User Documentation for IDA v2.3.0

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

April 2005

UCRL-SM-208112

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

Contents

List of Tables

List of Figures

Chapter 1

Introduction

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

IDA is a general purpose solver for the initial value problem for systems of differential-algebraic equations (DAEs). The name IDA stands for Implicit Differential-Algebraic solver. IDA is based on DASPK [3, 4], but is written in ANSI-standard C rather than Fortran 77. Its most notable feature is that, in the solution of the underlying nonlinear system at each time step, it offers a choice of Newton/direct methods or an Inexact Newton/Krylov (iterative) method. Thus IDA shares significant modules previously written within CASC at LLNL to support the ordinary differential equation (ODE) solvers CVODE [11, 8] and PVODE [6, 7], and also the nonlinear system solver KINSOL [9].

The Newton/Krylov method uses the GMRES (Generalized Minimal RESidual) linear iterative method [13], and requires almost no matrix storage for solving the Newton equations as compared to direct methods. However, the GMRES algorithm allows for a user-supplied preconditioner matrix, and for most problems preconditioning is essential for an efficient solution.

There are several motivations for choosing the C language for IDA. First, a general movement away from FORTRAN and toward C in scientific computing is apparent. Second, the pointer, structure, and dynamic memory allocation features in C are extremely useful in software of this complexity, with the great variety of method options offered. Finally, we prefer C over C++ for 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 v2.3.0

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

Changes in v2.2.0

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 §5.4.5 and §5.4.7.

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

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

Changes in v2.2.1

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

1.2 Reading this User Guide

The structure of this document is as follows:

- In Chapter 2 we begin with instructions for the installation of IDA, within the structure of SUNDIALS.
- In Chapter 3, we give short descriptions of the numerical methods implemented by IDA for the solution of initial value problems for systems of DAEs.
- The following chapter describes the structure of the SUNDIALS suite of solvers (§4.1) and the software organization of the IDA solver (§4.2).
- In Chapter 5, we give an overview of the usage of IDA, as well as a complete description of the user interface and of the user-defined routines for integration of IVP DAEs.
- Chapter 6 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, as well as details on the two NVECTOR implementations provided with SUNDIALS: a serial implementation (§6.1) and a parallel MPI implementation (§6.2).
- Chapter 8 describes in detail the generic linear solvers shared by all SUNDIALS solvers.
- Finally, Chapter 9 lists the constants used for input to and output from IDA.

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

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

Chapter 2

IDA Installation Procedure

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

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

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

from within the `sundials` directory.

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

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

```
% rm -rf sundials
```

from a shell command prompt.

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

2.1 Installation steps

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

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

2.2 Configuration options

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

General options

`--prefix=PREFIX`

Location for architecture-independent files.

Default: `PREFIX=`*build_tree*

`--includedir=DIR`

Alternate location for installation of header files.

Default: `DIR=PREFIX/include`

`--libdir=DIR`

Alternate location for installation of libraries.

Default: `DIR=PREFIX/lib`

`--disable-examples`

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

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

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

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

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

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

Table 2.1: SUNDIALS libraries and header files

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

`--disable-`*solver*

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

`--with-cppflags=ARG`

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

`--with-cflags=ARG`

Specify additional C compilation flags.

`--with-ldflags=ARG`

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

`--with-libs=ARG`

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

`--with-precision=ARG`

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

Default: `ARG=double`

Options for Fortran support

`--disable-f77`

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

`--with-fflags=ARG`

Specify additional FORTRAN compilation flags.

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

`--with-f77underscore=ARG`

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

Default: `ARG=one`

`--with-f77case=ARG`

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

Default: `ARG=lower`

Options for MPI support

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

`--disable-mpi`

Using this option will completely disable MPI support.

`--with-mpicc=ARG`

`--with-mpif77=ARG`

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

`--with-mpi-root=MPIDIR`

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

`--with-mpi-incdir=INCDIR`

`--with-mpi-libdir=LIBDIR`

`--with-mpi-libs=LIBS`

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

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

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

`--with-mpi-flags=ARG`

Specify additional MPI-specific flags.

Options for library support

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

`--enable-shared`

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

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

Options for cross-compilation

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

`--build=BUILD`

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

`--host=HOST`

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

Environment variables

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

`CC`

`F77`

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

`CFLAGS`

`FFLAGS`

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

2.3 Configuration examples

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

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

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

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

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

Chapter 3

Mathematical Considerations

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

$$F(t, y, y') = 0 \,, \quad y(t_0) = y_0 \,,\; y'(t_0) = y'_0 \,, \tag{3.1}$$

where y, y', and F are vectors in $\mathbf{R}^N$, t is the independent variable, $y' = dy/dt$, and initial conditions $y(t_0) = y_0$, $y'(t_0) = y'_0$ are given. (Often t is time, but it certainly need not be.)

Prior to integrating a DAE initial-value problem, an important requirement is that the pair of vectors y_0 and y'_0 are both initialized to satisfy the DAE residual $F(t_0, y_0, y'_0) = 0$. 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 [4]. 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 y'_d but not on any components of y'_a. The assumption that the system is "index one" means that for a given t and y_d, the system $F(t, y, y') = 0$ defines y_a uniquely. In this case, a solver within IDA computes y_a and 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 $y'(t_0)$; this is intended mainly for quasi-steady-state problems, where $y'(t_0) = 0$ is given. In both cases, IDA solves the system $F(t_0, y_0, y'_0) = 0$ for the unknown components of y_0 and 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 in IDA is variable-order, variable-coefficient BDF, in fixed-leading-coefficient form [1]. The method order ranges from 1 to 5, with the BDF of order q given by the multistep formula

$$\sum_{i=0}^{q} \alpha_{n,i} y_{n-i} = h_n y'_n \,, \tag{3.2}$$

where y_n and y'_n are the computed approximations to $y(t_n)$ and $y'(t_n)$, 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 (3.2) to the DAE system (3.1) results in a nonlinear algebraic system to be solved at each step:

$$G(y_n) \equiv F\left(t_n,\, y_n,\, h_n^{-1} \sum_{i=0}^{q} \alpha_{n,i} y_{n-i}\right) = 0 \,. \tag{3.3}$$

Regardless of the method options, the solution of the nonlinear system (3.3) is accomplished with some form of Newton iteration. This leads to a linear system for each Newton correction, of the form

$$J[y_{n(m+1)} - y_{n(m)}] = -G(y_{n(m)}) \,, \tag{3.4}$$

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

$$J = \frac{\partial G}{\partial y} = \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial y'}\,, \tag{3.5}$$

where $\alpha = \alpha_{n,0}/h_n$. The scalar α changes whenever the step size or method order changes. The linear systems are solved by one of three methods:

- direct dense solve (serial version only),
- direct banded solve (serial version only), or
- SPGMR = Scaled Preconditioned GMRES, with restarts allowed.

For the SPGMR case, preconditioning is allowed only on the left,[1] so that GMRES is applied to systems $(P^{-1}J)\Delta y = -P^{-1}G$.

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

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

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

In the case of a direct linear solver (dense or banded), the nonlinear iteration (3.4) 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 SPGMR 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 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.

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

$$S\|\delta_m\| < 0.33\,, \tag{3.7}$$

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 $\alpha \ne \bar{\alpha}$. Note that at $m = 1$, the convergence test (3.7) 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

[1] 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.

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 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 (3.5) can be either supplied by the user or have IDA compute one internally by difference quotients. In the latter case, we use the approximation

$$J_{ij} = [F_i(t, y+\sigma_j e_j, y' + \alpha\sigma_j e_j) - F_i(t,y,y')]/\sigma_j \,, \text{ with}$$
$$\sigma_j = \sqrt{U}\max\left\{|y_j|, |hy'_j|, W_j\right\}\operatorname{sign}(hy'_j)\,,$$

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 (3.6). In the SPGMR case, if a routine for Jv is not supplied, such products are approximated by

$$Jv = [F(t, y+\sigma v, y' + \alpha\sigma v) - F(t,y,y')]/\sigma\,,$$

where the increment σ is $1/\|v\|$.[2] 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

$$\|\mathrm{LTE}\|_{\mathrm{WRMS}} \leq 1\,.$$

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

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

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

$$\max\{|C|, \bar{C}\}\|\Delta_n\| \leq 1\,. \tag{3.8}$$

A user option is available by which the algebraic components of the error vector are omitted from the test (3.8), 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 (3.8) 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

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

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 $\mathrm{ELTE}(q') = C(q')\|\phi(q'+1)\|$

[2] All vectors v occurring here have been divided by the weights W_i and then scaled so as to have L_2 norm equal to 1. Thus, in fact $\sigma = 1/\|v\|_{\mathrm{WRMS}} = \sqrt{N}$.

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

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

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 (3.8) 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 ELTE(q), and, with safety factors, is given by

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

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:

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

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 [1] 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 $t = t_{\mathrm{out}}$ is overtaken, and then computes $y(t_{\mathrm{out}})$ 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 $t = t_{\mathrm{stop}}$.

Chapter 4

Code Organization

4.1 SUNDIALS organization

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

The various solvers of this family share many subordinate modules. For this reason, it is organized as a family, with a directory structure that exploits that sharing (see Fig. 4.1). The following is a list of the solver packages presently available:

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

4.2 IDA organization

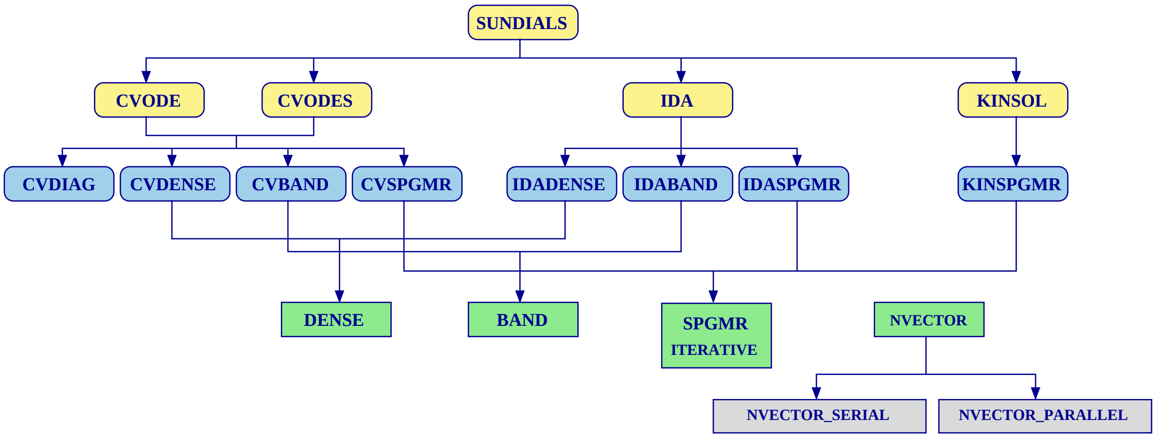
The IDA package is written in the ANSI C language. The following summarizes the basic structure of the package, although knowledge of this structure is not necessary for its use.

The overall organization of the IDA package is shown in Figure 4.2. The central integration module, implemented in the files `ida.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.

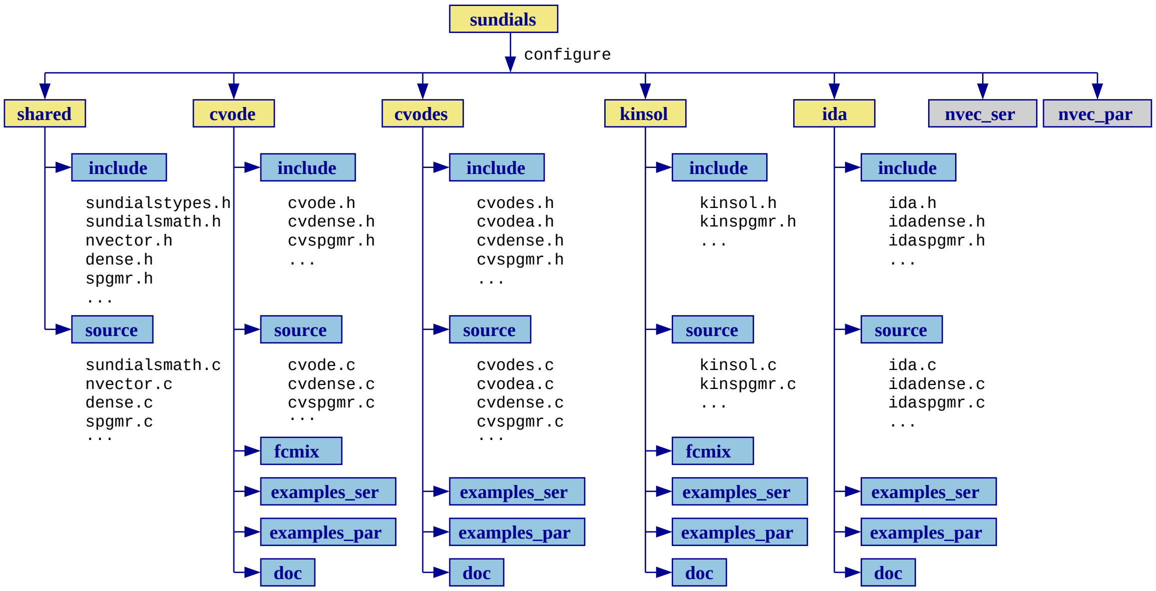
At present, the package includes the following three IDA linear system modules:

- IDADENSE: LU factorization and backsolving with dense matrices;
- IDABAND: LU factorization and backsolving with banded matrices;
- IDASPGMR: scaled preconditioned GMRES method.

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



(a) High-level diagram



(b) Directory structure

Figure 4.1: Organization of the SUNDIALS suite

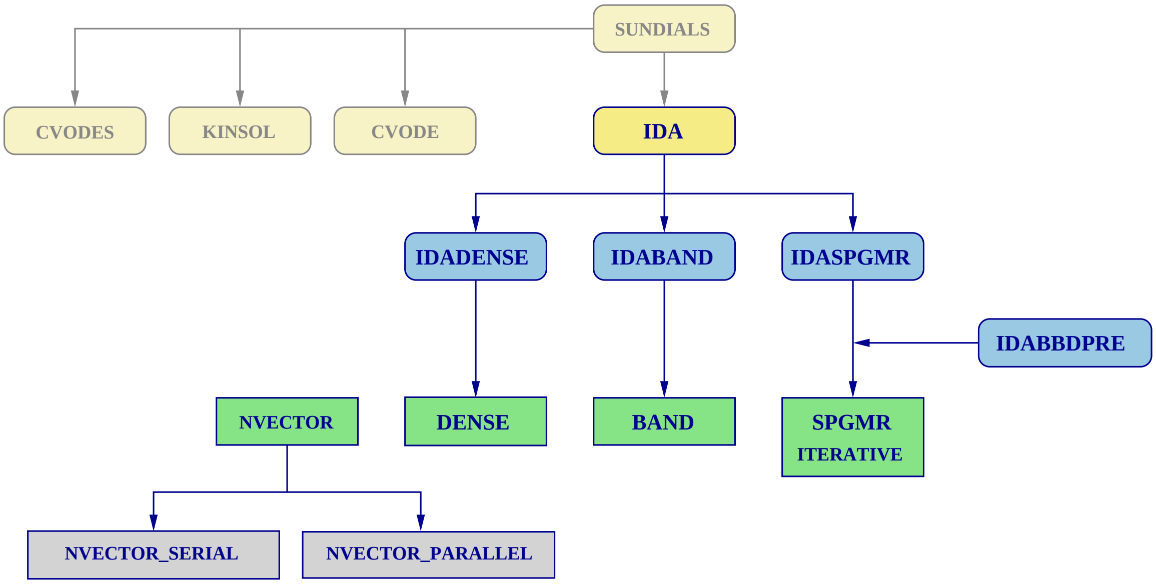


Figure 4.2: Overall structure diagram of the IDA package. Modules specific to IDA are distinguished by rounded boxes, while generic solver and auxiliary modules are in square boxes.

In the case of the direct IDADENSE and IDABAND methods, the package includes an algorithm for the approximation of the Jacobian by difference quotients, but the user also has the option of supplying the Jacobian (or an approximation to it) directly. In the case of the iterative IDASPGMR method, 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. In the case of IDASPGMR, the preconditioning must be supplied by the user in two phases: setup (preprocessing of Jacobian data) and solve. While there is no default choice of preconditioner analogous to the difference quotient approximation in the direct case, the references [2]-[5], 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, IDABAND, and IDASPGMR is a set of interface routines built on top of a generic solver module, named DENSE, BAND, and SPGMR, respectively. The interfaces deal with the use of these methods in the IDA context, whereas the generic solver is independent of the context. While the generic solvers here were generated with SUNDIALS in mind, our intention is that they be usable in other applications as general-purpose solvers. This separation also allows for any generic solver to be replaced by an improved version, with no necessity to revise the 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 5

Using IDA

This chapter is concerned with the use of IDA for the integration of DAEs. 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 listings of the sample programs in the companion document [10] 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.

The user should be aware that not all linear solver modules are compatible with all NVECTOR implementations. For example, NVECTOR_PARALLEL is not compatible with the direct dense or direct band linear solvers, since these linear solver modules need to form the complete system Jacobian. The IDADENSE and IDABAND modules can only be used with NVECTOR_SERIAL. 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 Chapter 9.

5.1 Data types

The `sundialstypes.h` file contains the definition of the type `realtype`, which is used by the SUNDIALS solvers for all floating-point data. The type `realtype` can be `float`, `double`, or `long double`, with the default being `double`. The user can change the precision of the SUNDIALS solvers arithmetic at the configuration stage (see §2.2).

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

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

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

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

A user program which uses the type `realtype` and the `RCONST` macro to handle floating-point constants is precision-independent except for any calls to precision-specific standard math library functions. (Our example programs use both `realtype` and `RCONST`.) Users can, however, use the type `double`, `float`, or `long double` in their code (assuming the typedef for `realtype` matches this choice). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use `realtype`, so long as the SUNDIALS libraries use the correct precision (for details see §2.2).

5.2 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 `sundialstypes.h`, which defines the types `realtype` and `booleantype` and the constants `FALSE` and `TRUE`.

The calling program must also include an NVECTOR implementation header file (see Chapter 6 for details). For the two 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.

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

- `idadense.h`, which is used with the dense direct linear solver in the context of IDA. This in turn includes a header file (`dense.h`) which defines the `DenseMat` type and corresponding accessor macros;
- `idaband.h`, which is used with the band direct linear solver in the context of IDA. This in turn includes a header file (`band.h`) which defines the `BandMat` type and corresponding accessor macros;
- `idaspgmr.h`, which is used with the Krylov solver SPGMR in the context of IDA. This in turn includes a header file (`iterative.h`) which enumerates the kind of preconditioning and the choices for the Gram-Schmidt process.

5.3 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 with [**P**] correspond to NVECTOR_PARALLEL, while steps marked with [**S**] correspond to NVECTOR_SERIAL.

1. **[P] Initialize MPI**

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

2. **Set problem dimensions**

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

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

3. **Set vector of initial values**

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

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

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

   Otherwise, make the call:

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

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

   and load initial values into the structure defined by:

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

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

   Here `comm` is the MPI communicator, set in one of two ways: If a proper subset of active 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 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 §5.4.1 for details.

5. **Allocate internal memory**

   Call `IDAMalloc(...);` to provide required problem specifications, allocate internal memory for IDA, and initialize IDA. `IDAMalloc` returns an error flag to indicate success or an illegal argument value. See §5.4.1 for details.

6. **Set optional inputs**

   Call `IDASet*` functions to change from their default values any optional inputs that control the behavior of IDA. See §5.4.5 for details.

7. **Attach linear solver module**

   Initialize the linear solver module with one of the following calls (for details see §5.4.2):

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

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

   `flag = IDASpgmr(...);`

8. **Set linear solver optional inputs**

   Call `IDA*Set*` functions from the selected linear solver module to change optional inputs specific to that linear solver. See §5.4.5 for details.

9. **Correct initial values**

   Optionally, call `IDACalcIC` to correct the initial values `y0` and `yp0`.

10. **Advance solution in time**

    For each point at which output is desired, call `flag = IDASolve(ida_mem, tout, &tret, yret, ypret, itask);` Set `itask` to specify 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 $y'(t)$.

See §5.4.4 for details.

11. **Get optional outputs**

    Call `IDA*Get*` functions to obtain optional output. See §5.4.7 for details.

12. **Deallocate memory for solution vector**

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

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

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

    and similarly for `ypret`.

13. **Free solver memory**

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

14. [**P**] **Finalize MPI**

    Call `MPI_Finalize();` to terminate MPI.

5.4 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 §5.4.5, 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 §5.3 for the correct order of these calls.

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

`IDAMalloc`

Call `flag = IDAMalloc(ida_mem, res, t0, y0, yp0, itol, reltol, abstol);`

Description The function `IDAMalloc` 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 F in the DAE. This function has the form `res(t, yy, yp, resval, res_data)` (for full details see §5.5).
- `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 y'.
itol	(int) is one of IDA_SS, IDA_SV, or IDA_WF, where itol=IDA_SS indicates scalar relative error tolerance and scalar absolute error tolerance, while itol=IDA_SV indicates scalar relative error tolerance and vector absolute error tolerance. The latter choice is important when the absolute error tolerance needs to be different for each component of the DAE. If itol=IDA_WF, the arguments reltol and abstol are ignored and the user is expected to provide a function to evaluate the error weight vector W from (3.6). See IDASetEwtFn in §5.4.5.
reltol	(realtype) is the relative error tolerance.
abstol	(void *) is a pointer to the absolute error tolerance. If itol=IDA_SS, abstol must be a pointer to a realtype variable. If itol=IDA_SV, abstol must be an N_Vector variable.

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

IDA_SUCCESS	The call to IDAMalloc 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 IDAMalloc has an illegal value.

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

The tolerance values in reltol and abstol may be changed between calls to IDASolve (see IDASetTolerances in §5.4.5).

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

IDAFree

Call IDAFree(ida_mem);

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

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

Return value The function IDAFree has no return value.

5.4.2 Linear solver specification functions

As previously explained, Newton iteration requires the solution of linear systems of the form (3.4). There are three IDA linear solvers currently available for this task: IDADENSE, IDABAND, and IDASPGMR. The first two are direct solvers and derive their name from the type of approximation used for the Jacobian $J = \partial F/\partial y + c_j \partial F/\partial y'$. IDADENSE and IDABAND work with dense and banded approximations to J, respectively. The third IDA linear solver, IDASPGMR, is an iterative solver. The SPGMR in the name indicates that it uses a scaled preconditioned GMRES method.

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, IDABand, IDASpgmr, 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, and SPGMR, are described separately in Chapter 8.

IDADense

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

Description The function `IDADense` selects the IDADENSE linear solver.

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

`IDADENSE_SUCCESS`	The IDADENSE initialization was successful.
`IDADENSE_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDADENSE_ILL_INPUT`	The IDADENSE solver is not compatible with the current NVECTOR module.
`IDADENSE_MEM_FAIL`	A memory allocation request failed.

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

IDABand

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

Description The function `IDABand` selects the IDABAND linear solver.

The user's main function must include the `idaband.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 may not be compatible with a particular implementation of the NVECTOR module. Of the two NVECTOR modules provided by SUNDIALS, only NVECTOR_SERIAL is compatible, while NVECTOR_PARALLEL is not. The half-bandwidths are to be set so that the nonzero locations (i, j) in the banded (approximate) Jacobian satisfy $-\texttt{mlower} \leq j - i \leq \texttt{mupper}$.

IDASpgmr

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

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

The user's main function must include the `idaspgmr.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_SPGMR_MAXL`$= 5$.

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

`IDASPGMR_SUCCESS`	The IDASPGMR initialization was successful.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_MEM_FAIL`	A memory allocation request failed.

5.4.3 Initial condition calculation function

`IDACalcIC` calculates corrected initial conditions for the DAE system for a class of index-one problems of semi-implicit form. It uses Newton iteration combined with a linesearch algorithm. Calling `IDACalcIC` is optional. It is only necessary when the initial conditions do not solve the given system; i.e., if `y0` and `yp0` are known to satisfy $F(t_0, y_0, y'_0) = 0$, then a call to `IDACalcIC` is *not* necessary.

A call to `IDACalcIC` must be preceded by successful calls to `IDACreate` and `IDAMalloc`, 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, t0, y0, yp0, 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.
- `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 y'.
- `icopt` (`int`) is the option of IDACalcIC to be used.

  `icopt`=`IDA_YA_YDP_INIT` directs `IDACalcIC` to compute the algebraic components of y and differential components of 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 y'. `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 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 `IDAMalloc` 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 is typically called before the first call to IDASolve and after IDAMalloc to compute consistent initial conditions for the DAE problem. However, it can be also called at any other further time to correct a pair (y, y').

5.4.4 IDA solver function

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

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.
yret	(N_Vector) the computed solution vector y.
ypret	(N_Vector) the computed solution vector 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 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. The IDA_NORMAL_TSTOP and IDA_ONE_STEP_TSTOP modes are similar to IDA_NORMAL and IDA_ONE_STEP, respectively, except that the integration never proceeds past the value tstop (specified through the function IDASetStopTime).

Return value On return, IDASolve returns vectors yret and ypret and a corresponding independent variable value $t =$*tret, such that (yret, ypret) are the computed values of $(y(t), y'(t))$.

In NORMAL mode with no errors, *tret will be equal to tout and yret $= y$(tout), ypret $= y'$(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 (see §5.4.5).
IDA_MEM_NULL	The ida_mem argument was NULL.
IDA_ILL_INPUT	One of the inputs to IDASolve is illegal. This includes the situation when a component of the error weight vectors becomes negative during internal time-stepping. The IDA_ILL_INPUT flag will also be returned if the linear solver function initialization

	(called by the user after calling `IDACreate`) failed to set the linear solver-specific `lsolve` field in `ida_mem`. In any case, the user should see the printed error message for more 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.

Notes The vector `yret` can occupy the same space as the `y0` vector of initial conditions that was passed to `IDAMalloc`, while the vector `ypret` can occupy the same space as the `yp0`.

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

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

5.4.5 Optional input functions

IDA provides an extensive list of functions that can be used to change various optional input parameters that control the behavior of the IDA solver from their default values. Table 5.1 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 §5.5.

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

Main solver optional input functions

The calls listed here can be executed in any order. However, if `IDASetErrFile` is to be called, that call should be 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.

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

`errfp` (`FILE *`) pointer to output file.

Table 5.1: Optional inputs for IDA, IDADENSE, IDABAND, and IDASPGMR

Optional input	Function name	Default
IDA main solver		
Pointer to an error file	`IDASetErrFile`	`stderr`
Data for residual function	`IDASetRdata`	`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`
Integration tolerances	`IDASetTolerances`	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`	(2/3)uround
IDADENSE linear solver		
Dense Jacobian function and data	`IDADenseSetJacFn`	internal DQ, NULL
IDABAND linear solver		
Band Jacobian function and data	`IDABandSetJacFn`	internal DQ, NULL
IDASPGMR linear solver		
Preconditioner functions and data	`IDASpgmrSetPreconditioner`	NULL, NULL, NULL
Jacobian times vector function and data	`IDASpgmrSetJacTimesVecFn`	internal DQ, NULL
Type of Gram-Schmidt orthogonalization	`IDASpgmrSetGSType`	classical GS
Maximum no. of restarts	`IDASpgmrSetMaxRestarts`	5
Factor in linear convergence test	`IDASpgmrSetEpsLin`	0.05
Factor in DQ increment calculation	`IDASpgmrSetIncrementFactor`	1.0

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 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`).

! 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.

`IDASetRdata`

Call `flag = IDASetRdata(ida_mem, res_data);`

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

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

`res_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 `res_data` is not specified, a `NULL` pointer is passed to all user functions that have it as an argument.

`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.

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 specified value `maxord` is negative, or larger than its previous value.

Notes The default value is 5. Since `maxord` affects the memory requirements for the internal IDA memory block, its value can not 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`.

	IDA_ILL_INPUT mxsteps is non-positive.
Notes	Passing mxsteps= 0 results in IDA using the default value (500).

`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.

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 $\|hy'\|_{\mathrm{WRMS}} = 1/2$, 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.

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.
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.
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.
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 3, Eq. (3.7).
Arguments	`ida_mem` (`void *`) pointer to the IDA memory block. `nlscoef` (`realtype`) coefficient in nonlinear convergence 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 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.
- `suppresslag` (`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 `suppresslag=TRUE` is selected, then the `id` vector must be set (through `IDASetId`) to specify the algebraic components.

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 §5.4.3).

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 successfuly 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.

`IDASetTolerances`

Call `flag = IDASetTolerances(ida_mem, itol, reltol, abstol);`

Description The function `IDASetTolerances` resets the integration tolerances.

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

Return value The return value `flag` (of type `int`) is one of
- `IDA_SUCCESS` The tolerances have been successfully set.
- `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
- `IDA_ILL_INPUT` An input argument has an illegal value.

Notes The integration tolerances are initially specified in the call to `IDAMalloc` (see §5.4.1). This function call to `IDASetTolerances` is needed only if the tolerances are being changed from their values between successive calls to `IDASolve`.

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

It is illegal to call `IDASetTolerances` before a call to `IDAMalloc`.

`IDASetEwtFn`

Call `flag = IDASetEwtFn(ida_mem, efun, edata);`

Description The function `IDASetEwtFn` specifies the user-defined function to be used in computing the error weight vector W in (3.6).

Arguments
- `ida_mem` (`void *`) pointer to the IDA memory block.
- `efun` (`IDAEwtFn`) is the C function which defines the `ewt` vector (see §5.5.2).
- `edata` (`void *`) pointer to user data passed to `efun` every time it is called.

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

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

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

It is illegal to call `IDASetEwtFn` before a call to `IDAMalloc`.

Linear solver optional input functions

The linear solver modules allow for various optional inputs, which are described here.

Dense Linear solver. The IDADENSE solver needs a function to compute a dense approximation to the Jacobian matrix $J(t, y, y')$. This function must be of type `IDADenseJacFn`. The user can supply his/her own dense Jacobian function, or use the default difference quotient function `IDADenseDQJac` that comes with the IDADENSE solver. To specify a user-supplied Jacobian function `djac` and associated user data `jac_data`, IDADENSE provides the function `IDADenseSetJacFn`. The IDADENSE solver passes the pointer `jac_data` to its dense Jacobian function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `jac_data` may be identical to `f_data`, if the latter was specified through `IDASetFdata`.

`IDADenseSetJacFn`

Call `flag = IDADenseSetJacFn(ida_mem, djac, jac_data);`

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

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

Return value The return value `flag` (of type `int`) is one of
- `IDADENSE_SUCCESS` The optional value has been successfully set.
- `IDADENSE_MEM_NULL` The `ida_mem` pointer is `NULL`.
- `IDADENSE_LMEM_NULL` The IDADENSE linear solver has not been initialized.

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

The function type `IDADenseJacFn` is described in §5.5.3.

Band Linear solver. The IDABAND solver needs a function to compute a banded approximation to the Jacobian matrix $J(t, y, y')$. This function must be of type `IDABandJacFn`. The user can supply his/her own banded Jacobian approximation function, or use the default difference quotient function `IDABandDQJac` that comes with the IDABAND solver. To specify a user-supplied Jacobian function `bjac` and associated user data `jac_data`, IDABAND provides the function `IDABandSetJacFn`. The IDABAND solver passes the pointer `jac_data` to its banded Jacobian approximation function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `jac_data` may be identical to `f_data`, if the latter was specified through `IDAodeSetFdata`.

`IDABandSetJacFn`

Call `flag = IDABandSetJacFn(ida_mem, bjac, jac_data);`

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

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

Return value The return value `flag` (of type `int`) is one of
- `IDABAND_SUCCESS` The optional value has been successfully set.
- `IDABAND_MEM_NULL` The `ida_mem` pointer is `NULL`.
- `IDABAND_LMEM_NULL` The IDABAND linear solver has not been initialized.

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

The function type `IDABandJacFn` is described in §5.5.4.

SPGMR Linear solver. If preconditioning is to be done within the SPGMR method, then the user must supply a preconditioner solve function, `psolve`, and specify its name through a call to `IDASpgmrSetPreconditioner`. The evaluation and preprocessing of any Jacobian-related data needed by the user's preconditioner solve function is done in the optional user-supplied function `psetup`. Both of these functions are fully specified in §5.5. If used, the `psetup` function should also be specified in the call to `IDASpgmrSetPreconditioner`. Optionally, the IDASPGMR solver passes the pointer it receives through `IDASpgmrSetPreconditioner` to the preconditioner setup and solve functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program. The pointer `prec_data` may be identical to `res_data`, if the latter was specified through `IDASetRdata`.

The IDASPGMR solver requires a function to compute an approximation to the product between the Jacobian matrix $J(t, y)$ and a vector v. The user can supply his/her own Jacobian times vector approximation function, or use the difference quotient function `IDASpgmrDQJtimes` that comes with the IDASPGMR solver. A user-defined Jacobian-vector function must be of type `IDASpgmrJtimesFn` and can be specified through a call to `IDASpgmrSetJacTimesVecFn` (see §5.5 for specification details). As with the preconditioner user data structure `p_data`, the user can also specify int the call to `IDASpgmrSetJacFn`, a pointer to a user-defined data structure, `jac_data`, which the IDASPGMR solver passes to the Jacobian times vector function `jtimes` each time it is called. The pointer `jac_data` may be identical to `P_data` and/or `f_data`.

`IDASpgmrSetPreconditioner`

Call `flag = IDASpgmrSetPrecSolveFn(ida_mem, psolve, psetup, p_data);`

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

Arguments
- `ida_mem` (`void *`) pointer to the IDA memory block.
- `psolve` (`IDASpgmrPrecSolveFn`) user-defined preconditioner solve function.
- `psetup` (`IDASpgmrPrecSetupFn`) user-defined preconditioner setup function.
- `p_data` (`void *`) pointer to the user-defined data structure.

Return value The return value `flag` (of type `int`) is one of
- `IDASPGMR_SUCCESS` The optional value has been successfully set.
- `IDASPGMR_MEM_NULL` The `ida_mem` pointer is `NULL`.
- `IDASPGMR_LMEM_NULL` The IDASPGMR linear solver has not been initialized.

Notes The function type `IDASpgmrPrecSolveFn` is described in §5.5.6. The function type `IDASpgmrPrecSetupFn` is described in §5.5.7.

`IDASpgmrSetJacTimesVecFn`

Call `flag = IDASpgmrSetJacTimesVecFn(ida_mem, jtimes, jac_data);`

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

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

Return value The return value `flag` (of type `int`) is one of
- `IDASPGMR_SUCCESS` The optional value has been successfully set.
- `IDASPGMR_MEM_NULL` The `ida_mem` pointer is `NULL`.
- `IDASPGMR_LMEM_NULL` The IDASPGMR linear solver has not been initialized.

Notes — By default, IDASPGMR uses the difference quotient function `IDASpgmrDQJtimes`. If `NULL` is passed to `jtimes`, this default function is used.

The function type `IDASpgmrJacTimesVecFn` is described in §5.5.5.

`IDASpgmrSetGSType`

Call — `flag = IDASpgmrSetGSType(ida_mem, gstype);`

Description — The function `IDASpgmrSetGSType` 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

`IDASPGMR_SUCCESS`	The optional value has been successfuly set.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_LMEM_NULL`	The IDASPGMR linear solver has not been initialized.
`IDASPGMR_ILL_INPUT`	The Gram-Schmidt orthogonalization type `gstype` is not valid.

Notes — The default value is `MODIFIED_GS`.

`IDASpgmrSetMaxRestarts`

Call — `flag = IDASpgmrSetMaxRestarts(ida_mem, maxrs);`

Description — The function `IDASpgmrSetMaxRestarts` 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

`IDASPGMR_SUCCESS`	The optional value has been successfuly set.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_LMEM_NULL`	The IDASPGMR linear solver has not been initialized.
`IDASPGMR_ILL_INPUT`	The `maxrs` argument is negative.

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

`IDASpgmrSetEpsLin`

Call — `flag = IDASpgmrSetEpsLin(ida_mem, eplifac);`

Description — The function `IDASpgmrSetEpsLin` specifies the factor by which the GMRES convergence test constant is reduced from the Newton iteration test constant. (See §3).

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

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

`IDASPGMR_SUCCESS`	The optional value has been successfuly set.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_LMEM_NULL`	The IDASPGMR linear solver has not been initialized.
`IDASPGMR_ILL_INPUT`	The factor `eplifac` is negative.

Notes — The default value is 0.05.

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

`IDASpgmrSetIncrementFactor`

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

Description The function `IDASpgmrSetIncrementFactor` specifies a factor in the increments to y used in the difference quotient approximations to the Jacobian-vector products. (See §3).

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

`IDASPGMR_SUCCESS`	The optional value has been successfuly set.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_LMEM_NULL`	The IDASPGMR linear solver has not been initialized.
`IDASPGMR_ILL_INPUT`	The increment factor was non-positive.

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

Initial condition calculation optional input functions

The following functions can be called to set optional inputs to control the initial conditions calculations.

`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.

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

`IDA_SUCCESS`	The optional value has been successfuly set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT`	The `epiccon` factor is negative (illegal).

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 y' to be accepted, the norm of $J^{-1}F(t_0, y, y')$ 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, $J = \partial F/\partial y + (1/h)\partial F/\partial y'$.

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 successfuly 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 successfuly 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 successfuly 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 successfuly 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`) Newton step tolerance.

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

IDA_SUCCESS	The optional value has been successfuly set.
IDA_MEM_NULL	The `ida_mem` pointer is `NULL`.
IDA_ILL_INPUT	The `steptol` tolerance is negative (illegal).

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

5.4.6 Interpolated output function

An optional function `IDAGetSolution` is available to obtain additional output values. This function must be called after a successful return from `IDASolve` and provides interpolated values of y and y' for any value of t in the last internal step taken by IDA.

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

`IDAGetSolution`

Call `flag = IDAGetSolution(ida_mem, t, yret, ypret);`

Description The function `IDAGetSolution` computes the interpolated values of y and y' for any value of t in the last internal step taken by IDA. 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`)
yret	(`N_Vector`) vector containing the interpolated $y(t)$.
ypret	(`N_Vector`) vector containing the interpolated $y'(t)$.

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

IDA_SUCCESS	`IDAGetSolution` succeeded.
IDA_MEM_NULL	The `ida_mem` argument was `NULL`.
IDA_BAD_T	`t` is not in the interval $[t_n - h_u, t_n]$.

Notes It is only legal to call the function `IDAGetSolution` after a successful return from `IDASolve`. See `IDAGetCurrentTime` and `IDAGetLastStep` for access to t_n and h_u.

5.4.7 Optional output functions

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

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 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.

Table 5.2: Optional outputs from IDA, IDADENSE, IDABAND, and IDASPGMR

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`
No. of nonlinear solver iterations	`IDAGetNumNonlinSolvIters`
No. of nonlinear convergence failures	`IDAGetNumNonlinSolvConvFails`
IDADENSE linear solver	
Size of IDADENSE real and integer workspace	`IDADenseGetWorkSpace`
No. of Jacobian evaluations	`IDADenseGetNumJacEvals`
No. of residual calls for finite diff. Jacobian evals.	`IDADenseGetNumResEvals`
Last return from a IDADENSE function	`IDADenseGetLastFlag`
IDABAND linear solver	
Size of IDABAND real and integer workspace	`IDABandGetWorkSpace`
No. of Jacobian evaluations	`IDABandGetNumJacEvals`
No. of residual calls for finite diff. Jacobian evals.	`IDABandGetNumResEvals`
Last return from a IDABAND function	`IDABandGetLastFlag`
IDASPGMR linear solver	
Size of IDASPGMR real and integer workspace	`IDASpgmrGetWorkSpace`
No. of linear iterations	`IDASpgmrGetNumLinIters`
No. of linear convergence failures	`IDASpgmrGetNumConvFails`
No. of preconditioner evaluations	`IDASpgmrGetNumPrecEvals`
No. of preconditioner solves	`IDASpgmrGetNumPrecSolves`
No. of Jacobian-vector product evaluations	`IDASpgmrGetNumJtimesEvals`
No. of residual calls for finite diff. Jacobian-vector evals.	`IDASpgmrGetNumResEvals`
Last return from a IDASPGMR function	`IDASpgmrGetLastFlag`

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 successfuly set.

IDA_MEM_NULL The ida_mem pointer is NULL.

Notes In terms of the problem size N and maximum method order maxord, the actual size of the real workspace given in realtype words is:

- Base value: lenrw $= 55 + (m+6)N$
- With itol = IDA_SV: lenrw = lenrw $+N$
- With constraint checking (see IDASetConstraints): lenrw = lenrw $+N$
- With id specified (see IDASetId): lenrw = lenrw $+N$

Here $m = max(3,$maxord$)$.

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

- Base value: leniw $= 38 + (m+6)N$
- With itol = IDA_SV: leniw = leniw $+N$
- With constraint checking: lenrw = lenrw $+N$
- With id specified: lenrw = lenrw $+N$

For the default value of maxord, the base values are lenrw $= 55 + 11N$ and leniw $= 38 + 11N$

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 successfuly 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 successfuly 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 successfuly 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 successfuly set.
- `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetLastOrder`

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

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.
- `qlast` (`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 successfuly set.
- `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetCurrentOrder`

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

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.
- `qcur` (`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 successfuly 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.

Return value The return value `flag` (of type `int`) is one of
`IDA_SUCCESS` The optional output value has been successfuly 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 successfuly 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 successfuly 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 successfuly 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 successfuly 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 reciprocals of the W_i of (3.6).

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 successfuly set.

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

Notes The user must allocate space for `eweight`.

`IDAGetIntegratorStats`

Call
```
flag = IDAGetIntegratorStats(ida_mem, &nsteps, &nrevals, &nlinsetups,
                             &netfails, &qlast, &qcur, &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.

`qlast` (`int`) method order used on the last internal step.

`qcur` (`int`) method order to be used on the next internal step.

`hinused` (`realtype`) actual value of initial step size.

`hlast` (`realtype`) step size taken on the last internal step.

`hcur` (`realtype`) step size to be attempted on the next internal step.

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

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

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

Linear solver optional output functions

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

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

`IDADenseGetWorkSpace`

Call `flag = IDADenseGetWorkSpace(ida_mem, &lenrwD, &leniwD);`

Description The function `IDADenseGetWorkSpace` returns the sizes of the IDADENSE real and integer workspaces.

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

`lenrwD`	(`long int`) the number of real values in the IDADENSE workspace.
`leniwD`	(`long int`) the number of integer values in the IDADENSE workspace.

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

`IDA_SUCCESS`	The optional output value has been successfuly set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_MEM_NULL`	The IDADENSE linear solver has not been initialized.

Notes: In terms of the problem size N, the actual size of the real workspace is $2N^2$ `realtype` words.

In terms of the problem size N, the actual size of the integer workspace is N integer words.

`IDADenseGetNumJacEvals`

Call: `flag = IDADenseGetNumJacEvals(ida_mem, &njevalsD);`

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

Arguments:

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

`IDADENSE_SUCCESS`	The optional output value has been successfuly set.
`IDADENSE_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDADENSE_LMEM_NULL`	The IDADENSE linear solver has not been initialized.

`IDADenseGetNumResEvals`

Call: `flag = IDADenseGetNumResEvals(ida_mem, &nrevalsD);`

Description: The function `IDADenseGetNumResEvals` returns the cumulative number of calls to the user residual function due to the finite difference dense Jacobian approximation.

Arguments:

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

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

`IDADENSE_SUCCESS`	The optional output value has been successfuly set.
`IDADENSE_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDADENSE_LMEM_NULL`	The IDADENSE linear solver has not been initialized.

Notes: The value `nrevalsD` is incremented only if the default `IDADenseDQJac` difference quotient function is used.

`IDADenseGetLastFlag`

Call: `flag = IDADenseGetLastFlag(ida_mem, &flag);`

Description: The function `IDADenseGetLastFlag` returns the last return value from an IDADENSE routine.

Arguments:

`ida_mem`	(`void *`) pointer to the IDA memory block.
`flag`	(`int`) the value of the last return flag from an IDADENSE function.

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

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

`IDADENSE_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDADENSE_LMEM_NULL`	The IDADENSE linear solver has not been initialized.

Notes: If the IDADENSE setup function failed (`IDASolve` returned `IDA_LSETUP_FAIL`), the value `flag` is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the dense Jacobian matrix.

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

`IDABandGetWorkSpace`

Call: `flag = IDABandGetWorkSpace(ida_mem, &lenrwB, &leniwB);`

Description: The function `IDABandGetWorkSpace` returns the sizes of the IDABAND real and integer workspaces.

Arguments:
- `ida_mem` (`void *`) pointer to the IDA memory block.
- `lenrwB` (`long int`) the number of real values in the IDABAND workspace.
- `leniwB` (`long int`) the number of integer values in the IDABAND workspace.

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

`IDABAND_SUCCESS`	The optional output value has been successfuly set.
`IDABAND_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDABAND_LMEM_NULL`	The IDABAND linear solver has not been initialized.

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

In terms of the problem size N, the actual size of the integer workspace is N integer words.

`IDABandGetNumJacEvals`

Call: `flag = IDABandGetNumJacEvals(ida_mem, &njevalsB);`

Description: The function `IDABandGetNumJacEvals` returns the cumulative number of calls to the banded Jacobian approximation function.

Arguments:
- `ida_mem` (`void *`) pointer to the IDA memory block.
- `njevalsB` (`long int`) the cumulative number of calls to the Jacobian function.

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

`IDABAND_SUCCESS`	The optional output value has been successfuly set.
`IDABAND_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDABAND_LMEM_NULL`	The IDABAND linear solver has not been initialized.

`IDABandGetNumResEvals`

Call: `flag = IDABandGetNumResEvals(ida_mem, &nrevalsB);`

Description: The function `IDABandGetNumResEvals` returns the cumulative number of calls to the user residual function due to the finite difference banded Jacobian approximation.

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

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

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

IDABAND_MEM_NULL	The ida_mem pointer is NULL.
IDABAND_LMEM_NULL	The IDABAND linear solver has not been initialized.

Notes: The value nrevalsB is incremented only if the default IDABandDQJac difference quotient function is used.

IDABandGetLastFlag

Call: `flag = IDABandGetLastFlag(ida_mem, &flag);`

Description: The function IDABandGetLastFlag returns the last return value from an IDABAND routine.

Arguments:
- ida_mem (void *) pointer to the IDA memory block.
- flag (int) the value of the last return flag from an IDABAND function.

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

IDABAND_SUCCESS	The optional output value has been successfully set.
IDABAND_MEM_NULL	The ida_mem pointer is NULL.
IDABAND_LMEM_NULL	The IDABAND linear solver has not been initialized.

Notes: If the IDABAND setup function failed (IDASolve returned IDA_LSETUP_FAIL), the value flag is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the banded Jacobian matrix.

SPGMR Linear solver. The following optional outputs are available from the IDASPGMR module: workspace requirements, number of linear iterations, number of linear convergence failures, number of calls to the preconditioner setup and solve routines, number of calls to the Jacobian-vector product routine, number of calls to the residual routine for finite-difference Jacobian-vector product approximation, and last return value from an IDASPGMR function.

IDASpgmrGetWorkSpace

Call: `flag = IDASpgmrGetWorkSpace(ida_mem, &lenrwSG, &leniwSG);`

Description: The function IDASpgmrGetWorkSpace returns the global sizes of the IDASPGMR real and integer workspaces.

Arguments:
- ida_mem (void *) pointer to the IDA memory block.
- lenrwSG (long int) global number of real values in the IDASPGMR workspace.
- leniwSG (long int) global number of integer values in the IDASPGMR workspace.

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

IDASPGMR_SUCCESS	The optional output value has been successfuly set.
IDASPGMR_MEM_NULL	The ida_mem pointer is NULL.
IDASPGMR_LMEM_NULL	The IDASPGMR 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 $N * (\texttt{maxl} + 5) + \texttt{maxl} * (\texttt{maxl} + 4) + 1$ realtype words.

IDASpgmrGetNumLinIters

Call: `flag = IDASpgmrGetNumLinIters(ida_mem, &nliters);`

Description: The function IDASpgmrGetNumLinIters 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

`IDASPGMR_SUCCESS`	The optional output value has been successfuly set.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_LMEM_NULL`	The IDASPGMR linear solver has not been initialized.

`IDASpgmrGetNumConvFails`

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

Description The function `IDASpgmrGetNumConvFails` 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

`IDASPGMR_SUCCESS`	The optional output value has been successfuly set.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_LMEM_NULL`	The IDASPGMR linear solver has not been initialized.

`IDASpgmrGetNumPrecEvals`

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

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

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

`IDASPGMR_SUCCESS`	The optional output value has been successfuly set.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_LMEM_NULL`	The IDASPGMR linear solver has not been initialized.

`IDASpgmrGetNumPrecSolves`

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

Description The function `IDASpgmrGetNumPrecSolves` 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

`IDASPGMR_SUCCESS`	The optional output value has been successfuly set.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_LMEM_NULL`	The IDASPGMR linear solver has not been initialized.

`IDASpgmrGetNumJtimesEvals`

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

Description The function `IDASpgmrGetNumJtimesEvals` 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

`IDASPGMR_SUCCESS`	The optional output value has been successfuly set.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_LMEM_NULL`	The IDASPGMR linear solver has not been initialized.

`IDASpgmrGetNumResEvals`

Call `flag = IDASpgmrGetNumResEvals(ida_mem, &nrevalsSG);`

Description The function `IDASpgmrGetNumResEvals` 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.
`nrevalsSG` (`long int`) the cumulative number of calls to the user residual function.

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

`IDA_SUCCESS`	The optional output value has been successfuly set.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_LMEM_NULL`	The IDASPGMR linear solver has not been initialized.

Notes The value `nrevalsSG` is incremented only if the default `IDASpgmrDQJtimes` difference quotient function is used.

`IDASpgmrGetLastFlag`

Call `flag = IDASpgmrGetLastFlag(ida_mem, &flag);`

Description The function `IDASpgmrGetLastFlag` returns the last return value from an IDASPGMR routine.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.
`flag` (`int`) the value of the last return flag from an IDASPGMR function.

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

`IDASPGMR_SUCCESS`	The optional output value has been successfully set.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_LMEM_NULL`	The IDASPGMR linear solver has not been initialized.

Notes If the IDASPGMR setup function failed (`IDASolve` returned `IDA_LSETUP_FAIL`), `flag` contains the return value of the preconditioner setup function `psetup`.

If the IDASPGMR solve function failed (`IDASolve` returned `IDA_LSETUP_FAIL`), `flag` contains the error return flag from `SpgmrSolve` and will be one of: `SPGMR_CONV_FAIL`, indicating a failure to converge; `SPGMR_QRFACT_FAIL`, indicating a singular matrix found during the QR factorization; `SPGMR_PSOLVE_FAIL_REC`, indicating that the preconditioner solve function `psolve` failed recoverably; `SPGMR_MEM_NULL`, indicating that the SPGMR memory is `NULL`; `SPGMR_ATIMES_FAIL`, indicating a failure in the Jacobian times vector function; `SPGMR_PSOLVE_FAIL_UNREC`, indicating that the preconditioner solve function `psolve` failed unrecoverably; `SPGMR_GS_FAIL`, indicating a failure in the Gram-Schmidt procedure; or `SPGMR_QRSOL_FAIL`, indicating that the matrix R was found to be singular during the QR solve phase.

5.4.8 IDA reinitialization function

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

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 `IDAMalloc`. 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 `Set` calls, as described in §5.4.2.

`IDAReInit`

Call `flag = IDAReInit(ida_mem, res, t0, y0, yp0, itol, reltol, abstol);`

Description The function `IDAReInit` provides required problem specifications and reinitializes IDA.

Arguments

- `ida_mem` (`void *`) pointer to the IDA memory block.
- `res` (`IDAResFn`) is the C function which computes F. This function has the form `f(t, y, yp, r, res_data)` (for full details see §5.5).
- `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 y'.
- `itol` (`int`) is one of `IDA_SS`, `IDA_SV`, or `IDA_WF`, where `IDA_SS` indicates scalar relative error tolerance and scalar absolute error tolerance, while `IDA_SV` indicates scalar relative error tolerance and vector absolute error tolerance. The latter choice is important when the absolute error tolerance needs to be different for each component of the DAE. If `itol`=`IDA_WF`, the arguments `reltol` and `abstol` are ignored and the user is expected to provide a function to evaluate the error weight vector W from (3.6). See `IDASetEwtFn` in §5.4.5.
- `reltol` (`realtype`) is the relative error tolerance.
- `abstol` (`void *`) is a pointer to the absolute error tolerance. If `itol`=`IDA_SS`, `abstol` must be a pointer to a `realtype` variable. If `itol`=`IDA_SV`, `abstol` must be an `N_Vector` variable.

Return value The return flag `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 `IDAMalloc`.
- `IDA_ILL_INPUT` An input argument to `IDAReInit` has an illegal value.

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

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

5.5 User-supplied functions

The user-supplied functions consist of one function defining the DAE residual, (optionally) a function that provides the error weight vector, (optionally) a function that provides Jacobian related information for the linear solver (if Newton iteration is chosen), and (optionally) one or two functions that define the preconditioner for use in the SPGMR algorithm.

5.5.1 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 *res_data);
```

Purpose This function computes the problem residual for given values of the independent variable t, state vector y, and derivative 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 $y'(t)$.
- `rr` is the output vector $F(t, y, y')$.
- `res_data` is a pointer to user data - the same as the `res_data` parameter passed to `IDASetRdata`.

Return value An `IDAResFn` function type should return a value of 0 if successful, a positive value if a recoverable error occured (e.g. `yy` has an illegal value), or a negative value if a nonrecoverable error occured.

In the latter case, the integrator halts. If a recoverable error occured, the integrator will attempt to correct and retry.

Notes Allocation of memory for `yp` is handled within IDA.

5.5.2 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 weights in the WRMS norm $\| v\|_{\mathrm{WRMS}} = \sqrt{(1/N)\sum_1^N W_i \cdot v_i}$. The function type `IDAEwtFn` is defined as follows:

IDAEwtFn

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

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

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

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

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

5.5.3 Jacobian information (direct method with dense Jacobian)

If the direct linear solver with dense treatment of the Jacobian is used (i.e. `IDADense` is called in Step 7 of §5.3), the user may provide a function of type `IDADenseJacFn` defined by

IDADenseJacFn

Definition
```
typedef int (*IDADenseJacFn)(long int Neq, realtype tt,
                             N_Vector yy, N_Vector yp, N_Vector rr,
                             realtype c_j, void *jac_data,
                             DenseMat Jac,
                             N_Vector tmp1, N_Vector tmp2,
                             N_Vector tmp3);
```

Purpose: This function computes the dense Jacobian (or an approximation to it) of the DAE system.

Arguments:
- `Neq` is the problem size (number of equations).
- `tt` is the current value of the independent variable t.
- `yy` is the current value of the dependent variable vector, $y(t)$.
- `yp` is the current value of $y'(t)$.
- `rr` is the current value of the vector $F(t, y, y')$.
- `c_j` is the scalar in the system Jacobian, proportional to the inverse of the step size.
- `jac_data` is a pointer to user data - the same as the `jac_data` parameter passed to `IDADenseSetJacData`.
- `Jac` is the output Jacobian matrix.
- `tmp1`
- `tmp2`
- `tmp3` are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDADenseJacFn` as temporary storage or work space.

Return value: An `IDADenseJacFn` function type should return 0 if successful, a positive value if a recoverable error occured, or a negative value if a nonrecoverable error occured.

In the case of a recoverable eror return, the integrator will attempt to recover by reducing the stepsize, and hence changing α in (3.5).

Notes: A user-supplied dense Jacobian function must load the `Neq` $\times$ `Neq` dense matrix `Jac` with an approximation to the Jacobian matrix J 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 `DenseMat` (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 `DenseMat` type. `DENSE_ELEM(Jac, i, j)` references the (`i`, `j`)-th element of the dense matrix `Jac` (`i`, `j`$= 0 \ldots$ `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 `j`th column of `Jac` (`j`$= 0 \ldots$ `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 `DenseMat` type and the accessor macros `DENSE_ELEM` and `DENSE_COL` are documented in §8.1.

If the user's `IDADenseJacFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, use the `IDAGet*` functions described in §5.4.7. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundialstypes.h`.

5.5.4 Jacobian information (direct method with banded Jacobian)

If the direct linear solver with banded treatment of the Jacobian is used (i.e. `IDABand` is called in Step 7 of §5.3), the user may provide a function of type `IDABandJacFn` defined as follows:

IDABandJacFn

Definition
```
typedef int (*IDABandJacFn)(long int Neq, long int mupper,
                            long int mlower, realtype tt,
                            N_Vector yy, N_Vector yp, N_Vector rr,
                            realtype c_j, void *jac_data,
                            BandMat Jac,
                            N_Vector tmp1, N_Vector tmp2,
                            N_Vector tmp3);
```

Purpose: This function computes the banded Jacobian (or a banded approximation to it).

Arguments:

Neq	is the problem size.
mlower	
mupper	are the lower and upper 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 $y'(t)$.
rr	is the current value of the vector $F(t, y, y')$.
c_j	is the scalar in the system Jacobian, proportional to the inverse of the step size.
jac_data	is a pointer to user data - the same as the jac_data parameter passed to IDABandSetJacData.
Jac	is the output Jacobian matrix.
tmp1	
tmp2	
tmp3	are pointers to memory allocated for variables of type N_Vector which can be used by IDABandJacFn as temporary storage or work space.

Return value: A IDABandJacFn function type should return 0 if successful, a positive value if a recoverable error occured, or a negative value if a nonrecoverable error occured.

In the case of a recoverable eror return, the integrator will attempt to recover by reducing the stepsize, and hence changing α in (3.5).

Notes: A user-supplied band Jacobian function must load the band matrix Jac of type BandMat with the elements of the Jacobian $J(t, y, y')$ 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 BandMat 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 $\leq$ m-n $\leq$ mlower. Alternatively, BAND_COL(Jac, j) returns a pointer to the diagonal element of the jth column of Jac, and if we assign this address to realtype *col_j, then the ith element of the jth column is given by BAND_COL_ELEM(col_j, i, j), counting from 0. Thus for (m, n) within the band, $J_{m,n}$ can be loaded by setting col_n = BAND_COL(Jac, n-1); BAND_COL_ELEM(col_n, m-1, n-1) = $J_{m,n}$. The elements of the jth column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type BandMat. The array col_n can be indexed from −mupper to mlower. For large problems, it is more efficient to use the combination of BAND_COL and BAND_COL_ELEM than to use the

BAND_ELEM. As in the dense case, these macros all number rows and columns starting from 0, not 1.

The BandMat type and the accessor macros BAND_ELEM, BAND_COL, and BAND_COL_ELEM are documented in §8.2.

If the user's IDABandJacFn function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, use the IDAGet* functions described in §5.4.7. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundialstypes.h.

5.5.5 Jacobian information (SPGMR matrix-vector product)

If an iterative SPGMR linear solver is selected (IDASpgmr is called in step 7 of §5.3) the user may provide a function of type IDASpgmrJacTimesVecFn in the following form:

IDASpgmrJacTimesVecFn

Definition
```
typedef int (*IDASpgmrJacTimesVecFn)(realtype tt,
                                     N_Vector yy, N_Vector yp, N_Vector rr,
                                     N_Vector v, N_Vector Jv,
                                     realtype c_j, void *jac_data,
                                     N_Vector tmp1, N_Vector tmp2);
```

Purpose
This function computes the product of the problem Jacobian and the vector v (or an approximation to it).

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 $y'(t)$.
- rr is the current value of the vector $F(t, y, y')$.
- v is the vector by which the Jacobian must be multiplied to the right.
- Jv is the output vector computed.
- c_j is the scalar in the system Jacobian, proportional to the inverse of the step size.
- jac_data is a pointer to user data - the same as the jac_data parameter passed to IDASpgmrSetJacData.
- tmp1
- tmp2 are pointers to memory allocated for variables of type N_Vector which can be used by IDASpgmrJacTimesVecFn 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 IDASpgmrJacTimesVecFn function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, use the IDAGet* functions described in §5.4.7. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundialstypes.h.

5.5.6 Preconditioning (SPGMR linear system solution)

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

`IDASpgmrPrecSolveFn`

Definition
```
typedef int (*IDASpgmrPrecSolveFn)(realtype tt,
                                   N_Vector yy, N_Vector yp, N_Vector rr,
                                   N_Vector rvec, N_Vector zvec,
                                   realtype c_j, realtype delta,
                                   void *prec_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 $y'(t)$.
- `rr` is the current value of the vector $F(t, y, y')$.
- `rvec` is the right-hand side vector of the linear system.
- `zvec` is the output vector computed.
- `c_j` is the scalar in the system Jacobian, proportional to the inverse of the step size.
- `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 §5.4.7).
- `prec_data` is a pointer to user data - the same as the `prec_data` parameter passed to the function `IDASpgmrSetPrecData`.
- `tmp` is a pointer to memory allocated for a variable of type `N_Vector` which can be used for work space.

Return value: The value to be returned by the preconditioner solve function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), negative for an unrecoverable error (in which case the integration is halted).

5.5.7 Preconditioning (SPGMR 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 `IDASpgmrPrecSetupFn`, defined as follows:

`IDASpgmrPrecSetupFn`

Definition
```
typedef int (*IDASpgmrPrecSetupFn)(realtype tt,
                                   N_Vector yy, N_Vector yp, N_Vector rr,
                                   realtype c_j, void *prec_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 `IDASpgmrPrecSetupFn` 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 $y'(t)$.
- `rr` is the current value of the vector $F(t, y, y')$.
- `c_j` is the scalar in the system Jacobian, proportional to the inverse of the step size.

`prec_data` is a pointer to user data, the same as the `prec_data` parameter passed to `IDASpgmrSetPrecData`.

`tmp1`

`tmp2`

`tmp3` are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDASpgmrPrecSetupFn` 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 `IDASpgmrPrecSetupFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, use the `IDAGet*` functions described in §5.4.7. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundialstypes.h`.

5.6 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 (3.4) that must be solved at each time step. The linear algebraic system is large, sparse, and structured. However, if a Krylov iterative method is to be effective in this setting, then a nontrivial preconditioner needs to be used. Otherwise, the rate of convergence of the Krylov iterative method is usually unacceptably slow. Unfortunately, an effective preconditioner tends to be problem-specific.

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used for several realistic, large-scale problems [12] 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 $G(t, y, y')$ which approximates the function $F(t, y, y')$ in the definition of the DAE system (3.1). However, the user may set $G = F$. Corresponding to the domain decomposition, there is a decomposition of the solution vector y into M disjoint blocks y_m, and a decomposition of G into blocks G_m. The block G_m depends on y_m and also on components of $y_{m'}$ associated with neighboring sub-domains (so-called ghost-cell data). Let $\bar{y}_m$ denote y_m augmented with those other components on which G_m

depends. Then we have

$$G(t, y) = [G_1(t, \bar{y}_1), G_2(t, \bar{y}_2), \ldots, G_M(t, \bar{y}_M)]^T \tag{5.1}$$

and each of the blocks $G_m(t, \bar{y}_m)$ is uncoupled from the others.

The preconditioner associated with this decomposition has the form

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

where

$$P_m \approx \partial G_m / \partial y_m + \alpha \partial G_m / \partial y'_m \tag{5.3}$$

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

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

reduces to solving each of the equations

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

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

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

The IDABBDPRE module calls two user-provided functions to construct P: a required function `Gres` (of type `IDABBDLocalFn`) which approximates the residual function $G(t, y) \approx F(t, y)$ 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 `res_data` as passed by the user to `IDASetRdata` and passed to the user's function `res`, and neither function has a return value. The user is responsible for providing space (presumably within `res_data`) for components of `yy` and `yp` that are communicated by `Gcomm` from the other processors, and that are then used by `Gres`, which is not expected to do any communication.

`IDABBDLocalFn`

Definition

```
typedef void (*IDABBDLocalFn)(long int Nlocal, realtype tt,
                              N_Vector yy, N_Vector yp, N_Vector gval,
                              void *res_data);
```

Purpose This function computes $G(t, y, y')$. 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.
	`res_data` is a pointer to user data - the same as the `res_data` parameter passed to `IDASetRdata`.
Return value	An `IDABBDLocalFn` function type does not have a return value.
Notes	This function assumes that all inter-processor communication of data needed to calculate `gval` has already been done, and this data is accessible within `res_data`. The case where G is mathematically identical to F is allowed.

`IDABBDCommFn`

Definition
```
typedef void (*IDABBDCommFn)(long int Nlocal, realtype tt,
                             N_Vector yy, N_Vector yp, void *res_data);
```

Purpose: This 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.
- `res_data` is a pointer to user data - the same as the `res_data` parameter passed to `IDASetRdata`.

Return value: An `IDABBDCommFn` function type does not have a return value.

Notes: The `Gcomm` function is expected to save communicated data in space defined within the structure `res_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 comunication was done in `res`, then `Gcomm` = `NULL` can be passed in the call to `IDABBDPrecAlloc` (see below).

Besides the header files required for the integration of the DAE problem (see §5.2), to use the IDABBDPRE module, the main program must include the header file `idabbdpre.h` which declares the needed function prototypes.

The following is a summary of the usage of this module and describes the sequence of calls in the user main program. Steps that are unchanged from the user main program presented in §5.3 are grayed-out.

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

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

   ```
   bbd_data = IDABBDPrecAlloc(ida_mem, Nlocal, mudq, mldq,
                              mukeep, mlkeep, dq_rel_yy, Gres, Gcomm);
   ```

   to allocate memory for and initialize a data structure `bbd_data` to be passed to the IDASPGMR linear solver. The last two arguments of `IDABBDPrecAlloc` are the two user-supplied functions described above.

8. **Attach the IDASPGMR linear solver**

   `flag = IDABBDSpgmr(ida_mem, maxl, bbd_data);`

   The function `IDABBDSpgmr` is a wrapper around the IDASPGMR specification function `IDASpgmr` and performs the following actions:

   - Attaches the IDASPGMR linear solver to the main IDA solver memory;
   - Sets the preconditioner data structure for IDABBDPRE;
   - Sets the preconditioner setup function for IDABBDPRE;
   - Sets the preconditioner solve function for IDABBDPRE;

9. **Set linear solver optional inputs**

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

10. **Advance solution in time**

11. **Deallocate memory for solution vector**

12. **Free the IDABBDPRE data structure**

    `IDABBDPrecFree(bbd_data);`

13. **Free solver memory**

14. **Finalize MPI**

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

`IDABBDPrecAlloc`

Call
```
bbd_data = IDABBDPrecAlloc(ida_mem, Nlocal, mudq, mldq,
                           mukeep, mlkeep, dq_rel_yy, Gres, Gcomm);
```

Description The function `IDABBDPrecAlloc` initializes and allocates 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 $G(t, y, y')$.
- `Gcomm` (`IDABBDCommFn`) the optional C function which performs all inter-process communication required for the computation of $G(t, y, y')$.

Return value If successful, `IDABBDPrecAlloc` returns a pointer to the newly created IDABBDPRE memory block (of type `void *`). If an error occurred, `IDABBDPrecAlloc` returns `NULL`.

Notes If one of the half-bandwidths `mudq` or `mldq` to be used in the difference-quotient calculation of the approximate Jacobian is negative or exceeds the value `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.

`IDABBDSpgmr`

Call `flag = IDABBDSpgmr(ida_mem, maxl, bbd_data);`

Description The function `IDABBDSpgmr` links the IDABBDPRE data to the IDASPGMR linear solver and attaches the latter to the IDA memory block.

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 `IDASPGMR_MAXL`= 5.
- `bbd_data` (`void *`) pointer to the IDABBDPRE data structure.

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

`IDASPGMR_SUCCESS`	The IDASPGMR initialization was successful.
`IDASPGMR_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPGMR_MEM_FAIL`	A memory allocation request failed.
`IDA_PDATA_NULL`	The IDABBDPRE preconditioner has not been initialized.

`IDABBDPrecFree`

Call `IDABBDPrecFree(bbd_data);`

Description The function `IDABBDPrecFree` frees the pointer allocated by `IDABBDPrecAlloc`.

Arguments The only argument of `IDABBDPrecFree` is the pointer to the IDABBDPRE data structure (of type `void *`).

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

The IDABBDPRE module also provides a reinitialization function to allow solving a sequence of problems of the same size with IDASPGMR/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(bbd_data, mudq, mldq, dq_rel_yy, Gres, Gcomm);`

Description The function `IDABBDPrecReInit` reinitializes the IDABBDPRE preconditioner.

Arguments
- `bbd_data` (`void *`) pointer to the IDABBDPRE data structure.
- `mudq` (`long int`) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mldq` (`long int`) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.

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 $G(t, y, y')$.

Gcomm (IDABBDCommFn) the optional C function which performs all inter-process communication required for the computation of $G(t, y, y')$.

Return value The return value of IDABBDPrecReInit is always IDA_SUCCESS.

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(bbd_data, &lenrwBBDP, &leniwBBDP);`

Description The function IDABBDPrecGetWorkSpace returns the local sizes of the IDABBDPRE real and integer workspaces.

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

IDA_SUCCESS The optional output value has been successfuly set.
IDA_PDATA_NULL The IDABBDPRE preconditioner has not been initialized.

Notes In terms of the local vector dimension N_l, the actual size of the real workspace is N_l (2 mlkeep + mukeep + smu +2) realtype words, where smu $= \min(N_l - 1,$ mukeep + mlkeep).

The actual size of the integer workspace is N_l integer words.

`IDABBDPrecGetNumGfnEvals`

Call `flag = IDABBDPrecGetNumGfnEvals(bbd_data, &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
bbd_data (void *) pointer to the IDABBDPRE data structure.
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

IDA_SUCCESS The optional output value has been successfuly set.
IDA_PDATA_NULL The IDABBDPRE preconditioner has not been initialized.

The costs associated with IDABBDPRE also include nlinsetups LU factorizations, nlinsetups calls to Gcomm, and npsolves banded backsolve calls, where nlinsetups and npsolves are optional IDA outputs (see §5.4.7).

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 two provided within SUNDIALS, a serial and an MPI parallel implementations.

The generic `N_Vector` type is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the vector, and an *ops* field pointing to a structure with generic vector operations. The type `N_Vector` is defined as

```
typedef struct _generic_N_Vector *N_Vector;

struct _generic_N_Vector {
    void *content;
    struct _generic_N_Vector_Ops *ops;
};
```

The `_generic_N_Vector_Ops` structure is essentially a list of pointers to the various actual vector operations, and is defined as

```
struct _generic_N_Vector_Ops {
  N_Vector    (*nvclone)(N_Vector);
  N_Vector    (*nvcloneempty)(N_Vector);
  void        (*nvdestroy)(N_Vector);
  void        (*nvspace)(N_Vector, long int *, long int *);
  realtype*   (*nvgetarraypointer)(N_Vector);
  void        (*nvsetarraypointer)(realtype *, N_Vector);
  void        (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
  void        (*nvconst)(realtype, N_Vector);
  void        (*nvprod)(N_Vector, N_Vector, N_Vector);
  void        (*nvdiv)(N_Vector, N_Vector, N_Vector);
  void        (*nvscale)(realtype, N_Vector, N_Vector);
  void        (*nvabs)(N_Vector, N_Vector);
  void        (*nvinv)(N_Vector, N_Vector);
  void        (*nvaddconst)(N_Vector, realtype, N_Vector);
  realtype    (*nvdotprod)(N_Vector, N_Vector);
  realtype    (*nvmaxnorm)(N_Vector);
  realtype    (*nvwrmsnorm)(N_Vector, N_Vector);
  realtype    (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvmin)(N_Vector);
```

```
  realtype    (*nvwl2norm)(N_Vector, N_Vector);
  realtype    (*nvl1norm)(N_Vector);
  void        (*nvcompare)(realtype, N_Vector, N_Vector);
  booleantype (*nvinvtest)(N_Vector, N_Vector);
  booleantype (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvminquotient)(N_Vector, N_Vector);
};
```

The generic NVECTOR module also defines and implements the vector operations acting on `N_Vector`. These routines are nothing but wrappers for the vector operations defined by a particular NVECTOR implementation, which are accessed through the *ops* field of the `N_Vector` structure. To illustrate this point we show below the implementation of a typical vector operation from the generic NVECTOR module, namely `N_VScale`, which performs the scaling of a vector `x` by a scalar `c`:

```
void N_VScale(realtype c, N_Vector x, N_Vector z)
{
   z->ops->nvscale(c, x, z);
}
```

Table 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 the data array.
N_VDestroy	`N_VDestroy(v);` Destroys the `N_Vector` `v` and frees memory allocated for its internal data.
N_VSpace	`N_VSpace(nvSpec, &lrw, &liw);` Returns storage requirements for one `N_Vector`. `lrw` contains the number of realtype words and `liw` contains the number of integer words.
N_VGetArrayPointer	`vdata = N_VGetArrayPointer(v);` Returns a pointer to a `realtype` array from the `N_Vector` `v`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the solver-specific interfaces to the dense and banded linear solvers, as well as the interfaces to the banded preconditioners provided with SUNDIALS.
N_VSetArrayPointer	`N_VSetArrayPointer(vdata, v);` Overwrites the data in an `N_Vector` with a given array of `realtype`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the interfaces to the dense linear solver.
N_VLinearSum	`N_VLinearSum(a, x, b, y, z);` Performs the operation $z = ax + by$, where a and b are scalars and x and y are of type `N_Vector`: $z_i = ax_i + by_i$, $i = 0, \ldots, n-1$.
N_VConst	`N_VConst(c, z);` Sets all components of the `N_Vector` `z` to `c`: $z_i = c$, $i = 0, \ldots, n-1$.
N_VProd	`N_VProd(x, y, z);` Sets the `N_Vector` `z` to be the component-wise product of the `N_Vector` inputs `x` and `y`: $z_i = x_i y_i$, $i = 0, \ldots, n-1$.
N_VDiv	`N_VDiv(x, y, z);` Sets the `N_Vector` `z` to be the component-wise ratio of the `N_Vector` inputs `x` and `y`: $z_i = x_i / y_i$, $i = 0, \ldots, n-1$. The y_i may not be tested for 0 values. It should only be called with an `x` that is guaranteed to have all nonzero components.

continued on next page

continued from last page	
Name	**Usage and Description**
N_VScale	`N_VScale(c, x, z);` Scales the `N_Vector` `x` by the scalar `c` and returns the result in `z`: $z_i = cx_i$, $i = 0, \ldots, n-1$.
N_VAbs	`N_VAbs(x, y);` Sets the components of the `N_Vector` `y` to be the absolute values of the components of the `N_Vector` `x`: $y_i = \lvert x_i \rvert$, $i = 0, \ldots, n-1$.
N_VInv	`N_VInv(x, z);` Sets the components of the `N_Vector` `z` to be the inverses of the components of the `N_Vector` `x`: $z_i = 1.0/x_i$, $i = 0, \ldots, n-1$. This routine may not check for division by 0. It should be called only with an `x` which is guaranteed to have all nonzero components.
N_VAddConst	`N_VAddConst(x, b, z);` Adds the scalar `b` to all components of `x` and returns the result in the `N_Vector` `z`: $z_i = x_i + b$, $i = 0, \ldots, n-1$.
N_VDotProd	`d = N_VDotProd(x, y);` Returns the value of the ordinary dot product of `x` and `y`: $d = \sum_{i=0}^{n-1} x_i y_i$.
N_VMaxNorm	`m = N_VMaxNorm(x);` Returns the maximum norm of the `N_Vector` `x`: $m = \max_i \lvert x_i \rvert$.
N_VWrmsNorm	`m = N_VWrmsNorm(x, w)` Returns the weighted root-mean-square norm of the `N_Vector` `x` with weight vector `w`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i)^2 \right) / n}$.
N_VWrmsNormMask	`m = N_VWrmsNormMask(x, w, id);` Returns the weighted root mean square norm of the `N_Vector` `x` with weight vector `w` built using only the elements of `x` corresponding to nonzero elements of the `N_Vector` `id`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i \operatorname{sign}(id_i))^2 \right) / n}$.
N_VMin	`m = N_VMin(x);` Returns the smallest element of the `N_Vector` `x`: $m = \min_i x_i$.
N_VWL2Norm	`m = N_VWL2Norm(x, w);` Returns the weighted Euclidean ℓ_2 norm of the `N_Vector` `x` with weight vector `w`: $m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}$.
N_VL1Norm	`m = N_VL1Norm(x);` Returns the ℓ_1 norm of the `N_Vector` `x`: $m = \sum_{i=0}^{n-1} \lvert x_i \rvert$.

continued on next page

continued from last page	
Name	**Usage and Description**
N_VCompare	`N_VCompare(c, x, z);` Compares the components of the `N_Vector` `x` to the scalar `c` and returns an `N_Vector` `z` such that: $z_i = 1.0$ if $\|x_i\| \geq c$ and $z_i = 0.0$ otherwise.
N_VInvTest	`t = N_VInvTest(x, z);` Sets the components of the `N_Vector` `z` to be the inverses of the components of the `N_Vector` `x`, with prior testing for zero values: $z_i = 1.0/x_i$, $i = 0, \ldots, n-1$. This routine returns `TRUE` if all components of `x` are nonzero (successful inversion) and returns `FALSE` otherwise.
N_VConstrMask	`t = N_VConstrMask(c, x, m);` Performs the following constraint tests: $x_i > 0$ if $c_i = 2$, $x_i \geq 0$ if $c_i = 1$, $x_i \leq 0$ if $c_i = -1$, $x_i < 0$ if $c_i = -2$. There is no constraint on x_i if $c_i = 0$. This routine returns `FALSE` if any element failed the constraint test, `TRUE` if all passed. It also sets a mask vector `m`, with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.
N_VMinQuotient	`minq = N_VMinQuotient(num, denom);` This routine returns the minimum of the quotients obtained by termwise dividing $\mathtt{num}_i$ by $\mathtt{denom}_i$. A zero element in `denom` will be skipped. If no such quotients are found, then the large value `BIG_REAL` (defined in the header file `sundialstypes.h`) is returned.

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 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 and provides the following user-callable routines:

- `N_VNew_Serial`

  This function creates and allocates memory for a serial `N_Vector`. Its only argument is the vector length.

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

- `N_VNewEmpty_Serial`

  This function creates a new serial `N_Vector` with an empty (`NULL`) data array.

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

- `N_VMake_Serial`

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

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

- `N_VNewVectorArray_Serial`

  This function creates an array of `count` serial vectors.

  ```
  N_Vector *N_VNewVectorArray_Serial(int count, long int vec_length);
  ```

- `N_VNewVectorArrayEmpty_Serial`

  This function creates an array of `count` serial vectors, each with an empty (`NULL`) data array.

  ```
  N_Vector *N_VNewVectorArrayEmpty_Serial(int count, long int vec_length);
  ```

- `N_VDestroyVectorArray_Serial`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VNewVectorArray_Serial` or with `N_VNewVectorArrayEmpty_Serial`.

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

- `N_VPrint_Serial`

  This function prints the content of a serial vector to `stdout`.

  ```
  void N_VPrint_Serial(N_Vector v);
  ```

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_S(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_S(v,i)` within the loop.

- ! The NVECTOR_SERIAL constructor functions `N_VNewEmpty_Serial`, `N_VMake_Serial`, and `N_VNewVectorArrayEmpty_Serial` set the field *own_data* = `FALSE`. The functions `N_VDestroy_Serial` and `N_VDestroyVectorArray_Serial` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- ! To maximize efficiency, vector operations in the NVECTOR_SERIAL implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.2 The NVECTOR_PARALLEL implementation

The parallel implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_PARALLEL, defines the *content* field of `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to the beginning of a contiguous local data array, an MPI communicator, an a boolean flag *own_data* indicating ownership of the data array *data*.

```
struct _N_VectorContent_Parallel {
  long int local_length;
  long int global_length;
  booleantype own_data;
  realtype *data;
  MPI_Comm comm;
};
```

The following seven macros are provided to access the content of a NVECTOR_PARALLEL vector. The suffix `_P` in the names denotes parallel version.

- `NV_CONTENT_P`

  This macro gives access to the contents of the parallel vector `N_Vector`.

  The assignment `v_cont = NV_CONTENT_P(v)` sets `v_cont` to be a pointer to the `N_Vector` content structure of type `struct _N_VectorParallelContent`.

  Implementation:

  ```
  #define NV_CONTENT_P(v) ( (N_VectorContent_Parallel)(v->content) )
  ```

- `NV_OWN_DATA_P`, `NV_DATA_P`, `NV_LOCLENGTH_P`, `NV_GLOBLENGTH_P`

  These macros give individual access to the parts of the content of a parallel `N_Vector`.

  The assignment `v_data = NV_DATA_P(v)` sets `v_data` to be a pointer to the first component of the local data for the `N_Vector v`. The assignment `NV_DATA_P(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

  The assignment `v_llen = NV_LOCLENGTH_P(v)` sets `v_llen` to be the length of the local part of `v`. The call `NV_LENGTH_P(v) = llen_v` sets the local length of `v` to be `llen_v`.

  The assignment `v_glen = NV_GLOBLENGTH_P(v)` sets `v_glen` to be the global length of the vector `v`. The call `NV_GLOBLENGTH_P(v) = glen_v` sets the global length of `v` to be `glen_v`.

  Implementation:

```
#define NV_OWN_DATA_P(v)   ( NV_CONTENT_P(v)->own_data )
#define NV_DATA_P(v)       ( NV_CONTENT_P(v)->data )
#define NV_LOCLENGTH_P(v)  ( NV_CONTENT_P(v)->local_length )
#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )
```

- NV_COMM_P

  This macro provides access to the MPI communicator used by the NVECTOR_PARALLEL vectors.

  Implementation:

  ```
  #define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
  ```

- NV_Ith_P

  This macro gives access to the individual components of the local data array of an N_Vector.

  The assignment r = NV_Ith_P(v,i) sets r to be the value of the i-th component of the local part of v. The assignment NV_Ith_P(v,i) = r sets the value of the i-th component of the local part of v to be r.

  Here i ranges from 0 to $n-1$, where n is the local length.

  Implementation:

  ```
  #define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
  ```

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Table 6.1 and provides the following user-callable routines:

- N_VNew_Parallel

  This function creates and allocates memory for a parallel vector.

  ```
  N_Vector N_VNew_Parallel(MPI_Comm comm,
                           long int local_length,
                           long int global_length);
  ```

- N_VNewEmpty_Parallel

  This function creates a new parallel N_Vector with an empty (NULL) data array.

  ```
  N_Vector N_VNewEmpty_Parallel(MPI_Comm comm,
                                long int local_length,
                                long int global_length);
  ```

- N_VMake_Parallel

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

  ```
  N_Vector N_VMake_Parallel(MPI_Comm comm,
                            long int local_length,
                            long int global_length,
                            realtype *v_data);
  ```

- N_VNewVectorArray_Parallel

  This function creates an array of count parallel vectors.

  ```
  N_Vector *N_VNewVectorArray_Parallel(int count,
                                       MPI_Comm comm,
                                       long int local_length,
                                       long int global_length);
  ```

- N_VNewVectorArrayEmpty_Parallel

  This function creates an array of count parallel vectors, each with an empty (NULL) data array.

```
N_Vector *N_VNewVectorArrayEmpty_Parallel(int count,
                                          MPI_Comm comm,
                                          long int local_length,
                                          long int global_length);
```

- N_VDestroyVectorArray_Parallel

  This function frees memory allocated for the array of count variables of type N_Vector created with N_VNewVectorArray_Parallel or with N_VNewVectorArrayEmpty_Parallel.

```
void N_VDestroyVectorArray_Parallel(N_Vector *vs, int count);
```

- N_VPrint_Parallel

  This function prints the content of a parallel vector to stdout.

```
void N_VPrint_Parallel(N_Vector v);
```

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the local component array via v_data = NV_DATA_P(v) and then access v_data[i] within the loop than it is to use NV_Ith_P(v,i) within the loop.

- ! The NVECTOR_PARALLEL constructor functions N_VNewEmpty_Parallel, N_VMake_Parallel, and N_VNewVectorArrayEmpty_Parallel set the field *own_data* = FALSE. The functions N_VDestroy_Parallel and N_VDestroyVectorArray_Parallel will not attempt to free the pointer *data* for any N_Vector with *own_data* set to FALSE. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- ! To maximize efficiency, vector operations in the NVECTOR_PARALLEL implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsability to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

6.3 NVECTOR functions used by IDA

In Table 6.2 below, we list the vector functions 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 four columns show function usage within each of the three IDA linear solvers and the IDABBDPRE preconditioner module.

There is one subtlety in the IDASPGMR column hidden by the table. The N_VDotProd function is called both within the implementation file idaspgmr.c for the IDASPGMR solver and within the implementation files spgmr.c and iterative.c for the generic SPGMR solver upon which the IDASPGMR solver is implemented. Also, although N_VDiv and N_VProd are not called within the implementation file idaspgmr.c, they are called within the implementation file spgmr.c and so are required by the IDASPGMR solver module. This issue does not arise for the other two 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.

Four of the functions listed in Table 6.1, N_VWL2Norm, N_VL1Norm, N_VCloneEmpty, 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	IDADENSE	IDABAND	IDASPGMR	IDABBDPRE
N_VClone	✓			✓	✓
N_VDestroy	✓			✓	✓
N_VSpace	✓				
N_VGetArrayPointer		✓	✓		✓
N_VSetArrayPointer		✓			
N_VLinearSum	✓	✓		✓	
N_VConst	✓			✓	
N_VProd	✓			✓	
N_VDiv	✓			✓	
N_VScale	✓	✓	✓	✓	✓
N_VAbs	✓				
N_VInv	✓				
N_VAddConst	✓				
N_VDotProd				✓	
N_VMaxNorm	✓				
N_VWrmsNorm	✓				
N_VMin	✓				
N_VMinQuotient	✓				
N_VConstrMask	✓				
N_VWrmsNormMask	✓				
N_VCompare	✓				

Chapter 7

Providing Alternate Linear Solver Modules

The central IDA module interfaces with the linear solver module to be used by way of calls to five routines. These are denoted here by `linit`, `lsetup`, `lsolve`, `lperf`, and `lfree`. Briefly, their purposes are as follows:

- `linit`: initialize and allocate memory specific to the linear solver;
- `lsetup`: evaluate and preprocess the Jacobian or preconditioner;
- `lsolve`: solve the linear system;
- `lperf`: monitor performance and issue warnings;
- `lfree`: free the linear solver memory.

A linear solver module must also provide a user-callable specification routine (like those described in §5.4.2) which will attach the above five routines to the main IDA memory block. The return value of the specification routine should be: `SUCCESS` = 0 if the routine was successful, `LMEM_FAIL` = -1 if a memory allocation failed, or `LIN_ILL_INPUT` = -2 if some input was illegal.

These five routines, 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 routines. Note that the call list of each routine includes a pointer to the main IDA memory block, by which the routine can access various data related to the IDA solution. The contents of this memory block are given in the file `ida.h` (but not reproduced here, for the sake of space).

Initialization routine. The type definition of `linit` is

`linit`

Definition `int (*linit)(IDAMem IDA_mem);`

Purpose The purpose of `linit` is to complete initializations for a specific linear solver, such as counters and statistics.

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.

Notes If an error does occur, an appropriate message should be sent to `IDA_mem->ida_errfp`.

Setup routine. 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`. It may re-compute Jacobian-related data if it deems necessary.

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 y' vector for the current IDA internal step.
- `resp` is the value of the residual function at `yyp` and `ypp`, i.e. $F(t_n, y_{pred}, y'_{pred})$.
- `vtemp1`
- `vtemp2`
- `vtemp3` are temporary variables of type `N_Vector` provided for use by `lsetup`.

Return value The `lsetup` routine should return `0` if successful, a positive value for a recoverable error, and a negative value for an unrecoverable error.

Solve routine. The type definition of `lsolve` is

`lsolve`

Definition `int (*lsolve)(IDAMem IDA_mem, N_Vector b, N_Vector weight, N_Vector ycur, N_Vector ypcur, N_Vector rescur);`

Purpose The routine `lsolve` must solve the linear equation $Mx = b$, where M is some approximation to $J = dF/dy + c_j dF/dy'$ and the right-hand side vector b is input.

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 reciprocals of the W_i of (3.6).
- `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 $y'(t_n)$.
- `rescur` is a vector that contains $F(t_n, y_{cur}, y'_{cur})$.

Return value `lsolve` returns a positive value for a recoverable error and a negative value for an unrecoverable error. Success is indicated by a `0` return value.

Performance monitoring routine. The type definition of `lperf` is

`lperf`

Definition `int (*lperf)(IDAMem IDA_mem, int perftask);`

Purpose The routine `lperf` is to monitor the performance of the linear solver.

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.

Return value The `lperf` return value is ignored.

Memory deallocation routine. The type definition of `lfree` is `lfree`

Definition `void (*lfree)(IDAMem IDA_mem);`

Purpose The routine `lfree` 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 This routine has no return value.

Notes This routine is called once a problem has been completed and the linear solver is no longer needed.

Chapter 8

Generic Linear Solvers in SUNDIALS

In this section, we describe three generic linear solver code modules that are included in IDA, but which are of potential use as generic packages in themselves, either in conjunction with the use of IDA or separately. These modules are:

- The DENSE matrix package, which includes the matrix type `DenseMat`, macros and functions for `DenseMat` matrices, and functions for small dense matrices treated as simple array types.
- The BAND matrix package, which includes the matrix type `BandMat`, macros and functions for `BandMat` matrices, and functions for small band matrices treated as simple array types.
- The SPGMR package, which includes a solver for the scaled preconditioned GMRES method.

For the sake of space, the functions for `DenseMat` and `BandMat` matrices and the functions in SPGMR are only summarized briefly, since they are less likely to be of direct use in connection with IDA. The functions for small dense matrices are fully described, because we expect that they will be useful in the implementation of preconditioners used with the combination of IDA and the IDASPGMR solver.

8.1 The DENSE module

8.1.1 Type DenseMat

The type `DenseMat` is defined to be a pointer to a structure with a size and a data field:

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

The *size* field indicates the number of columns (which is the same as the number of rows) of a dense matrix, while the *data* field is a two dimensional array used for component storage. The elements of a dense matrix are stored columnwise (i.e columns are stored one on top of the other in memory). If `A` is of type `DenseMat`, then the (`i`,`j`)-th element of `A` (with $0 \leq$ `i`, `j` $\leq$ `size`-1) is given by the expression `(A->data)[j][i]` or by the expression `(A->data)[0][j*size+i]`. The macros below allow a user to efficiently access individual matrix elements without writing out explicit data structure references and without knowing too much about the underlying element storage. The only storage assumption needed is that elements are stored columnwise and that a pointer to the `j`-th column of elements can be obtained via the `DENSE_COL` macro. Users should use these macros whenever possible.

8.1.2 Accessor Macros

The following two macros are defined by the DENSE module to provide access to data in the `DenseMat` type:

- `DENSE_ELEM`

  Usage : `DENSE_ELEM(A,i,j) = a_ij;` or `a_ij = DENSE_ELEM(A,i,j);`

  `DENSE_ELEM` references the (`i`,`j`)-th element of the $N \times N$ `DenseMat A`, $0 \leq$ `i`, `j` $\leq N-1$.

- `DENSE_COL`

  Usage : `col_j = DENSE_COL(A,j);`

  `DENSE_COL` references the `j`-th column of the $N \times N$ `DenseMat A`, $0 \leq$ `j` $\leq N-1$. The type of the expression `DENSE_COL(A,j)` is `realtype *` . After the assignment in the usage above, `col_j` may be treated as an array indexed from 0 to $N-1$. The (`i`, `j`)-th element of `A` is referenced by `col_j[i]`.

8.1.3 Functions

The following functions for `DenseMat` matrices are available in the DENSE package. For full details, see the header file `dense.h`.

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

8.1.4 Small Dense Matrix Functions

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

- `denalloc`

  `denalloc(n)` allocates storage for an `n` by `n` dense matrix. It returns a pointer to the newly allocated storage if successful. If the memory request cannot be satisfied, then `denalloc` returns `NULL`. The underlying type of the dense matrix returned is `realtype**`. If we allocate a dense matrix `realtype** a` by `a = denalloc(n)`, then `a[j][i]` references the (`i`,`j`)-th element of the matrix `a`, $0 \leq$ `i`, `j` $\leq$ `n`-1, and `a[j]` is a pointer to the first element in the `j`-th column of `a`. The location `a[0]` contains a pointer to `n`^{2} contiguous locations which contain the elements of `a`.

- `denallocpiv`

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

- `gefa`

  `gefa(a,n,p)` factors the `n` by `n` dense matrix `a`. It overwrites the elements of `a` with its LU factors and keeps track of the pivot rows chosen in the pivot array `p`.

  A successful LU factorization leaves the matrix `a` and the pivot array `p` with the following information:

  1. `p[k]` contains the row number of the pivot element chosen at the beginning of elimination step `k`, `k` $= 0, 1, ...,$`n`-1.
  2. If the unique LU factorization of `a` is given by $Pa = LU$, where P is a permutation matrix, L is a lower triangular matrix with all 1's on the diagonal, and U is an upper triangular matrix, then the upper triangular part of `a` (including its diagonal) contains U and the strictly lower triangular part of `a` contains the multipliers, $I - L$.

     `gefa` returns 0 if successful. Otherwise it encountered a zero diagonal element during the factorization. In this case it returns the column index (numbered from one) at which it encountered the zero.

- `gesl`

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

- `denzero`

  `denzero(a,n)` sets all the elements of the `n` by `n` dense matrix `a` to be 0.0;

- `dencopy`

  `dencopy(a,b,n)` copies the `n` by `n` dense matrix `a` into the `n` by `n` dense matrix `b`;

- `denscale`

  `denscale(c,a,n)` scales every element in the `n` by `n` dense matrix `a` by `c`;

- `denaddI`

  `denaddI(a,n)` increments the `n` by `n` dense matrix `a` by the identity matrix;

- `denfreepiv`

  `denfreepiv(p)` frees the pivot array `p` allocated by `denallocpiv`;

- `denfree`

  `denfree(a)` frees the dense matrix `a` allocated by `denalloc`;

- `denprint`

  `denprint(a,n)` prints the `n` by `n` dense matrix `a` to standard output as it would normally appear on paper. It is intended as a debugging tool with small values of `n`. The elements are printed using the `%g` option. A blank line is printed before and after the matrix.

8.2 The BAND module

8.2.1 Type BandMat

The type `BandMat` is the type of a large band matrix A (possibly distributed). It is defined to be a pointer to a structure defined by:

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

The fields in the above structure are:

- *size* is the number of columns (which is the same as the number of rows);
- *mu* is the upper half-bandwidth, $0 \leq mu \leq size-1$;
- *ml* is the lower half-bandwidth, $0 \leq ml \leq size-1$;
- *smu* is the storage upper half-bandwidth, $mu \leq smu \leq size-1$. The `BandFactor` routine writes the LU factors into the storage for A. The upper triangular factor U, however, may have an upper half-bandwidth as big as min(*size*−1,*mu*+*ml*) because of partial pivoting. The *smu* field holds the upper half-bandwidth allocated for A.
- *data* is a two dimensional array used for component storage. The elements of a band matrix of type `BandMat` are stored columnwise (i.e. columns are stored one on top of the other in memory). Only elements within the specified half-bandwidths are stored.

  If we number rows and columns in the band matrix starting from 0, then

  – *data[0]* is a pointer to (*smu*+*ml*+1)**size* contiguous locations which hold the elements within the band of A

  – *data[j]* is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from *smu*−*mu* (to access the uppermost element within the band in the j-th column) to *smu*+*ml* (to access the lowest element within the band in the j-th column). Indices from 0 to *smu*−*mu*−1 give access to extra storage elements required by `BandFactor`.

  – *data[j][i-j+smu]* is the (i,j)-th element, $j-mu \leq i \leq j+ml$.

The macros below allow a user to access individual matrix elements without writing out explicit data structure references and without knowing too much about the underlying element storage. The only storage assumption needed is that elements are stored columnwise and that a pointer into the j-th column of elements can be obtained via the `BAND_COL` macro. Users should use these macros whenever possible.

See Figure 8.1 for a diagram of the `BandMat` type.

8.2.2 Accessor Macros

The following three macros are defined by the BAND module to provide access to data in the `BandMat` type:

- `BAND_ELEM`

  Usage : `BAND_ELEM(A,i,j) = a_ij;` or `a_ij = BAND_ELEM(A,i,j);`

  `BAND_ELEM` references the (`i`,`j`)-th element of the $N \times N$ band matrix `A`, where $0 \leq$ `i`, `j` $\leq N-1$. The location (`i`,`j`) should further satisfy `j`−`(A->mu)` ≤ `i` ≤ `j`+`(A->ml)`.

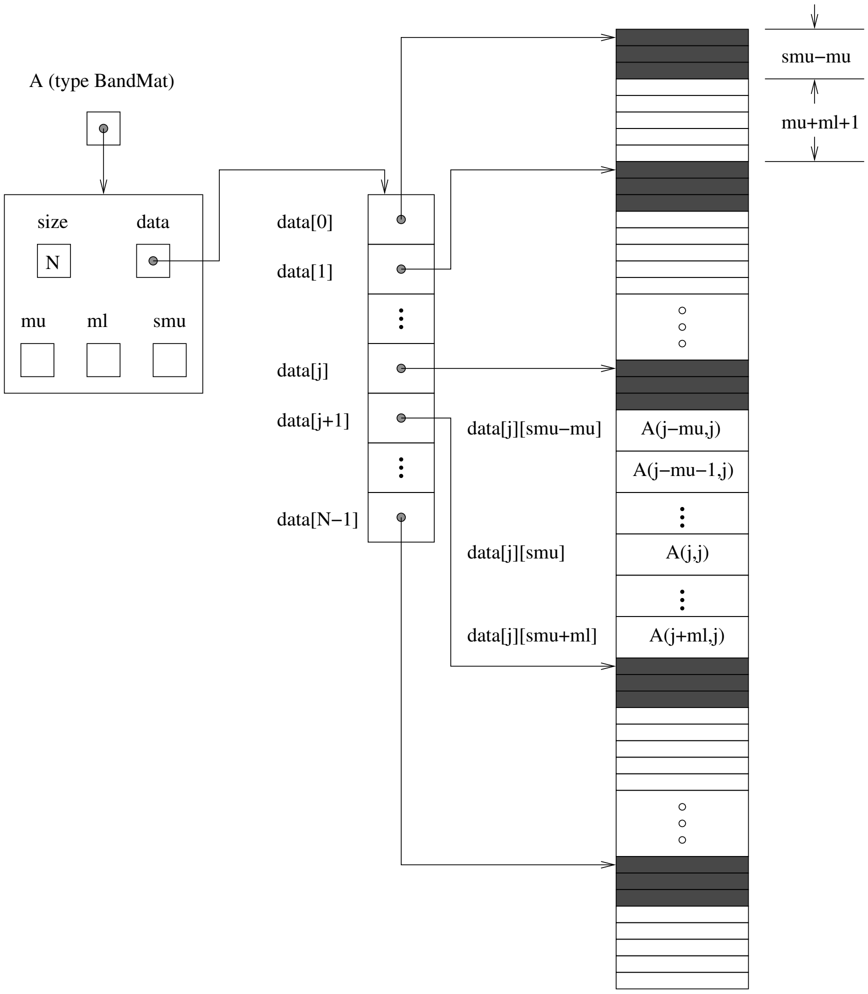


Figure 8.1: Diagram of the storage for a band matrix of type `BandMat`. Here `A` is an $N \times N$ band matrix of type `BandMat` with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to $N-1$ and the (i,j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the `BandFactor` and `BandBacksolve` routines.

- BAND_COL

  Usage : `col_j = BAND_COL(A,j);`

  BAND_COL references the diagonal element of the j-th column of the $N \times N$ band matrix A, $0 \leq$ j $\leq N-1$. The type of the expression BAND_COL(A,j) is `realtype *`. The pointer returned by the call BAND_COL(A,j) can be treated as an array which is indexed from $-$(A->mu) to (A->ml).

- BAND_COL_ELEM

  Usage : `BAND_COL_ELEM(col_j,i,j) = a_ij;` or `a_ij = BAND_COL_ELEM(col_j,i,j);`

  This macro references the (i,j)-th entry of the band matrix A when used in conjunction with BAND_COL to reference the j-th column through col_j. The index (i,j) should satisfy j$-$(A->mu) $\leq$ i $\leq$ j$+$(A->ml).

8.2.3 Functions

The following functions for BandMat matrices are available in the BAND package. For full details, see the header file band.h.

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

8.3 The SPGMR module

The SPGMR package, in the files spgmr.h and spgmr.c, includes an implementation of the scaled preconditioned GMRES method. A separate code module, iterative.h and iterative.c, contains auxiliary functions that support SPGMR, and also other Krylov solvers to be added later. For full details, including usage instructions, see the files spgmr.h and iterative.h.

Functions. The following functions are available in the SPGMR package:

- SpgmrMalloc: allocation of memory for SpgmrSolve;
- SpgmrSolve: solution of $Ax = b$ by the SPGMR method;
- SpgmrFree: free memory allocated by SpgmrMalloc.

The following functions are available in the support package iterative.h and iterative.c:

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

Chapter 9

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.

9.1 IDA input constants

IDA main solver module

`IDA_SS`	1	Scalar relative tolerance, scalar absolute tolerance.
`IDA_SV`	2	Scalar relative tolerance, vector absolute tolerance.
`IDA_NORMAL`	1	Solver returns at specified output time.
`IDA_ONE_STEP`	2	Solver returns after each successful step.
`IDA_NORMAL_TSTOP`	3	Solver returns at specified output time, but does not proceed past the specified stopping time.
`IDA_ONE_STEP_TSTOP`	4	Solver returns after each successful step, but does not proceed past the specified stopping time.
`IDA_YA_YDP_INIT`	1	Compute y_a and y'_d, given y_d.
`IDA_Y_INIT`	2	Compute y, given 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.

9.2 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_MEM_NULL`	-1	The `ida_mem` argument was `NULL`.
`IDA_ILL_INPUT`	-2	One of the function inputs is illegal.
`IDA_NO_MALLOC`	-3	The IDA memory was not allocated by a call to `IDAMalloc`.
`IDA_TOO_MUCH_WORK`	-4	The solver took `mxstep` internal steps but could not reach tout.

IDA_TOO_MUCH_ACC	-5	The solver could not satisfy the accuracy demanded by the user for some internal step.
IDA_ERR_FAIL	-6	Error test failures occurred too many times during one internal time step or minimum step size was reached.
IDA_CONV_FAIL	-7	Convergence test failures occurred too many times during one internal time step or minimum step size was reached.
IDA_LINIT_FAIL	-8	The linear solver's initialization function failed.
IDA_LSETUP_FAIL	-9	The linear solver's setup function failed in an unrecoverable manner.
IDA_LSOLVE_FAIL	-10	The linear solver's solve function failed in an unrecoverable manner.
IDA_RES_FAIL	-11	The user-provided residual function failed in an unrecoverable manner.
IDA_CONSTR_FAIL	-12	The inequality constraints were violated and the solver was unable to recover.
IDA_REP_RES_FAIL	-13	The user-provided residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.
IDA_MEM_FAIL	-14	A memory allocation failed.
IDA_BAD_T	-15	The time t s outside the last step taken.
IDA_BAD_EWT	-16	Zero value of some error weight component.
IDA_FIRST_RES_FAIL	-17	The user-provided residual function failed recoverably on the first call.
IDA_LINESEARCH_FAIL	-18	The line search failed.
IDA_NO_RECOVERY	-19	The residual function, linear solver setup function, or linear solver solve function had a recoverable failure, but IDACalcIC could not recover.
IDA_PDATA_NULL	-20	The preconditioner module has not been initialized.

IDADENSE linear solver module

IDADENSE_SUCCESS	0	Successful function return.
IDADENSE_MEM_NULL	-1	The ida_mem argument was NULL.
IDADENSE_LMEM_NULL	-2	The IDADENSE linear solver has not been initialized.
IDADENSE_ILL_INPUT	-3	The IDADENSE solver is not compatible with the current NVECTOR module.
IDADENSE_MEM_FAIL	-4	A memory allocation request failed.

IDABAND linear solver module

IDABAND_SUCCESS	0	Successful function return.
IDABAND_MEM_NULL	-1	The ida_mem argument was NULL.
IDABAND_LMEM_NULL	-2	The IDABAND linear solver has not been initialized.
IDABAND_ILL_INPUT	-3	The IDABAND solver is not compatible with the current NVECTOR module.
IDABAND_MEM_FAIL	-4	A memory allocation request failed.

IDASPGMR linear solver module

IDASPGMR_SUCCESS	0	Successful function return.
IDASPGMR_MEM_NULL	-1	The ida_mem argument was NULL.
IDASPGMR_LMEM_NULL	-2	The IDASPGMR linear solver has not been initialized.
IDASPGMR_ILL_INPUT	-3	The IDASPGMR solver is not compatible with the current NVECTOR module.

IDASPGMR_MEM_FAIL	-4	A memory allocation request failed.

SPGMR generic linear solver module

SPGMR_SUCCESS	0	Converged.
SPGMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPGMR_CONV_FAIL	2	Failure to converge.
SPGMR_QRFACT_FAIL	3	A singular matrix was found during the QR factorization.
SPGMR_PSOLVE_FAIL_REC	4	The preconditioner solve function failed recoverably.
SPGMR_MEM_NULL	-1	The SPGMR memory is NULL
SPGMR_ATIMES_FAIL	-2	The Jacobian tims vector function failed.
SPGMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPGMR_GS_FAIL	-4	Failure in the Gram-Schmidt procedure.
SPGMR_QRSOL_FAIL	-5	The matrix R was found to be singular during the QR solve phase.

Bibliography

[1] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM, Philadelphia, Pa, 1996.

[2] P. N. Brown and A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.

[3] 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.

[4] 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.

[5] G. D. Byrne. Pragmatic Experiments with Krylov Methods in the Stiff ODE Setting. In J.R. Cash and I. Gladwell, editors, *Computational Ordinary Differential Equations*, pages 323–356, Oxford, 1992. Oxford University Press.

[6] G. D. Byrne and A. C. Hindmarsh. User Documentation for PVODE, An ODE Solver for Parallel Computers. Technical Report UCRL-ID-130884, LLNL, May 1998.

[7] G. D. Byrne and A. C. Hindmarsh. PVODE, An ODE Solver for Parallel Computers. *Intl. J. High Perf. Comput. Apps.*, 13(4):254–365, 1999.

[8] S. D. Cohen and A. C. Hindmarsh. CVODE, a Stiff/Nonstiff ODE Solver in C. *Computers in Physics*, 10(2):138–143, 1996.

[9] A. M. Collier, A. C. Hindmarsh, R. Serban, and C.S. Woodward. User Documentation for KINSOL v2.2.0. Technical Report UCRL-SM-208116, LLNL, 2004.

[10] A. C. Hindmarsh and R. Serban. Example Programs for IDA v2.2.0. Technical Report UCRL-SM-208113, LLNL, 2004.

[11] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v2.2.0. Technical Report UCRL-SM-208108, LLNL, 2004.

[12] 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.

[13] 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.

Index