by a scalar;
- `BandGBTRF`: LU factorization with partial pivoting;
- `BandGBTRS`: solution of $Ax = b$ using LU factorization;
- `BandMatvec`: compute the product $y = Ax$, for a square band matrix A;

The following functions for small band matrices are available in the BAND package:

- `newBandMat`

  `newBandMat(n, smu, ml)` allocates storage for an `n` by `n` band matrix with lower half-bandwidth `ml`.

- `destroyMat`

  `destroyMat(a)` frees the band matrix `a` allocated by `newBandMat`;

- newLintArray

  newLintArray(n) allocates an array of n integers, all long int. It returns a pointer to the first element in the array if successful. It returns NULL if the memory request could not be satisfied.

- newIntArray

  newIntArray(n) allocates an array of n integers, all int. It returns a pointer to the first element in the array if successful. It returns NULL if the memory request could not be satisfied.

- newRealArray

  newRealArray(n) allocates an array of n realtype values. It returns a pointer to the first element in the array if successful. It returns NULL if the memory request could not be satisfied.

- destroyArray

  destroyArray(p) frees the array p allocated by newLintArray, newIntArray, or newRealArray;

- bandCopy

  bandCopy(a,b,n,a_smu, b_smu,copymu, copyml) copies the n by n band matrix a into the n by n band matrix b;

- bandScale

  bandScale(c,a,n,mu,ml,smu) scales every element in the n by n band matrix a by c;

- bandAddIdentity

  bandAddIdentity(a,n,smu) increments the n by n band matrix a by the identity matrix;

- bandGETRF

  bandGETRF(a,n,mu,ml,smu,p) factors the n by n band matrix a, using Gaussian elimination with row pivoting. It overwrites the elements of a with its LU factors and keeps track of the pivot rows chosen in the pivot array p.

- bandGETRS

  bandGETRS(a,n,smu,ml,p,b) solves the n by n linear system $ax = b$. It assumes that a (of size n × n) has been LU-factored and the pivot array p has been set by a successful call to bandGETRF(a,n,mu,ml,smu,p). The solution x is written into the b array.

- bandMatvec

  bandMatvec(a,x,y,n,mu,ml,smu) calculates the product $y = ax$ for a given vector x of length n, and n by n band matrix a.

8.2 The SLS module

SUNDIALS provides a compressed-sparse-column matrix type and sparse matrix support functions. In addition, SUNDIALS provides interfaces to the publically available KLU and SuperLU_MT sparse direct solver packages. The files comprising the SLS matrix module, used in the KLU and SUPERLUMT linear solver packages, and their locations in the SUNDIALS *srcdir*, are as follows:

- header files (located in *srcdir*/include/sundials)
  sundials_sparse.h, sundials_klu_impl.h,
  sundials_superlumt_impl.h, sundials_types.h,
  sundials_math.h, sundials_config.h

- source files (located in *srcdir*/src/sundials)
  sundials_sparse.c, sundials_math.c

Only two of the preprocessing directives in the header file `sundials_config.h` are relevant to the SLS package by itself:

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  ```
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
  ```
- (optional) use of generic math functions: `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the macros `SUNMIN` and `SUNMAX`, and the function `SUNRabs`.

8.2.1 Type SlsMat

The type `SlsMat`, defined in `sundials_sparse.h` is a pointer to a structure defining a generic compressed-sparse-column matrix, and is used with all linear solvers in the *sls* family:

```
typedef struct _SlsMat {
  int M;
  int N;
  int NNZ;
  realtype *data;
  int *rowvals;
  int *colptrs;
} *SlsMat;
```

The fields of this structure are as follows (see Figure 8.2 for a diagram of the underlying compressed-sparse-column representation in a sparse matrix of type `SlsMat`). Note that a sparse matrix of type `SlsMat` need not be square.

M - number of rows

N - number of columns

NNZ - maximum number of nonzero entries in the matrix (allocated length of `data` and `rowvals` arrays)

data - pointer to a contiguous block of `realtype` variables (of length `NNZ`), containing the values of the nonzero entries in the matrix

rowvals - pointer to a contiguous block of `int` variables (of length `NNZ`), containing the row indices of each nonzero entry held in `data`

colptrs - pointer to a contiguous block of `int` variables (of length `N+1`). Each entry provides the index of the first column entry into the `data` and `rowvals` arrays, e.g. if `colptr[3]=7`, then the first nonzero entry in the fourth column of the matrix is located in `data[7]`, and is located in row `rowvals[7]` of the matrix. The last entry contains the total number of nonzero values in the matrix and hence points one past the end of the active data in the `data` and `rowvals` arrays.

For example, the 5×4 matrix

$$\begin{bmatrix} 0 & 3 & 1 & 0 \\ 3 & 0 & 0 & 2 \\ 0 & 7 & 0 & 0 \\ 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

could be stored in a `SlsMat` structure as either

```
M = 5;
N = 4;
NNZ = 8;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
rowvals = {1, 3, 0, 2, 0, 1, 3, 4};
colptrs = {0, 2, 4, 5, 8};
```

or

```
M = 5;
N = 4;
NNZ = 10;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *};
rowvals = {1, 3, 0, 2, 0, 1, 3, 4, *, *};
colptrs = {0, 2, 4, 5, 8};
```

where the first has no unused space, and the second has additional storage (the entries marked with * may contain any values). Note in both cases that the final value in `colptrs` is 8. The work associated with operations on the sparse matrix is proportional to this value and so one should use the best understanding of the number of nonzeroes here.

8.2.2 Functions in the SLS module

The SLS module defines functions that act on sparse matrices of type `SlsMat`. For full details, see the header file `sundials_sparse.h`.

- `NewSparseMat`

  `NewSparseMat(M, N, NNZ)` allocates storage for an `M` by `N` sparse matrix, with storage for up to `NNZ` nonzero entries.

- `SlsConvertDls`

  `SlsConvertDls(A)` converts a dense or band matrix `A` of type `DlsMat` into a new sparse matrix of type `SlsMat` by retaining only the nonzero values of the matrix `A`.

- `DestroySparseMat`

  `DestroySparseMat(A)` frees the memory for a sparse matrix `A` allocated by either `NewSparseMat` or `SlsConvertDls`.

- `SlsSetToZero(A)` zeros out the `SlsMat` matrix `A`. The storage for `A` is left unchanged.

- `CopySparseMat`

  `CopySparseMat(A, B)` copies the `SlsMat A` into the `SlsMat B`. It is assumed that the matrices have the same row/column dimensions. If `B` has insufficient storage to hold all the nonzero entries of `A`, the data and row index arrays in `B` are reallocated to match those in `A`.

- `ScaleSparseMat`

  `ScaleSparseMat(c, A)` scales every element in the `SlsMat A` by the `realtype` scalar `c`.

- `AddIdentitySparseMat`

  `AddIdentitySparseMat(A)` increments the `SlsMat A` by the identity matrix. If `A` is not square, only the existing diagonal values are incremented. Resizes the `data` and `rowvals` arrays of `A` to allow for new nonzero entries on the diagonal.

- `SlsAddMat`

  `SlsAddMat(A, B)` adds two `SlsMat` matrices `A` and `B`, placing the result back in `A`. Resizes the `data` and `rowvals` arrays of `A` upon completion to exactly match the nonzero storage for the result. Upon successful completion, the return value is zero; otherwise -1 is returned.

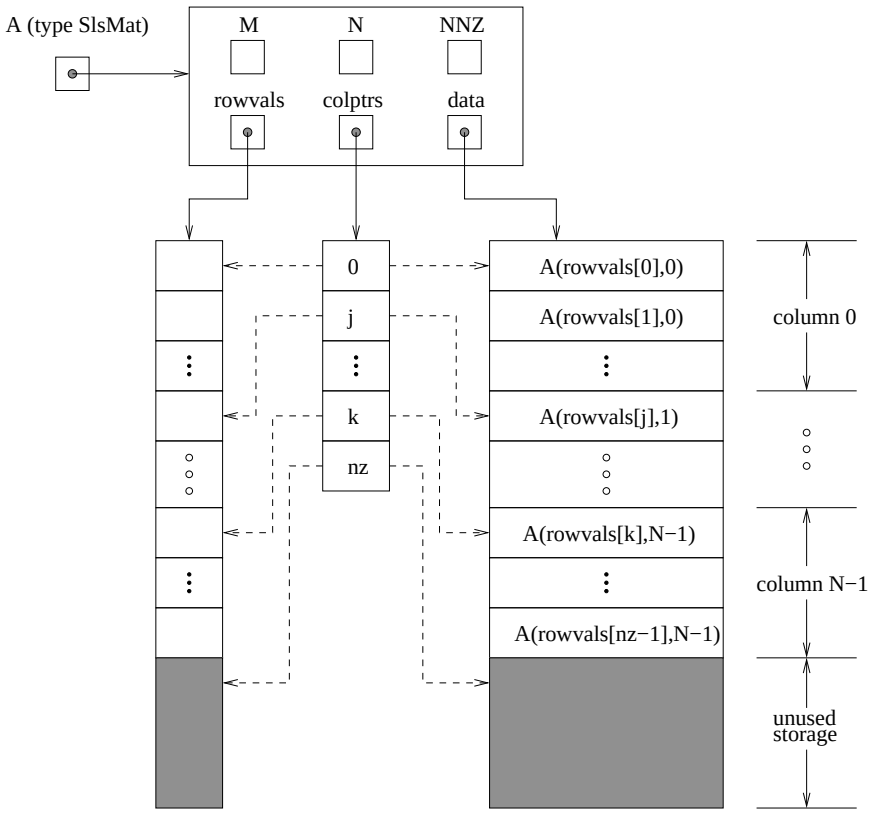


Figure 8.2: Diagram of the storage for a compressed-sparse-column matrix of type `SlsMat`. Here `A` is an $M \times N$ sparse matrix of type `SlsMat` with storage for up to `NNZ` nonzero entries (the allocated length of both `data` and `rowvals`). The entries in `rowvals` may assume values from 0 to $M-1$, corresponding to the row index (zero-based) of each nonzero value. The entries in `data` contain the values of the nonzero entries, with the row i, column j entry of `A` (again, zero-based) denoted as `A(i,j)`. The `colptrs` array contains $N+1$ entries; the first N denote the starting index of each column within the `rowvals` and `data` arrays, while the final entry points one past the final nonzero entry. Here, although `NNZ` values are allocated, only `nz` are actually filled in; the greyed-out portions of `data` and `rowvals` indicate extra allocated space.

- `ReallocSparseMat`

  `ReallocSparseMat(A)` eliminates unused storage in the `SlsMat A` by resizing the internal `data` and `rowvals` arrays to contain exactly `colptrs[N]` values.

- `SlsMatvec`

  `SlsMatvec(A, x, y)` computes the sparse matrix-vector product, $y = Ax$. If the `SlsMat A` is a sparse matrix of dimension $M \times N$, then it is assumed that `x` is a `realtype` array of length N, and `y` is a `realtype` array of length M. Upon successful completion, the return value is zero; otherwise -1 is returned.

- `PrintSparseMat`

  `PrintSparseMat(A)` Prints the `SlsMat` matrix `A` to standard output.

8.2.3 The KLU solver

KLU is a sparse matrix factorization and solver library written by Tim Davis [1, 12]. KLU has a symbolic factorization routine that computes the permutation of the linear system matrix to block triangular form and the permutations that will pre-order the diagonal blocks (the only ones that need to be factored) to reduce fill-in (using AMD, COLAMD, CHOLAMD, natural, or an ordering given by the user). Note that SUNDIALS uses the COLAMD ordering by default with KLU.

KLU breaks the factorization into two separate parts. The first is a symbolic factorization and the second is a numeric factorization that returns the factored matrix along with final pivot information. KLU also has a refactor routine that can be called instead of the numeric factorization. This routine will reuse the pivot information. This routine also returns diagnostic information that a user can examine to determine if numerical stability is being lost and a full numerical factorization should be done instead of the refactor.

The KLU interface in SUNDIALS will perform the symbolic factorization once. It then calls the numerical factorization once and will call the refactor routine until estimates of the numerical conditioning suggest a new factorization should be completed. The KLU interface also has a `ReInit` routine that can be used to force a full refactorization at the next solver setup call.

In order to use the SUNDIALS interface to KLU, it is assumed that KLU has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with KLU (see Appendix A for details).

Designed for serial calculations only, KLU is supported for calculations employing SUNDIALS' serial or shared-memory parallel NVECTOR modules (see Sections 6.1, 6.3 and 6.4 for details).

8.2.4 The SUPERLUMT solver

SUPERLUMT is a threaded sparse matrix factorization and solver library written by X. Sherry Li [2, 20, 13]. The package performs matrix factorization using threads to enhance efficiency in shared memory parallel environments. It should be noted that threads are only used in the factorization step.

In order to use the SUNDIALS interface to SUPERLUMT, it is assumed that SUPERLUMT has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with SUPERLUMT (see Appendix A for details).

Designed for serial and threaded calculations only, SUPERLUMT is supported for calculations employing SUNDIALS' serial or shared-memory parallel NVECTOR modules (see Sections 6.1, 6.3 and 6.4 for details).

8.3 The SPILS modules: SPGMR, SPFGMR, SPBCG, and SPTFQMR

The *spils* modules contain implementations of some of the most commonly use scaled preconditioned Krylov solvers. A linear solver module from the *spils* family can be used in conjunction with any

NVECTOR implementation library.

8.3.1 The SPGMR module

The SPGMR package, in the files `sundials_spgmr.h` and `sundials_spgmr.c`, includes an implementation of the scaled preconditioned GMRES method. A separate code module, implemented in `sundials_iterative.(h,c)`, contains auxiliary functions that support SPGMR, as well as the other Krylov solvers in SUNDIALS (SPFGMR, SPBCG, and SPTFQMR). For full details, including usage instructions, see the header files `sundials_spgmr.h` and `sundials_iterative.h`.

The files comprising the SPGMR generic linear solver, and their locations in the SUNDIALS *srcdir*, are as follows:

- header files (located in *srcdir*`/include/sundials`)
  `sundials_spgmr.h`, `sundials_iterative.h`, `sundials_nvector.h`,
  `sundials_types.h`, `sundials_math.h`, `sundials_config.h`
- source files (located in *srcdir*`/src/sundials`)
  `sundials_spgmr.c`, `sundials_iterative.c`, `sundials_nvector.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are required to use the SPGMR package by itself:

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  ```
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
  ```
- (optional) use of generic math functions:
  ```
  #define SUNDIALS_USE_GENERIC_MATH 1
  ```

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the macros `SUNMIN`, `SUNMAX`, and `SUNSQR`, and the functions `SUNRabs` and `SUNRsqrt`.

The generic NVECTOR files, `sundials_nvector.(h,c)` are needed for the definition of the generic `N_Vector` type and functions. The NVECTOR functions used by the SPGMR module are: `N_VDotProd`, `N_VLinearSum`, `N_VScale`, `N_VProd`, `N_VDiv`, `N_VConst`, `N_VClone`, `N_VCloneVectorArray`, `N_VDestroy`, and `N_VDestroyVectorArray`.

The nine files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into an SPGMR library or into a larger user code.

The following functions are available in the SPGMR package:

- `SpgmrMalloc`: allocation of memory for `SpgmrSolve`;
- `SpgmrSolve`: solution of $Ax = b$ by the SPGMR method;
- `SpgmrFree`: free memory allocated by `SpgmrMalloc`.

The following functions are available in the support package `sundials_iterative.(h,c)`:

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

8.3.2 The SPFGMR module

The SPFGMR package, in the files `sundials_spfgmr.h` and `sundials_spfgmr.c`, includes an implementation of the scaled preconditioned Flexible GMRES method. For full details, including usage instructions, see the file `sundials_spfgmr.h`.

The files needed to use the SPFGMR module by itself are the same as for the SPGMR module, but with `sundials_spfgmr.(h,c)` in place of `sundials_spgmr.(h,c)`.

The following functions are available in the SPFGMR package:

- `SpfgmrMalloc`: allocation of memory for `SpfgmrSolve`;
- `SpfgmrSolve`: solution of $Ax = b$ by the SPFGMR method;
- `SpfgmrFree`: free memory allocated by `SpfgmrMalloc`.

8.3.3 The SPBCG module

The SPBCG package, in the files `sundials_spbcgs.h` and `sundials_spbcgs.c`, includes an implementation of the scaled preconditioned Bi-CGStab method. For full details, including usage instructions, see the file `sundials_spbcgs.h`.

The files needed to use the SPBCG module by itself are the same as for the SPGMR module, but with `sundials_spbcgs.(h,c)` in place of `sundials_spgmr.(h,c)`.

The following functions are available in the SPBCG package:

- `SpbcgMalloc`: allocation of memory for `SpbcgSolve`;
- `SpbcgSolve`: solution of $Ax = b$ by the SPBCG method;
- `SpbcgFree`: free memory allocated by `SpbcgMalloc`.

8.3.4 The SPTFQMR module

The SPTFQMR package, in the files `sundials_sptfqmr.h` and `sundials_sptfqmr.c`, includes an implementation of the scaled preconditioned TFQMR method. For full details, including usage instructions, see the file `sundials_sptfqmr.h`.

The files needed to use the SPTFQMR module by itself are the same as for the SPGMR module, but with `sundials_sptfqmr.(h,c)` in place of `sundials_spgmr.(h,c)`.

The following functions are available in the SPTFQMR package:

- `SptfqmrMalloc`: allocation of memory for `SptfqmrSolve`;
- `SptfqmrSolve`: solution of $Ax = b$ by the SPTFQMR method;
- `SptfqmrFree`: free memory allocated by `SptfqmrMalloc`.

Appendix A

SUNDIALS Package Installation Procedure

The installation of any SUNDIALS package is accomplished by installing the SUNDIALS suite as a whole, according to the instructions that follow. The same procedure applies whether or not the downloaded file contains one or all solvers in SUNDIALS.

The SUNDIALS suite (or individual solvers) are distributed as compressed archives (`.tar.gz`). The name of the distribution archive is of the form *solver*`-x.y.z.tar.gz`, where *solver* is one of: `sundials`, `cvode`, `cvodes`, `arkode`, `ida`, `idas`, or `kinsol`, and `x.y.z` represents the version number (of the SUNDIALS suite or of the individual solver) . To begin the installation, first uncompress and expand the sources, by issuing

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

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

Starting with version `2.6.0` of SUNDIALS, CMake is the only supported method of installation. The explanations on the installation procedure begins with a few common observations:

- The remainder of this chapter will follow these conventions:

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

  builddir is the (temporary) directory under which SUNDIALS is built.

  instdir is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory *instdir*`/include` while libraries are installed under *instdir*`/lib`, with *instdir* specified at configuration time.

- For SUNDIALS CMake-based installation, in-source builds are prohibited; in other words, the build directory *builddir* can **not** be the same as *srcdir* and such an attempt will lead to an error. This prevents "polluting" the source tree and allows efficient builds for different configurations and/or options.

- The installation directory *instdir* can **not** be the same as the source directory *srcdir*.

- By default, only the libraries and header files are exported to the installation directory *instdir*. If enabled by the user (with the appropriate toggle for CMake), the examples distributed with SUNDIALS will be built together with the solver libraries but the installation step will result in exporting (by default in a subdirectory of the installation directory) the example sources and sample outputs together with automatically generated configuration files that reference the *installed* SUNDIALS headers and libraries. As such, these configuration files for the SUNDIALS examples can be used as "templates" for your own problems. CMake installs `CMakeLists.txt` files and also (as an option available only under Unix/Linux) `Makefile` files. Note this installation

approach also allows the option of building the SUNDIALS examples without having to install them. (This can be used as a sanity check for the freshly built libraries.)

- Even if generation of shared libraries is enabled, only static libraries are created for the FCMIX modules. (Because of the use of fixed names for the Fortran user-provided subroutines, FCMIX shared libraries would result in "undefined symbol" errors at link time.)

A.1 CMake-based installation

CMake-based installation provides a platform-independent build system. CMake can generate Unix and Linux Makefiles, as well as KDevelop, Visual Studio, and (Apple) XCode project files from the same configuration file. In addition, CMake also provides a GUI front end and which allows an interactive build and installation process.

The SUNDIALS build process requires CMake version `2.8.1` or higher and a working compiler. On Unix-like operating systems, it also requires Make (and `curses`, including its development libraries, for the GUI front end to CMake, `ccmake`), while on Windows it requires Visual Studio. While many Linux distributions offer CMake, the version included is probably out of date. Many new CMake features have been added recently, and you should download the latest version from `http://www.cmake.org`. Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake website. Once CMake is installed, Linux/Unix users will be able to use `ccmake`, while Windows users will be able to use `CMakeSetup`.

As previously noted, when using CMake to configure, build and install SUNDIALS, it is always required to use a separate build directory. While in-source builds are possible, they are explicitly prohibited by the SUNDIALS CMake scripts (one of the reasons being that, unlike autotools, CMake does not provide a `make distclean` procedure and it is therefore difficult to clean-up the source tree after an in-source build). By ensuring a separate build directory, it is an easy task for the user to clean-up all traces of the build by simply removing the build directory. CMake does generate a `make clean` which will remove files generated by the compiler and linker.

A.1.1 Configuring, building, and installing on Unix-like systems

The default CMake configuration will build all included solvers and associated examples and will build static and shared libraries. The *installdir* defaults to `/usr/local` and can be changed by setting the `CMAKE_INSTALL_PREFIX` variable. Support for FORTRAN and all other options are disabled.

CMake can be used from the command line with the `cmake` command, or from a `curses`-based GUI by using the `ccmake` command. Examples for using both methods will be presented. For the examples shown it is assumed that there is a top level SUNDIALS directory with appropriate source, build and install directories:

```
% mkdir (...)sundials/instdir
% mkdir (...)sundials/builddir
% cd (...)sundials/builddir
```

Building with the GUI

Using CMake with the GUI follows this general process:

- Select and modify values, run configure (`c` key)
- New values are denoted with an asterisk
- To set a variable, move the cursor to the variable and press enter
  - If it is a boolean (ON/OFF) it will toggle the value
  - If it is string or file, it will allow editing of the string

  - For file and directories, the `<tab>` key can be used to complete

- Repeat until all values are set as desired and the generate option is available (`g` key)
- Some variables (advanced variables) are not visible right away
- To see advanced variables, toggle to advanced mode (`t` key)
- To search for a variable press `/` key, and to repeat the search, press the `n` key

To build the default configuration using the GUI, from the *builddir* enter the ccmake command and point to the *srcdir*:

```
% ccmake ../srcdir
```

The default configuration screen is shown in Figure A.1.

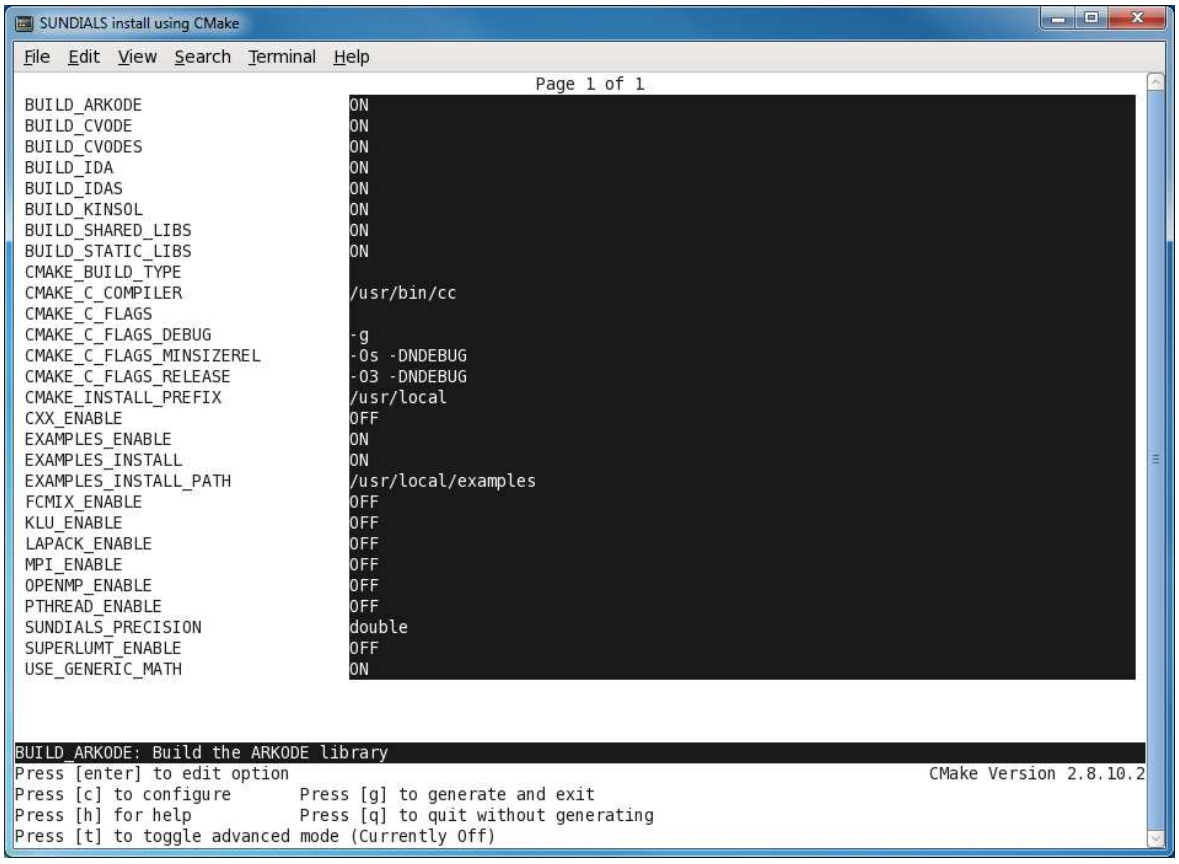


Figure A.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

The default *instdir* for both SUNDIALS and corresponding examples can be changed by setting the `CMAKE_INSTALL_PREFIX` and the `EXAMPLES_INSTALL_PATH` as shown in figure A.2.

Pressing the (`g` key) will generate makefiles including all dependencies and all rules to build SUNDIALS on this system. Back at the command prompt, you can now run:

```
% make
```

To install SUNDIALS in the installation directory specified in the configuration, simply run:

```
% make install
```

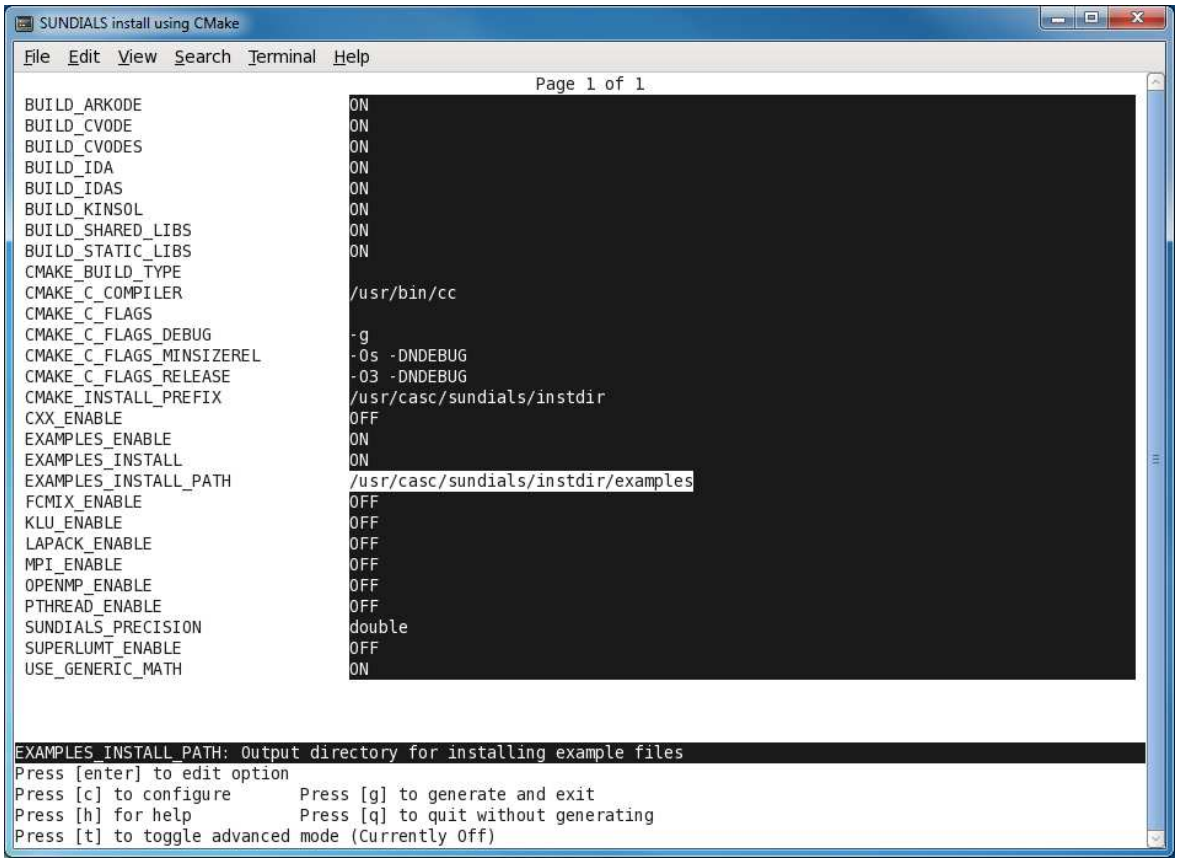


Figure A.2: Changing the *instdir* for SUNDIALS and corresponding examples

Building from the command line

Using CMake from the command line is simply a matter of specifying CMake variable settings with the cmake command. The following will build the default configuration:

```
% cmake -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> ../srcdir
% make
% make install
```

A.1.2 Configuration options (Unix/Linux)

A complete list of all available options for a CMake-based SUNDIALS configuration is provide below. Note that the default values shown are for a typical configuration on a Linux system and are provided as illustration only.

`BUILD_ARKODE` - Build the ARKODE library
Default: ON

`BUILD_CVODE` - Build the CVODE library
Default: ON

`BUILD_CVODES` - Build the CVODES library
Default: ON

`BUILD_IDA` - Build the IDA library
Default: ON

`BUILD_IDAS` - Build the IDAS library
Default: ON

`BUILD_KINSOL` - Build the KINSOL library
Default: ON

`BUILD_SHARED_LIBS` - Build shared libraries
Default: OFF

`BUILD_STATIC_LIBS` - Build static libraries
Default: ON

`CMAKE_BUILD_TYPE` - Choose the type of build, options are: None (CMAKE_C_FLAGS used) Debug Release RelWithDebInfo MinSizeRel
Default:

`CMAKE_C_COMPILER` - C compiler
Default: /usr/bin/cc

`CMAKE_C_FLAGS` - Flags for C compiler
Default:

`CMAKE_C_FLAGS_DEBUG` - Flags used by the compiler during debug builds
Default: -g

`CMAKE_C_FLAGS_MINSIZEREL` - Flags used by the compiler during release minsize builds
Default: -Os -DNDEBUG

`CMAKE_C_FLAGS_RELEASE` - Flags used by the compiler during release builds
Default: -O3 -DNDEBUG

`CMAKE_Fortran_COMPILER` - Fortran compiler
Default: /usr/bin/gfortran
Note: Fortran support (and all related options) are triggered only if either Fortran-C support is enabled (`FCMIX_ENABLE` is ON) or Blas/Lapack support is enabled (`LAPACK_ENABLE` is ON).

`CMAKE_Fortran_FLAGS` - Flags for Fortran compiler
Default:

`CMAKE_Fortran_FLAGS_DEBUG` - Flags used by the compiler during debug builds
Default:

`CMAKE_Fortran_FLAGS_MINSIZEREL` - Flags used by the compiler during release minsize builds
Default:

`CMAKE_Fortran_FLAGS_RELEASE` - Flags used by the compiler during release builds
Default:

`CMAKE_INSTALL_PREFIX` - Install path prefix, prepended onto install directories
Default: /usr/local
Note: The user must have write access to the location specified through this option. Exported SUNDIALS header files and libraries will be installed under subdirectories `include` and `lib` of `CMAKE_INSTALL_PREFIX`, respectively.

`EXAMPLES_ENABLE` - Build the SUNDIALS examples
Default: ON

EXAMPLES_INSTALL - Install example files
Default: ON
Note: This option is triggered only if building example programs is enabled (EXAMPLES_ENABLE ON). If the user requires installation of example programs then the sources and sample output files for all SUNDIALS modules that are currently enabled will be exported to the directory specified by EXAMPLES_INSTALL_PATH. A CMake configuration script will also be automatically generated and exported to the same directory. Additionally, if the configuration is done under a Unix-like system, makefiles for the compilation of the example programs (using the installed SUNDIALS libraries) will be automatically generated and exported to the directory specified by EXAMPLES_INSTALL_PATH.

EXAMPLES_INSTALL_PATH - Output directory for installing example files
Default: /usr/local/examples
Note: The actual default value for this option will an examples subdirectory created under CMAKE_INSTALL_PREFIX.

FCMIX_ENABLE - Enable Fortran-C support
Default: OFF

KLU_ENABLE - Enable KLU support
Default: OFF

LAPACK_ENABLE - Enable Lapack support
Default: OFF
Note: Setting this option to ON will trigger the two additional options see below.

LAPACK_LIBRARIES - Lapack (and Blas) libraries
Default: /usr/lib/liblapack.so;/usr/lib/libblas.so
Note: CMake will search for these libraries in your LD_LIBRARY_PATH prior to searching default system paths.

MPI_ENABLE - Enable MPI support
Default: OFF
Note: Setting this option to ON will trigger several additional options related to MPI.

MPI_MPICC - mpicc program
Default:

MPI_RUN_COMMAND - Specify run command for MPI
Default: mpirun
Note: This can either be set to mpirun for OpenMPI or srun if jobs are managed by SLURM - Simple Linux Utility for Resource Management as exists on LLNL's high performance computing clusters.

MPI_MPIF77 - mpif77 program
Default:
Note: This option is triggered only if using MPI compiler scripts (MPI_USE_MPISCRIPTS is ON) and Fortran-C support is enabled (FCMIx_ENABLE is ON).

OPENMP_ENABLE - Enable OpenMP support
Default: OFF
Turn on support for the OpenMP based nvector.

PTHREAD_ENABLE - Enable Pthreads support
Default: OFF
Turn on support for the Pthreads based nvector.

SUNDIALS_PRECISION - Precision used in SUNDIALS, options are: double, single or extended
Default: double

SUPERLUMT_ENABLE - Enable SUPERLU_MT support
Default: OFF

USE_GENERIC_MATH - Use generic (stdc) math libraries
Default: ON

A.1.3 Configuration examples

The following examples will help demonstrate usage of the CMake configure options.
To configure SUNDIALS using the default C and Fortran compilers, and default `mpicc` and `mpif77` parallel compilers, enable compilation of examples, and install libraries, headers, and example sources under subdirectories of `/home/myname/sundials/`, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
> /home/myname/sundials/srcdir
%
% make install
%
```

To disable installation of the examples, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
> -DEXAMPLES_INSTALL=OFF \
> /home/myname/sundials/srcdir
%
% make install
%
```

A.1.4 Working with external Libraries

The SUNDIALS Suite contains many options to enable implementation flexibility when developing solutions. The following are some notes addressing specific configurations when using the supported third party libraries.

Building with LAPACK and BLAS

To enable LAPACK and BLAS libraries, set the `LAPACK_ENABLE` option to `ON`. If the directory containing the LAPACK and BLAS libraries is in the `LD_LIBRARY_PATH` environment variable, CMake will set the `LAPACK_LIBRARIES` variable accordingly, otherwise CMake will attemp to find the LAPACK and BLAS libraries in standard system locations. To explicitly tell CMake what libraries to use, the `LAPACK_LIBRARIES` varible can be set to the desired libraries. Example:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DLAPACK_LIBRARIES=/mypath/lib/liblapack.so;/mypath/lib/libblas.so \
> /home/myname/sundials/srcdir
%
% make install
%
```

Building with KLU

The KLU libraries are part of SuiteSparse, a suite of sparse matrix software, available from the Texas A&M University website: `http://faculty.cse.tamu.edu/davis/suitesparse.html`. SUNDIALS has been tested with SuiteSparse version 4.2.1. To enable KLU, set `KLU_ENABLE` to `ON`, set `KLU_INCLUDE_DIR` to the `include` path of the KLU installation and set `KLU_LIBRARY_DIR` to the `lib` path of the KLU installation. The CMake configure will result in populating the following variables: `AMD_LIBRARY`, `AMD_LIBRARY_DIR`, `BTF_LIBRARY`, `BTF_LIBRARY_DIR`, `COLAMD_LIBRARY`, `COLAMD_LIBRARY_DIR`, and `KLU_LIBRARY`

Building with SuperLU_MT

The SuperLU_MT libraries are available for download from the Lawrence Berkeley National Laboratory website: `http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu_mt`. SUNDIALS has been tested with SuperLU_MT version 2.4. To enable SuperLU_MT, set `SUPERLUMT_ENABLE` to `ON`, set `SUPERLUMT_INCLUDE_DIR` to the `SRC` path of the SuperLU_MT installation, and set the variable `SUPERLUMT_LIBRARY_DIR` to the `lib` path of the SuperLU_MT installation. At the same time, the variable `SUPERLUMT_THREAD_TYPE` must be set to either `Pthread` or `OpenMP`.

Do not mix thread types when building SUNDIALS solvers. If threading is enabled for SUNDIALS by having either `OPENMP_ENABLE` or `PTHREAD_ENABLE` set to `ON` then SuperLU_MT should be set to use the same threading type.

A.2 Building and Running Examples

Each of the SUNDIALS solvers is distributed with a set of examples demonstrating basic usage. To build and install the examples, set both `EXAMPLES_ENABLE` and `EXAMPLES_INSTALL` to `ON`. Specify the installation path for the examples with the variable `EXAMPLES_INSTALL_PATH`. CMake will generate `CMakeLists.txt` configuration files (and `Makefile` files if on Linux/Unix) that reference the *installed* SUNDIALS headers and libraries.

Either the `CMakeLists.txt` file or the traditional `Makefile` may be used to build the examples as well as serve as a template for creating user developed solutions. To use the supplied `Makefile` simply run `make` to compile and generate the executables. To use CMake from within the installed example directory, run `cmake` (or `ccmake` to use the GUI) followed by `make` to compile the example code. Note that if CMake is used, it will overwrite the traditional `Makefile` with a new CMake-generated `Makefile`. The resulting output from running the examples can be compared with example output bundled in the SUNDIALS distribution.

NOTE: There will potentially be differences in the output due to machine architecture, compiler versions, use of third party libraries etc.

A.3 Configuring, building, and installing on Windows

CMake can also be used to build SUNDIALS on Windows. To build SUNDIALS for use with Visual Studio the following steps should be performed:

1. Unzip the downloaded tar file(s) into a directory. This will be the *srcdir*
2. Create a separate *builddir*
3. Open a Visual Studio Command Prompt and cd to *builddir*
4. Run cmake-gui ../*srcdir*
   (a) Hit Configure
   (b) Check/Uncheck solvers to be built
   (c) Change CMAKE_INSTALL_PREFIX to *instdir*

(d) Set other options as desired

(e) Hit Generate

5. Back in the VS Command Window:

(a) Run msbuild ALL_BUILD.vcxproj

(b) Run msbuild INSTALL.vcxproj

The resulting libraries will be in the *instdir*. The SUNDIALS project can also now be opened in Visual Studio. Double click on the ALL_BUILD.vcxproj file to open the project. Build the whole *solution* to create the SUNDIALS libraries. To use the SUNDIALS libraries in your own projects, you must set the include directories for your project, add the SUNDIALS libraries to your project solution, and set the SUNDIALS libraries as dependencies for your project.

A.4 Installed libraries and exported header files

Using the CMake SUNDIALS build system, the command

```
% make install
```

will install the libraries under *libdir* and the public header files under *includedir*. The values for these directories are *instdir*`/lib` and *instdir*`/include`, respectively. The location can be changed by setting the CMake variable `CMAKE_INSTALL_PREFIX`. Although all installed libraries reside under *libdir*`/lib`, the public header files are further organized into subdirectories under *includedir*`/include`.

The installed libraries and exported header files are listed for reference in Tables A.1 and A.2. The file extension *.lib* is typically `.so` for shared libraries and `.a` for static libraries. Note that, in the Tables, names are relative to *libdir* for libraries and to *includedir* for header files.

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

Table A.1: SUNDIALS libraries and header files

SHARED	Libraries	n/a	
	Header files	sundials/sundials_config.h	sundials/sundials_types.h
		sundials/sundials_math.h	
		sundials/sundials_nvector.h	sundials/sundials_fnvector.h
		sundials/sundials_direct.h	sundials/sundials_lapack.h
		sundials/sundials_dense.h	sundials/sundials_band.h
		sundials/sundials_sparse.h	
		sundials/sundials_iterative.h	sundials/sundials_spgmr.h
		sundials/sundials_spbcgs.h	sundials/sundials_sptfqmr.h
		sundials/sundials_pcg.h	sundials/sundials_spfgmr.h
NVECTOR_SERIAL	Libraries	libsundials_nvecserial.*lib*	libsundials_fnvecserial.a
	Header files	nvector/nvector_serial.h	
NVECTOR_PARALLEL	Libraries	libsundials_nvecparallel.*lib*	libsundials_fnvecparallel.a
	Header files	nvector/nvector_parallel.h	
NVECTOR_OPENMP	Libraries	libsundials_nvecopenmp.*lib*	libsundials_fnvecopenmp.a
	Header files	nvector/nvector_openmp.h	
NVECTOR_PTHREADS	Libraries	libsundials_nvecpthreads.*lib*	libsundials_fnvecpthreads.a
	Header files	nvector/nvector_pthreads.h	
CVODE	Libraries	libsundials_cvode.*lib*	libsundials_fcvode.a
	Header files	cvode/cvode.h	cvode/cvode_impl.h
		cvode/cvode_direct.h	cvode/cvode_lapack.h
		cvode/cvode_dense.h	cvode/cvode_band.h
		cvode/cvode_diag.h	
		cvode/cvode_sparse.h	cvode/cvode_klu.h
		cvode/cvode_superlumt.h	
		cvode/cvode_spils.h	cvode/cvode_spgmr.h
		cvode/cvode_sptfqmr.h	cvode/cvode_spbcgs.h
		cvode/cvode_bandpre.h	cvode/cvode_bbdpre.h
CVODES	Libraries	libsundials_cvodes.*lib*	
	Header files	cvodes/cvodes.h	cvodes/cvodes_impl.h
		cvodes/cvodes_direct.h	cvodes/cvodes_lapack.h
		cvodes/cvodes_dense.h	cvodes/cvodes_band.h
		cvodes/cvodes_diag.h	
		cvodes/cvodes_sparse.h	cvodes/cvodes_klu.h
		cvodes/cvodes_superlumt.h	
		cvodes/cvodes_spils.h	cvodes/cvodes_spgmr.h
		cvodes/cvodes_sptfqmr.h	cvodes/cvodes_spbcgs.h
		cvodes/cvodes_bandpre.h	cvodes/cvodes_bbdpre.h
ARKODE	Libraries	libsundials_arkode.*lib*	libsundials_farkode.a
	Header files	arkode/arkode.h	arkode/arkode_impl.h
		arkode/arkode_direct.h	arkode/arkode_lapack.h
		arkode/arkode_dense.h	arkode/arkode_band.h
		arkode/arkode_sparse.h	arkode/arkode_klu.h
		arkode/arkode_superlumt.h	
		arkode/arkode_spils.h	arkode/arkode_spgmr.h
		arkode/arkode_sptfqmr.h	arkode/arkode_spbcgs.h
		arkode/arkode_pcg.h	arkode/arkode_spfgmr.h
		arkode/arkode_bandpre.h	arkode/arkode_bbdpre.h

Table A.2: SUNDIALS libraries and header files (cont.)

IDA	Libraries	libsundials_ida.*lib*	libsundials_fida.a
	Header files	ida/ida.h	ida/ida_impl.h
		ida/ida_direct.h	ida/ida_lapack.h
		ida/ida_dense.h	ida/ida_band.h
		ida/ida_sparse.h	ida/ida_klu.h
		ida/ida_superlumt.h	
		ida/ida_spils.h	ida/ida_spgmr.h
		ida/ida_spbcgs.h	ida/ida_sptfqmr.h
		ida/ida_bbdpre.h	
IDAS	Libraries	libsundials_idas.*lib*	
	Header files	idas/idas.h	idas/idas_impl.h
		idas/idas_direct.h	idas/idas_lapack.h
		idas/idas_dense.h	idas/idas_band.h
		idas/idas_sparse.h	idas/idas_klu.h
		idas/idas_superlumt.h	
		idas/idas_spils.h	idas/idas_spgmr.h
		idas/idas_spbcgs.h	idas/idas_sptfqmr.h
		idas/idas_bbdpre.h	
KINSOL	Libraries	libsundials_kinsol.*lib*	libsundials_fkinsol.a
	Header files	kinsol/kinsol.h	kinsol/kinsol_impl.h
		kinsol/kinsol_direct.h	kinsol/kinsol_lapack.h
		kinsol/kinsol_dense.h	kinsol/kinsol_band.h
		kinsol/kinsol_sparse.h	kinsol/kinsol_klu.h
		kinsol/kinsol_superlumt.h	
		kinsol/kinsol_spils.h	kinsol/kinsol_spgmr.h
		kinsol/kinsol_spbcgs.h	kinsol/kinsol_sptfqmr.h
		kinsol/kinsol_bbdpre.h	kinsol/kinsol_spfgmr.h

Appendix B

IDA Constants

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

B.1 IDA input constants

IDA **main solver module**		
IDA_NORMAL	1	Solver returns at specified output time.
IDA_ONE_STEP	2	Solver returns after each successful step.
IDA_YA_YDP_INIT	1	Compute y_a and $\dot{y}_d$, given y_d.
IDA_Y_INIT	2	Compute y, given $\dot{y}$.
Iterative linear solver module		
PREC_NONE	0	No preconditioning
PREC_LEFT	1	Preconditioning on the left.
MODIFIED_GS	1	Use modified Gram-Schmidt procedure.
CLASSICAL_GS	2	Use classical Gram-Schmidt procedure.

B.2 IDA output constants

IDA **main solver module**		
IDA_SUCCESS	0	Successful function return.
IDA_TSTOP_RETURN	1	IDASolve succeeded by reaching the specified stopping point.
IDA_ROOT_RETURN	2	IDASolve succeeded and found one or more roots.
IDA_WARNING	99	IDASolve succeeded but an unusual situation occurred.
IDA_TOO_MUCH_WORK	-1	The solver took mxstep internal steps but could not reach tout.
IDA_TOO_MUCH_ACC	-2	The solver could not satisfy the accuracy demanded by the user for some internal step.
IDA_ERR_FAIL	-3	Error test failures occurred too many times during one internal time step or minimum step size was reached.
IDA_CONV_FAIL	-4	Convergence test failures occurred too many times during one internal time step or minimum step size was reached.

IDA_LINIT_FAIL	-5	The linear solver's initialization function failed.
IDA_LSETUP_FAIL	-6	The linear solver's setup function failed in an unrecoverable manner.
IDA_LSOLVE_FAIL	-7	The linear solver's solve function failed in an unrecoverable manner.
IDA_RES_FAIL	-8	The user-provided residual function failed in an unrecoverable manner.
IDA_REP_RES_FAIL	-9	The user-provided residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.
IDA_RTFUNC_FAIL	-10	The rootfinding function failed in an unrecoverable manner.
IDA_CONSTR_FAIL	-11	The inequality constraints were violated and the solver was unable to recover.
IDA_FIRST_RES_FAIL	-12	The user-provided residual function failed recoverably on the first call.
IDA_LINESEARCH_FAIL	-13	The line search failed.
IDA_NO_RECOVERY	-14	The residual function, linear solver setup function, or linear solver solve function had a recoverable failure, but IDACalcIC could not recover.
IDA_MEM_NULL	-20	The ida_mem argument was NULL.
IDA_MEM_FAIL	-21	A memory allocation failed.
IDA_ILL_INPUT	-22	One of the function inputs is illegal.
IDA_NO_MALLOC	-23	The IDA memory was not allocated by a call to IDAInit.
IDA_BAD_EWT	-24	Zero value of some error weight component.
IDA_BAD_K	-25	The k-th derivative is not available.
IDA_BAD_T	-26	The time t is outside the last step taken.
IDA_BAD_DKY	-27	The vector argument where derivative should be stored is NULL.

IDADLS **linear solver modules**

IDADLS_SUCCESS	0	Successful function return.
IDADLS_MEM_NULL	-1	The ida_mem argument was NULL.
IDADLS_LMEM_NULL	-2	The IDADLS linear solver has not been initialized.
IDADLS_ILL_INPUT	-3	The IDADLS solver is not compatible with the current NVECTOR module.
IDADLS_MEM_FAIL	-4	A memory allocation request failed.
IDADLS_JACFUNC_UNRECVR	-5	The Jacobian function failed in an unrecoverable manner.
IDADLS_JACFUNC_RECVR	-6	The Jacobian function had a recoverable error.

IDASLS **linear solver module**

IDASLS_SUCCESS	0	Successful function return.
IDASLS_MEM_NULL	-1	The ida_mem argument was NULL.
IDASLS_LMEM_NULL	-2	The IDASLS linear solver has not been initialized.
IDASLS_ILL_INPUT	-3	The IDASLS solver is not compatible with the current NVECTOR module or other input is invalid.
IDASLS_MEM_FAIL	-4	A memory allocation request failed.
IDASLS_JAC_NOSET	-5	The Jacobian evaluation routine was not been set before the linear solver setup routine was called.

IDASLS_PACKAGE_FAIL	-6	An external package call return a failure error code.
IDASLS_JACFUNC_UNRECVR	-7	The Jacobian function failed in an unrecoverable manner.
IDASLS_JACFUNC_RECVR	-8	The Jacobian function had a recoverable error.

IDASPILS **linear solver modules**

IDASPILS_SUCCESS	0	Successful function return.
IDASPILS_MEM_NULL	-1	The ida_mem argument was NULL.
IDASPILS_LMEM_NULL	-2	The IDASPILS linear solver has not been initialized.
IDASPILS_ILL_INPUT	-3	The IDASPILS solver is not compatible with the current NVECTOR module.
IDASPILS_MEM_FAIL	-4	A memory allocation request failed.
IDASPILS_PMEM_NULL	-5	The preconditioner module has not been initialized.

SPGMR **generic linear solver module**

SPGMR_SUCCESS	0	Converged.
SPGMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPGMR_CONV_FAIL	2	Failure to converge.
SPGMR_QRFACT_FAIL	3	A singular matrix was found during the QR factorization.
SPGMR_PSOLVE_FAIL_REC	4	The preconditioner solve function failed recoverably.
SPGMR_ATIMES_FAIL_REC	5	The Jacobian-times-vector function failed recoverably.
SPGMR_PSET_FAIL_REC	6	The preconditioner setup routine failed recoverably.
SPGMR_MEM_NULL	-1	The SPGMR memory is NULL
SPGMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPGMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPGMR_GS_FAIL	-4	Failure in the Gram-Schmidt procedure.
SPGMR_QRSOL_FAIL	-5	The matrix R was found to be singular during the QR solve phase.
SPGMR_PSET_FAIL_UNREC	-6	The preconditioner setup routine failed unrecoverably.

SPFGMR **generic linear solver module (only available in** KINSOL **and** ARKODE**)**

SPFGMR_SUCCESS	0	Converged.
SPFGMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPFGMR_CONV_FAIL	2	Failure to converge.
SPFGMR_QRFACT_FAIL	3	A singular matrix was found during the QR factorization.
SPFGMR_PSOLVE_FAIL_REC	4	The preconditioner solve function failed recoverably.
SPFGMR_ATIMES_FAIL_REC	5	The Jacobian-times-vector function failed recoverably.
SPFGMR_PSET_FAIL_REC	6	The preconditioner setup routine failed recoverably.
SPFGMR_MEM_NULL	-1	The SPFGMR memory is NULL
SPFGMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPFGMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPFGMR_GS_FAIL	-4	Failure in the Gram-Schmidt procedure.
SPFGMR_QRSOL_FAIL	-5	The matrix R was found to be singular during the QR solve phase.
SPFGMR_PSET_FAIL_UNREC	-6	The preconditioner setup routine failed unrecoverably.

SPBCG **generic linear solver module**

SPBCG_SUCCESS	0	Converged.
SPBCG_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPBCG_CONV_FAIL	2	Failure to converge.
SPBCG_PSOLVE_FAIL_REC	3	The preconditioner solve function failed recoverably.
SPBCG_ATIMES_FAIL_REC	4	The Jacobian-times-vector function failed recoverably.
SPBCG_PSET_FAIL_REC	5	The preconditioner setup routine failed recoverably.
SPBCG_MEM_NULL	-1	The SPBCG memory is NULL
SPBCG_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPBCG_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPBCG_PSET_FAIL_UNREC	-4	The preconditioner setup routine failed unrecoverably.

SPTFQMR **generic linear solver module**

SPTFQMR_SUCCESS	0	Converged.
SPTFQMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPTFQMR_CONV_FAIL	2	Failure to converge.
SPTFQMR_PSOLVE_FAIL_REC	3	The preconditioner solve function failed recoverably.
SPTFQMR_ATIMES_FAIL_REC	4	The Jacobian-times-vector function failed recoverably.
SPTFQMR_PSET_FAIL_REC	5	The preconditioner setup routine failed recoverably.
SPTFQMR_MEM_NULL	-1	The SPTFQMR memory is NULL
SPTFQMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed.
SPTFQMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPTFQMR_PSET_FAIL_UNREC	-4	The preconditioner setup routine failed unrecoverably.

Bibliography

[1] KLU Sparse Matrix Factorization Library. http://faculty.cse.tamu.edu/davis/suitesparse.html.

[2] SuperLU_MT Threaded Sparse Matrix Factorization Library. http://crd-legacy.lbl.gov/ xiaoye/-SuperLU/.

[3] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM, Philadelphia, Pa, 1996.

[4] P. N. Brown and A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.

[5] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems. *SIAM J. Sci. Comput.*, 15:1467–1488, 1994.

[6] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Consistent Initial Condition Calculation for Differential-Algebraic Systems. *SIAM J. Sci. Comput.*, 19:1495–1512, 1998.

[7] G. D. Byrne. Pragmatic Experiments with Krylov Methods in the Stiff ODE Setting. In J.R. Cash and I. Gladwell, editors, *Computational Ordinary Differential Equations*, pages 323–356, Oxford, 1992. Oxford University Press.

[8] G. D. Byrne and A. C. Hindmarsh. User Documentation for PVODE, An ODE Solver for Parallel Computers. Technical Report UCRL-ID-130884, LLNL, May 1998.

[9] G. D. Byrne and A. C. Hindmarsh. PVODE, An ODE Solver for Parallel Computers. *Intl. J. High Perf. Comput. Apps.*, 13(4):254–365, 1999.

[10] S. D. Cohen and A. C. Hindmarsh. CVODE, a Stiff/Nonstiff ODE Solver in C. *Computers in Physics*, 10(2):138–143, 1996.

[11] A. M. Collier, A. C. Hindmarsh, R. Serban, and C.S. Woodward. User Documentation for KINSOL v2.7.0. Technical Report UCRL-SM-208116, LLNL, 2011.

[12] T. A. Davis and P. N. Ekanathan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3), 2010.

[13] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[14] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.

[15] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.

[16] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS, suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, (31):363–396, 2005.

[17] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v2.7.0. Technical Report UCRL-SM-208108, LLNL, 2011.

[18] A. C. Hindmarsh, R. Serban, and A. Collier. Example Programs for IDA v2.7.0. Technical Report UCRL-SM-208113, LLNL, 2011.

[19] A. C. Hindmarsh and A. G. Taylor. PVODE and KINSOL: Parallel Software for Differential and Nonlinear Systems. Technical Report UCRL-ID-129739, LLNL, February 1998.

[20] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[21] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.

[22] H. A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.

Index

AddIdentitySparseMat, 121

BAND generic linear solver
  functions, 118
    small matrix, 118–119
  macros, 115
  type DlsMat, 112–115
BAND_COL, 62, 115
BAND_COL_ELEM, 62, 115
BAND_ELEM, 62, 115
bandAddIdentity, 119
bandCopy, 119
bandGETRF, 119
bandGETRS, 119
bandMatvec, 119
bandScale, 119
Bi-CGStab method, 41, 125
BIG_REAL, 16, 95

CLASSICAL_GS, 39
CONSTR_VEC, 85
CopySparseMat, 121

data types
  Fortran, 73
DENSE generic linear solver
  functions
    large matrix, 115–116
    small matrix, 116–118
  macros, 115
  type DlsMat, 112–115
DENSE_COL, 61, 115
DENSE_ELEM, 61, 115
denseAddIdentity, 117
denseCopy, 117
denseGEQRF, 117
denseGETRF, 117
denseGETRS, 117
denseMatvec, 118
denseORMQR, 118
densePOTRF, 117
densePOTRS, 117
denseScale, 117
destroyArray, 117, 119
destroyMat, 116, 118
DestroySparseMat, 121
DlsMat, 61, 62, 112

eh_data, 59
error messages, 29
  redirecting, 31
  user-defined handler, 31, 59

FGMRES method, 125
FIDA interface module
  interface to the IDABBDPRE module, 88–89
  optional input and output, 84
  rootfinding, 87
  usage, 75–84
  user-callable functions, 74–75
  user-supplied functions, 75
FIDABAND, 78
FIDABANDSETJAC, 79
FIDABBDINIT, 88
FIDABBDOPT, 89
FIDABBDREINIT, 89
FIDABJAC, 78
FIDACOMMFN, 89
FIDADENSE, 77
FIDADENSESETJAC, 78
FIDADJAC, 78
FIDAEWT, 77
FIDAEWTSET, 77
FIDAFREE, 84
FIDAGETDKY, 83
FIDAGETERRWEIGHTS, 84
FIDAGETESTLOCALERR, 87
FIDAGLOCFN, 89
FIDAJTIMES, 81, 90
FIDAKLU, 79
FIDAKLURENIT, 79
FIDAMALLOC, 77
FIDAMALLOC, 76
FIDAPSET, 82
FIDAPSOL, 82
FIDAREINIT, 83
FIDARESFUN, 75
FIDASETIIN, 84

FIDASETRIN, 84
FIDASETVIN, 84
FIDASOLVE, 83
FIDASPBCG, 80
FIDASPBCGREINIT, 83
FIDASPGMR, 80
FIDASPGMRREINIT, 83
FIDASPILSSETJAC, 81
FIDASPILSSETPREC, 81
FIDASPJAC, 80
FIDASPTFQMR, 81
FIDASPTFQMRREINIT, 84
FIDASUPERLUMT, 79
FIDATOLREINIT, 84
FNVINITOMP, 76
FNVINITP, 76
FNVINITPTS, 76
FNVINITS, 76

generic linear solvers
  BAND, 112
  DENSE, 112
  KLU, 119
  SLS, 119
  SPBCG, 125
  SPFGMR, 125
  SPGMR, 124
  SPTFQMR, 125
  SUPERLUMT, 119
  use in IDA, 14
GMRES method, 124
Gram-Schmidt procedure, 39

half-bandwidths, 24, 61–63, 69
header files, 16, 68

ID_VEC, 85
IDA
  motivation for writing in C, 1
  package structure, 11
IDA linear solvers
  built on generic solvers, 23
  header files, 16
  IDABAND, 24
  IDADENSE, 23
  IDAKLU, 24
  IDASPBCG, 26
  IDASPGMR, 25
  IDASPTFQMR, 26
  IDASUPERLUMT, 25
  implementation details, 14
  list of, 11–14
  NVECTOR compatibility, 15
  selecting one, 23
ida.h, 16

IDA_BAD_DKY, 44
IDA_BAD_EWT, 27
IDA_BAD_K, 44
IDA_BAD_T, 44
ida_band.h, 17
IDA_CONSTR_FAIL, 27, 29
IDA_CONV_FAIL, 27, 29
ida_dense.h, 16
IDA_ERR_FAIL, 29
IDA_FIRST_RES_FAIL, 27
IDA_ILL_INPUT, 20, 21, 27, 29, 32–35, 41–43, 51, 58
ida_klu.h, 17
ida_lapack.h, 17
IDA_LINESEARCH_FAIL, 27
IDA_LINIT_FAIL, 27, 29
IDA_LSETUP_FAIL, 27, 29
IDA_LSOLVE_FAIL, 27, 29
IDA_MEM_FAIL, 20
IDA_MEM_NULL, 20, 21, 27, 29, 31–35, 41–44, 46–52, 58
IDA_NO_MALLOC, 21, 22, 27, 58
IDA_NO_RECOVERY, 27
IDA_NORMAL, 28
IDA_ONE_STEP, 28
IDA_REP_RES_ERR, 29
IDA_RES_FAIL, 27, 29
IDA_ROOT_RETURN, 28
IDA_RTFUNC_FAIL, 29, 60
ida_spbcgs.h, 17
ida_spgmr.h, 17
ida_sptfqmr.h, 17
IDA_SUCCESS, 20, 21, 27, 28, 31–35, 41–44, 52, 58
ida_superlumt.h, 17
IDA_TOO_MUCH_ACC, 29
IDA_TOO_MUCH_WORK, 29
IDA_TSTOP_RETURN, 28
IDA_WARNING, 59
IDA_Y_INIT, 27
IDA_YA_YDP_INIT, 27
IDABAND linear solver
  Jacobian approximation used by, 36
  memory requirements, 52
  NVECTOR compatibility, 24
  optional input, 35–36
  optional output, 52–54
  selection of, 24
  use in FIDA, 78
IDABand, 19, 23, 24, 61
IDABAND_ILL_INPUT, 24
IDABAND_MEM_FAIL, 24
IDABAND_MEM_NULL, 24
IDABAND_SUCCESS, 24
IDABBDPRE preconditioner

description, 66–67
optional output, 70–71
usage, 68–69
user-callable functions, 69–70
user-supplied functions, 67–68
IDABBDPrecGetNumGfnEvals, 71
IDABBDPrecGetWorkSpace, 70
IDABBDPrecInit, 69
IDABBDPrecReInit, 70
IDACalcIC, 27
IDACreate, 20
IDADENSE linear solver
Jacobian approximation used by, 35
memory requirements, 52
NVECTOR compatibility, 23
optional input, 35–36
optional output, 52–54
selection of, 23
use in FIDA, 77
IDADense, 18, 23, 60
IDADLS_ILL_INPUT, 23
IDADLS_LMEM_NULL, 36, 53
IDADLS_MEM_FAIL, 23
IDADLS_MEM_NULL, 23, 36, 53
IDADLS_SUCCESS, 23, 36, 53
IDADlsBandJacFn, 61
IDADlsDenseJacFn, 60
IDADlsGetLastFlag, 53
IDADlsGetNumJacEvals, 53
IDADlsGetNumResEvals, 53
IDADlsGetReturnFlagName, 54
IDADlsGetWorkSpace, 52
IDADlsSetBandJacFn, 36
IDADlsSetDenseJacFn, 35
IDAErrHandlerFn, 59
IDAEwtFn, 59
IDAFree, 19, 20
IDAGetActualInitStep, 48
IDAGetConsistentIC, 51
IDAGetCurrentOrder, 48
IDAGetCurrentStep, 48
IDAGetCurrentTime, 49
IDAGetDky, 44
IDAGetErrWeights, 49
IDAGetEstLocalErrors, 49
IDAGetIntegratorStats, 50
IDAGetLastOrder, 47
IDAGetLastStep, 48
IDAGetNonlinSolvStats, 51
IDAGetNumBacktrackOps, 51
IDAGetNumErrTestFails, 47
IDAGetNumGEvals, 52
IDAGetNumLinSolvSetups, 47
IDAGetNumNonlinSolvConvFails, 50
IDAGetNumNonlinSolvIters, 50
IDAGetNumResEvals, 47
IDAGetNumSteps, 46
IDAGetReturnFlagName, 51
IDAGetRootInfo, 52
IDAGetTolScaleFactor, 49
IDAGetWorkSpace, 46
IDAInit, 20, 57
IDAKLU, 19, 23, 25, 63
IDAKLU linear solver
Jacobian approximation used by, 36
matrix reordering algorithm specification, 37
NVECTOR compatibility, 24
optional input, 36–38
optional output, 54
reinitialization, 37
selection of, 24
IDAKLUReInit, 37
IDAKLUSetOrdering, 37
IDALapackBand, 19, 23, 24, 61
IDALapackDense, 19, 23, 24, 60
IDAReInit, 57, 58
IDAResFn, 20, 58
IDARootFn, 60
IDARootInit, 28
IDASetConstraints, 35
IDASetErrFile, 31
IDASetErrHandlerFn, 31
IDASetId, 35
IDASetInitStep, 32
IDASetLineSearchOffIC, 42
IDASetMaxConvFails, 34
IDASetMaxErrTestFails, 33
IDASetMaxNonlinIters, 33
IDASetMaxNumItersIC, 42
IDASetMaxNumJacsIC, 42
IDASetMaxNumSteps, 32
IDASetMaxNumStepsIC, 41
IDASetMaxOrd, 32
IDASetMaxStep, 33
IDASetNoInactiveRootWarn, 43
IDASetNonlinConvCoef, 34
IDASetNonlinConvCoefIC, 41
IDASetRootDirection, 43
IDASetStepToleranceIC, 43
IDASetStopTime, 33
IDASetSuppressAlg, 34
IDASetUserData, 31
IDASLS_ILL_INPUT, 25, 37, 38
IDASLS_LMEM_NULL, 37, 54
IDASLS_MEM_FAIL, 25, 37
IDASLS_MEM_NULL, 25, 37, 38, 54
IDASLS_PACKAGE_FAIL, 25
IDASLS_SUCCESS, 25, 37, 38, 54

IDASlsGetLastFlag, 54
IDASlsGetNumJacEvals, 54
IDASlsGetReturnFlagName, 54
IDASlsSetSparseJacFn, 36
IDASlsSparseJacFn, 63
IDASolve, 19, 28
IDASPARSE linear solver
  use in FIDA, 79
IDASPBCG linear solver
  Jacobian approximation used by, 38
  memory requirements, 55
  optional input, 38–41
  optional output, 55–57
  preconditioner setup function, 38, 65
  preconditioner solve function, 38, 64
  selection of, 26
  use in FIDA, 80
IDASpbcg, 19, 23, 26
IDASPGMR linear solver
  Jacobian approximation used by, 38
  memory requirements, 55
  optional input, 38–41
  optional output, 55–57
  preconditioner setup function, 38, 65
  preconditioner solve function, 38, 64
  selection of, 25
  use in FIDA, 80
IDASpgmr, 19, 23, 26
IDASPILS_ILL_INPUT, 39, 40, 69
IDASPILS_LMEM_NULL, 39–41, 55–57, 69, 70
IDASPILS_MEM_FAIL, 26, 69
IDASPILS_MEM_NULL, 26, 39–41, 55–57
IDASPILS_PMEM_NULL, 70, 71
IDASPILS_SUCCESS, 26, 39–41, 57
IDASpilsGetLastFlag, 57
IDASpilsGetNumConvFails, 55
IDASpilsGetNumJtimesEvals, 56
IDASpilsGetNumLinIters, 55
IDASpilsGetNumPrecEvals, 56
IDASpilsGetNumPrecSolves, 56
IDASpilsGetNumResEvals, 56
IDASpilsGetReturnFlagName, 57
IDASpilsGetWorkSpace, 55
IDASpilsJacTimesVecFn, 64
IDASpilsPrecSetupFn, 65
IDASpilsPrecSolveFn, 64
IDASpilsSetEpsLin, 40
IDASpilsSetGSType, 39
IDASpilsSetIncrementFactor, 40
IDASpilsSetJacTimesFn, 39
IDASpilsSetMaxl, 41
IDASpilsSetMaxRestarts, 40
IDASpilsSetPreconditioner, 39
IDASPTFQMR linear solver
  Jacobian approximation used by, 38
  memory requirements, 55
  optional input, 38–41
  optional output, 55–57
  preconditioner setup function, 38, 65
  preconditioner solve function, 38, 64
  selection of, 26
  use in FIDA, 81
IDASptfqmr, 19, 23, 26
IDASStolerances, 21
IDASUPERLUMT linear solver
  Jacobian approximation used by, 36
  matrix reordering algorithm specification, 37
  NVECTOR compatibility, 25
  optional input, 36–38
  optional output, 54
  selection of, 25
IDASuperLUMT, 19, 23, 25, 63
IDASuperLUMTSetOrdering, 38
IDASVtolerances, 21
IDAWFtolerances, 21
INIT_STEP, 85
IOUT, 84, 86
itask, 28

Jacobian approximation function
  band
    difference quotient, 36
    use in FIDA, 78
    user-supplied, 36, 61–63
  dense
    difference quotient, 35
    use in FIDA, 78
    user-supplied, 35, 60–61
  Jacobian times vector
    difference quotient, 38
    use in FIDA, 81
    user-supplied, 39, 64
  sparse
    user-supplied, 36, 63–64

KLU sparse linear solver
  type SlsMat, 120

LS_OFF_IC, 85

MAX_CONVFAIL, 85
MAX_ERRFAIL, 85
MAX_NITERS, 85
MAX_NITERS_IC, 85
MAX_NJE_IC, 85
MAX_NSTEPS, 85
MAX_NSTEPS_IC, 85
MAX_ORD, 85
MAX_STEP, 85

maxl, 26
maxord, 58
memory requirements
IDA solver, 46
IDABAND linear solver, 52
IDABBDPRE preconditioner, 70
IDADENSE linear solver, 52
IDASPGMR linear solver, 55
MODIFIED_GS, 39
MPI, 4

N_VCloneEmptyVectorArray, 92
N_VCloneEmptyVectorArray_OpenMP, 101
N_VCloneEmptyVectorArray_Parallel, 99
N_VCloneEmptyVectorArray_Pthreads, 103
N_VCloneEmptyVectorArray_Serial, 96
N_VCloneVectorArray, 92
N_VCloneVectorArray_OpenMP, 101
N_VCloneVectorArray_Parallel, 99
N_VCloneVectorArray_Pthreads, 103
N_VCloneVectorArray_Serial, 96
N_VDestroyVectorArray, 92
N_VDestroyVectorArray_OpenMP, 101
N_VDestroyVectorArray_Parallel, 99
N_VDestroyVectorArray_Pthreads, 103
N_VDestroyVectorArray_Serial, 97
N_Vector, 16, 91
N_VMake_OpenMP, 101
N_VMake_Parallel, 98
N_VMake_Pthreads, 103
N_VMake_Serial, 96
N_VNew_OpenMP, 100
N_VNew_Parallel, 98
N_VNew_Pthreads, 102
N_VNew_Serial, 96
N_VNewEmpty_OpenMP, 101
N_VNewEmpty_Parallel, 98
N_VNewEmpty_Pthreads, 103
N_VNewEmpty_Serial, 96
N_VPrint_OpenMP, 101
N_VPrint_Parallel, 99
N_VPrint_Pthreads, 103
N_VPrint_Serial, 97
newBandMat, 118
newDenseMat, 116
newIntArray, 116, 119
newLintArray, 116, 119
newRealArray, 116, 119
NewSparseMat, 121
NLCONV_COEF, 85
NLCONV_COEF_IC, 85
NV_COMM_P, 98
NV_CONTENT_OMP, 100
NV_CONTENT_P, 97
NV_CONTENT_PT, 102
NV_CONTENT_S, 95
NV_DATA_OMP, 100
NV_DATA_P, 97
NV_DATA_PT, 102
NV_DATA_S, 96
NV_GLOBLENGTH_P, 97
NV_Ith_OMP, 100
NV_Ith_P, 98
NV_Ith_PT, 102
NV_Ith_S, 96
NV_LENGTH_OMP, 100
NV_LENGTH_PT, 102
NV_LENGTH_S, 96
NV_LOCLENGTH_P, 97
NV_NUM_THREADS_OMP, 100
NV_NUM_THREADS_PT, 102
NV_OWN_DATA_OMP, 100
NV_OWN_DATA_P, 97
NV_OWN_DATA_PT, 102
NV_OWN_DATA_S, 96
NVECTOR module, 91
nvector_openmp.h, 16
nvector_parallel.h, 16
nvector_pthreads.h, 16
nvector_serial.h, 16

openMP, 4
optional input
band linear solver, 35–36
dense linear solver, 35–36
initial condition calculation, 41–43
iterative linear solver, 38–41
rootfinding, 43
solver, 31–35
sparse linear solver, 36–38
optional output
band linear solver, 52–54
band-block-diagonal preconditioner, 70–71
dense linear solver, 52–54
initial condition calculation, 51
interpolated solution, 44
iterative linear solver, 55–57
solver, 44–51
sparse linear solver, 54

portability, 16
Fortran, 73
preconditioning
advice on, 9, 14
band-block diagonal, 66
setup and solve phases, 14
user-supplied, 38–39, 64, 65
PrintSparseMat, 123

Pthreads, 4

RCONST, 16
ReallocSparseMat, 123
realtype, 16
reinitialization, 57
residual function, 58
Rootfinding, 9, 19, 27, 87
ROUT, 84, 86

ScaleSparseMat, 121
SLS sparse linear solver
    functions
        small matrix, 121–123
SlsAddMat, 121
SlsConvertDls, 121
SlsMat, 120
SlsMatvec, 123
SMALL_REAL, 16
SPBCG generic linear solver
    description of, 125
    functions, 125
SPFGMR generic linear solver
    description of, 125
    functions, 125
SPGMR generic linear solver
    description of, 124
    functions, 124
    support functions, 124
SPTFQMR generic linear solver
    description of, 125
    functions, 125
step size bounds, 32–33
STEP_TOL_IC, 85
STOP_TIME, 85
sundials_nvector.h, 16
sundials_types.h, 16
SUPERLUMT sparse linear solver
    type SlsMat, 120
SUPPRESS_ALG, 85

TFQMR method, 41, 125
tolerances, 6, 22, 59

UNIT_ROUNDOFF, 16
User main program
    FIDA usage, 75
    FIDABBD usage, 88
    IDA usage, 17
    IDABBDPRE usage, 68
user_data, 31, 59, 60, 67, 68

weighted root-mean-square norm, 6