User Documentation for IDAS v1.2.2 (SUNDIALS v2.6.2)

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

July 30, 2015

UCRL-SM-234051

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

Contents

List of Tables

List of Figures

Chapter 1

Introduction

IDAS is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [19]. This suite consists of CVODE, ARKODE, KINSOL, and IDA, and variants of these with sensitivity analysis capabilities, CVODES and IDAS.

IDAS is a general purpose solver for the initial value problem (IVP) for systems of differential-algebraic equations (DAEs). The name IDAS stands for Implicit Differential-Algebraic solver with Sensitivity capabilities. IDAS is an extension of the IDA solver within SUNDIALS, itself based on DASPK [5, 6]; however, like all SUNDIALS solvers, IDAS is written in ANSI-standard C rather than FORTRAN77. Its most notable features are that, (1) in the solution of the underlying nonlinear system at each time step, it offers a choice of Newton/direct methods and a choice of Inexact Newton/Krylov (iterative) methods; (2) it is written in a *data-independent* manner in that it acts on generic vectors without any assumptions on the underlying organization of the data; and (3) it provides a flexible, extensible framework for sensitivity analysis, using either *forward* or *adjoint* methods. Thus IDAS shares significant modules previously written within CASC at LLNL to support the ordinary differential equation (ODE) solvers CVODE [20, 12] and PVODE [8, 9], the DAE solver IDA [22] on which IDAS is based, the sensitivity-enabled ODE solver CVODES [21, 29], and also the nonlinear system solver KINSOL [13].

The Newton/Krylov methods in IDAS are: the GMRES (Generalized Minimal RESidual) [28], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [31], and TFQMR (Transpose-Free Quasi-Minimal Residual) linear iterative methods [17]. As Krylov methods, these require almost no matrix storage for solving the Newton equations as compared to direct methods. However, the algorithms allow for a user-supplied preconditioner matrix, and for most problems preconditioning is essential for an efficient solution.

For very large DAE systems, the Krylov methods are preferable over direct linear solver methods, and are often the only feasible choice. Among the three Krylov methods in IDAS, we recommend GMRES as the best overall choice. However, users are encouraged to compare all three, especially if encountering convergence failures with GMRES. Bi-CGFStab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for GMRES depends on the desired Krylov subspace size.

IDAS is written with a functionality that is a superset of that of IDA. Sensitivity analysis capabilities, both forward and adjoint, have been added to the main integrator. Enabling forward sensitivity computations in IDAS will result in the code integrating the so-called *sensitivity equations* simultaneously with the original IVP, yielding both the solution and its sensitivity with respect to parameters in the model. Adjoint sensitivity analysis, most useful when the gradients of relatively few functionals of the solution with respect to many parameters are sought, involves integration of the original IVP forward in time followed by the integration of the so-called *adjoint equations* backward in time. IDAS provides the infrastructure needed to integrate any final-condition ODE dependent on the solution of the original IVP (in particular the adjoint system).

There are several motivations for choosing the C language for IDAS. First, a general movement away from FORTRAN and toward C in scientific computing was apparent. Second, the pointer, structure, and dynamic memory allocation features in C are extremely useful in software of this complexity, with the great variety of method options offered. Finally, we prefer C over C++ for IDAS 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 v1.2.0

Two major additions were made to the linear system solvers that are available for use with the IDAS solver. First, in the serial case, an interface to the sparse direct solver KLU was added. Second, an interface to SuperLU_MT, the multi-threaded version of SuperLU, was added as a thread-parallel sparse direct solver option, to be used with the serial version of the NVECTOR module. As part of these additions, a sparse matrix (CSC format) structure was added to IDAS.

Otherwise, only relatively minor modifications were made to IDAS:

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

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

An option was added in the case of Adjoint Sensitivity Analysis with dense or banded Jacobian: With a call to `IDADlsSetDenseJacFnBS` or `IDADlsSetBandJacFnBS`, the user can specify a user-supplied Jacobian function of type `IDADls***JacFnBS`, for the case where the backward problem depends on the forward sensitivities.

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

For the Adjoint Sensitivity Analysis case in which the backward problem depends on the forward sensitivities, options have been added to allow for user-supplied `pset`, `psolve`, and `jtimes` functions.

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

In the User Guide, a paragraph was added in Section 6.2.1 on `IDAAdjReInit`, and a paragraph was added in Section 6.2.9 on `IDAGetAdjY`.

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

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

Changes in v1.1.0

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

Errors in the logic for the integration of backward problems were identified and fixed.

A large number of minor errors have been fixed. Among these are the following: A missing vector pointer setting was added in `IDASensLineSrch`. In `IDACompleteStep`, conditionals around lines loading a new column of three auxiliary divided difference arrays, for a possible order increase, were fixed. After the solver memory is created, it is set to zero before being filled. In each linear solver interface function, the linear solver memory is freed on an error return, and the `**Free` function now includes a line setting to NULL the main memory pointer to the linear solver memory. A memory leak was fixed in two of the `IDASp***Free` functions. In the rootfinding functions `IDARcheck1/IDARcheck2`, when an exact zero is found, the array `glo` of g values at the left endpoint is adjusted, instead of shifting the t location `tlo` slightly. In the installation files, we modified the treatment of the macro SUNDIALS_USE_GENERIC_MATH, so that the parameter GENERIC_MATH_LIB is either defined (with no value) or not defined.

1.2 Reading this User Guide

The structure of this document is as follows:

- In Chapter 2, we give short descriptions of the numerical methods implemented by IDAS for the solution of initial value problems for systems of DAEs, continue with short descriptions of preconditioning (§2.2) and rootfinding (§2.3), and then give an overview of the mathematical aspects of sensitivity analysis, both forward (§2.5) and adjoint (§2.6).
- The following chapter describes the structure of the SUNDIALS suite of solvers (§3.1) and the software organization of the IDAS solver (§3.2).
- Chapter 4 is the main usage document for IDAS for simulation applications. It includes a complete description of the user interface for the integration of DAE initial value problems. Readers that are not interested in using IDAS for sensitivity analysis can then skip the next two chapters.
- Chapter 5 describes the usage of IDAS for forward sensitivity analysis as an extension of its IVP integration capabilities. We begin with a skeleton of the user main program, with emphasis on the steps that are required in addition to those already described in Chapter 4. Following that we provide detailed descriptions of the user-callable interface routines specific to forward sensitivity analysis and of the additonal optional user-defined routines.
- Chapter 6 describes the usage of IDAS for adjoint sensitivity analysis. We begin by describing the IDAS checkpointing implementation for interpolation of the original IVP solution during integration of the adjoint system backward in time, and with an overview of a user's main program. Following that we provide complete descriptions of the user-callable interface routines for adjoint sensitivity analysis as well as descriptions of the required additional user-defined routines.
- Chapter 7 gives a brief overview of the generic NVECTOR module shared amongst the various components of SUNDIALS, as well as details on the NVECTOR implementations provided with SUNDIALS: a serial implementation (§7.1), a distributed memory parallel implementation based on MPI (§7.2), and two thread-parallel implementations based on openMP (§7.3) and Pthreads (§7.4), respectively.
- Chapter 8 describes the specifications of linear solver modules as supplied by the user.
- Chapter 9 describes in detail the generic linear solvers shared by all SUNDIALS solvers.
- Finally, in the appendices, we provide detailed instructions for the installation of IDAS, within the structure of SUNDIALS (Appendix A), as well as a list of all the constants used for input to and output from IDAS functions (Appendix B).

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

Chapter 2

Mathematical Considerations

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

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

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

Additionally, if (2.1) depends on some parameters $p \in \mathbf{R}^{N_p}$, i.e.

$$\begin{aligned} &F(t, y, \dot{y}, p) = 0 \\ &y(t_0) = y_0(p)\,,\ \dot{y}(t_0) = \dot{y}_0(p)\,, \end{aligned} \tag{2.2}$$

IDAS can also compute first order derivative information, performing either *forward sensitivity analysis* or *adjoint sensitivity analysis*. In the first case, IDAS computes the sensitivities of the solution with respect to the parameters p, while in the second case, IDAS computes the gradient of a *derived function* with respect to the parameters p.

2.1 IVP solution

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

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

$$\sum_{i=0}^{q} \alpha_{n,i} y_{n-i} = h_n \dot{y}_n\,, \tag{2.3}$$

where y_n and $\dot{y}_n$ are the computed approximations to $y(t_n)$ and $\dot{y}(t_n)$, respectively, and the step size is $h_n = t_n - t_{n-1}$. The coefficients $\alpha_{n,i}$ are uniquely determined by the order q, and the history of the step sizes. The application of the BDF (2.3) to the DAE system (2.1) results in a nonlinear algebraic system to be solved at each step:

$$G(y_n) \equiv F\left(t_n,\, y_n,\, h_n^{-1}\sum_{i=0}^{q}\alpha_{n,i}y_{n-i}\right) = 0\,. \tag{2.4}$$

Regardless of the method options, the solution of the nonlinear system (2.4) is accomplished with some form of Newton iteration. This leads to a linear system for each Newton correction, of the form

$$J[y_{n(m+1)} - y_{n(m)}] = -G(y_{n(m)})\,, \tag{2.5}$$

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

$$J = \frac{\partial G}{\partial y} = \frac{\partial F}{\partial y} + \alpha\frac{\partial F}{\partial \dot{y}}\,, \tag{2.6}$$

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

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

- dense direct solvers, using either an internal implementation or a Blas/Lapack implementation (serial or threaded vector modules only),
- band direct solvers, using either an internal implementation or a Blas/Lapack implementation (serial or threaded vector modules only),
- sparse direct solver interfaces, using either the KLU sparse solver library [14, 1], or the thread-enabled SuperLU_MT sparse solver library [25, 15, 2] (serial or threaded vector modules only) [Note that users will need to download and install the KLU or SuperLU_MT packages independent of IDAS],
- SPGMR, a scaled preconditioned GMRES (Generalized Minimal Residual method) solver without restarts,
- SPBCG, a scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method) solver, or
- SPTFQMR, a scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method) solver.

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

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

$$W_i = 1/[\text{RTOL}\cdot|y_i| + \text{ATOL}_i]\,. \tag{2.7}$$

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

In the case of a direct linear solver (dense, band, or sparse), the nonlinear iteration (2.5) is a Modified Newton iteration, in that the Jacobian J is fixed (and usually out of date), with a coefficient $\bar{\alpha}$ in place of α in J. When using one of the Krylov methods SPGMR, SPBCG, or SPTFQMR as the linear solver, the iteration is an Inexact Newton iteration, using the current Jacobian (through matrix-free products Jv), in which the linear residual $J\Delta y + G$ is nonzero but controlled. The Jacobian matrix J (direct cases) or preconditioner matrix P (SPGMR/SPBCG/SPTFQMR case) is updated when:

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

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

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

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

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

$$S \|\delta_m\| < 0.33 \,, \tag{2.8}$$

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 (2.8) uses an old value for S. Therefore, at the first Newton iteration, we make an additional test and stop the iteration if $\|\delta_1\| < 0.33 \cdot 10^{-4}$ (since such a δ_1 is probably just noise and therefore not appropriate for use in evaluating R). We allow only a small number (default value 4) of Newton iterations. If convergence fails with J or P current, we are forced to reduce the step size h_n, and we replace h_n by $h_n/4$. The integration is halted after a preset number (default value 10) of convergence failures. Both the maximum allowable Newton iterations and the maximum nonlinear convergence failures can be changed by the user from their default values.

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

In the direct linear solver cases, the Jacobian J defined in (2.6) can be either supplied by the user or have IDAS compute one internally by difference quotients. In the latter case, we use the approximation

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

where U is the unit roundoff, h is the current step size, and W_j is the error weight for the component y_j defined by (2.7). In the SPGMR/SPBCG/SPTFQMR case, if a routine for Jv is not supplied, such products are approximated by

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

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

We note that with the sparse direct solvers, the Jacobian *must* be supplied by a user routine in compressed-sparse-column format.

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

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

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

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

and so the norm of LTE is estimated as $|C| \cdot \|\Delta_n\|$. In addition, IDAS 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 IDAS is

$$\max\{|C|, \bar{C}\}\|\Delta_n\| \leq 1 \,. \tag{2.9}$$

A user option is available by which the algebraic components of the error vector are omitted from the test (2.9), if these have been so identified.

In IDAS, the local error test is tightly coupled with the logic for selecting the step size and order. First, there is an initial phase that is treated specially; for the first few steps, the step size is doubled and the order raised (from its initial value of 1) on every step, until (a) the local error test (2.9) 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, IDAS uses a different set of local error estimates, based on the asymptotic behavior of the local error in the case of fixed step sizes. At each of the orders q' equal to q, $q-1$ (if $q>1$), $q-2$ (if $q>2$), or $q+1$ (if $q<5$), there are constants $C(q')$ such that the norm of the local truncation error at order q' satisfies

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

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

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

The step/order selection begins with a test for monotonicity that is made even *before* the local error test is performed. Namely, the order is reset to $q' = q-1$ if (a) $q=2$ and $T(1) \leq T(2)/2$, or (b) $q>2$ and $\max\{T(q-1), T(q-2)\} \leq T(q)$; otherwise $q' = q$. Next the local error test (2.9) is performed, and if it fails, the step is redone at order $q \leftarrow q'$ and a new step size h'. The latter is based on the h^{q+1} asymptotic behavior of $\text{ELTE}(q)$, and, with safety factors, is given by

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

The value of η is adjusted so that $0.25 \leq \eta \leq 0.9$ before setting $h \leftarrow h' = \eta h$. If the local error test fails a second time, IDAS uses $\eta = 0.25$, and on the third and subsequent failures it uses $q=1$ and $\eta = 0.25$. After 10 failures, IDAS 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, IDAS 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 [3] for details.

IDAS 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, IDAS 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, IDAS 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 IDAS not to integrate past a given stopping point $t = t_{\mathrm{stop}}$.

2.2 Preconditioning

When using a Newton method to solve the nonlinear system (2.5), IDAS makes repeated use of a linear solver to solve linear systems of the form $J\Delta y = -G$. If this linear system solve is done with one of the scaled preconditioned iterative linear solvers, these solvers are rarely successful if used without preconditioning; it is generally necessary to precondition the system in order to obtain acceptable efficiency. A system $Ax = b$ can be preconditioned on the left, on the right, or on both sides. The Krylov method is then applied to a system with the matrix $P^{-1}A$, or AP^{-1}, or $P_L^{-1}AP_R^{-1}$, instead of A. However, within IDAS, preconditioning is allowed *only* on the left, so that the iterative method is applied to systems $(P^{-1}J)\Delta y = -P^{-1}G$. Left preconditioning is required to make the norm of the linear residual in the Newton iteration meaningful; in general, $\|J\Delta y + G\|$ is meaningless, since the weights used in the WRMS-norm correspond to y.

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

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

2.3 Rootfinding

The IDAS solver has been augmented to include a rootfinding feature. This means that, while integrating the Initial Value Problem (2.1), IDAS can also find the roots of a set of user-defined functions

$g_i(t, y, \dot{y})$ that depend on t, the solution vector $y = y(t)$, and its $t-$derivative $\dot{y}(t)$. The number of these root functions is arbitrary, and if more than one g_i is found to have a root in any given interval, the various root locations are found and reported in the order that they occur on the t axis, in the direction of integration.

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

The basic scheme used is to check for sign changes of any $g_i(t)$ over each time step taken, and then (when a sign change is found) to home in on the root (or roots) with a modified secant method [18]. In addition, each time g is computed, IDAS checks to see if $g_i(t) = 0$ exactly, and if so it reports this as a root. However, if an exact zero of any g_i is found at a point t, IDAS computes g at $t + \delta$ for a small increment δ, slightly further in the direction of integration, and if any $g_i(t + \delta) = 0$ also, IDAS stops and reports an error. This way, each time IDAS takes a time step, it is guaranteed that the values of all g_i are nonzero at some past value of t, beyond which a search for roots is to be done.

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

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

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

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

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

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

2.4 Pure quadrature integration

In many applications, and most notably during the backward integration phase of an adjoint sensitivity analysis run (see §2.6) it is of interest to compute integral quantities of the form

$$z(t) = \int_{t_0}^{t} q(\tau, y(\tau), \dot{y}(\tau), p) \, d\tau \; . \tag{2.10}$$

The most effective approach to compute $z(t)$ is to extend the original problem with the additional ODEs (obtained by applying Leibnitz's differentiation rule):

$$\dot{z} = q(t, y, \dot{y}, p)\,, \quad z(t_0) = 0\,. \tag{2.11}$$

Note that this is equivalent to using a quadrature method based on the underlying linear multistep polynomial representation for $y(t)$.

This can be done at the "user level" by simply exposing to IDAS the extended DAE system (2.2)+(2.10). However, in the context of an implicit integration solver, this approach is not desirable since the nonlinear solver module will require the Jacobian (or Jacobian-vector product) of this extended DAE. Moreover, since the additional states z do not enter the right-hand side of the ODE (2.10) and therefore the residual of the extended DAE system does not depend on z, it is much more efficient to treat the ODE system (2.10) separately from the original DAE system (2.2) by "taking out" the additional states z from the nonlinear system (2.4) that must be solved in the correction step of the LMM. Instead, "corrected" values z_n are computed explicitly as

$$z_n = \frac{1}{\alpha_{n,0}} \left(h_n q(t_n, y_n, \dot{y}_n, p) - \sum_{i=1}^{q} \alpha_{n,i} z_{n-i} \right) \,,$$

once the new approximation y_n is available.

The quadrature variables z can be optionally included in the error test, in which case corresponding relative and absolute tolerances must be provided.

2.5 Forward sensitivity analysis

Typically, the governing equations of complex, large-scale models depend on various parameters, through the right-hand side vector and/or through the vector of initial conditions, as in (2.2). In addition to numerically solving the DAEs, it may be desirable to determine the sensitivity of the results with respect to the model parameters. Such sensitivity information can be used to estimate which parameters are most influential in affecting the behavior of the simulation or to evaluate optimization gradients (in the setting of dynamic optimization, parameter estimation, optimal control, etc.).

The *solution sensitivity* with respect to the model parameter p_i is defined as the vector $s_i(t) = \partial y(t)/\partial p_i$ and satisfies the following *forward sensitivity equations* (or *sensitivity equations* for short):

$$\begin{aligned} & \frac{\partial F}{\partial y} s_i + \frac{\partial F}{\partial \dot{y}} \dot{s}_i + \frac{\partial F}{\partial p_i} = 0 \\ & s_i(t_0) = \frac{\partial y_0(p)}{\partial p_i}\,, \; \dot{s}_i(t_0) = \frac{\partial \dot{y}_0(p)}{\partial p_i}\,, \end{aligned} \tag{2.12}$$

obtained by applying the chain rule of differentiation to the original DAEs (2.2).

When performing forward sensitivity analysis, IDAS carries out the time integration of the combined system, (2.2) and (2.12), by viewing it as a DAE system of size $N(N_s + 1)$, where N_s is the number of model parameters p_i, with respect to which sensitivities are desired ($N_s \le N_p$). However, major improvements in efficiency can be made by taking advantage of the special form of the sensitivity equations as linearizations of the original DAEs. In particular, the original DAE system and all sensitivity systems share the same Jacobian matrix J in (2.6).

The sensitivity equations are solved with the same linear multistep formula that was selected for the original DAEs and the same linear solver is used in the correction phase for both state and sensitivity variables. In addition, IDAS offers the option of including (*full error control*) or excluding (*partial error control*) the sensitivity variables from the local error test.

2.5.1 Forward sensitivity methods

In what follows we briefly describe three methods that have been proposed for the solution of the combined DAE and sensitivity system for the vector $\hat{y} = [y, s_1, \ldots, s_{N_s}]$.

- *Staggered Direct* In this approach [11], the nonlinear system (2.4) is first solved and, once an acceptable numerical solution is obtained, the sensitivity variables at the new step are found by directly solving (2.12) after the BDF discretization is used to eliminate $\dot{s}_i$. Although the system matrix of the above linear system is based on exactly the same information as the matrix J in (2.6), it must be updated and factored at every step of the integration, in contrast to an evaluation of J which is updated only occasionally. For problems with many parameters (relative to the problem size), the staggered direct method can outperform the methods described below [24]. However, the computational cost associated with matrix updates and factorizations makes this method unattractive for problems with many more states than parameters (such as those arising from semidiscretization of PDEs) and is therefore not implemented in IDAS.

- *Simultaneous Corrector* In this method [26], the discretization is applied simultaneously to both the original equations (2.2) and the sensitivity systems (2.12) resulting in an "extended" nonlinear system $\hat{G}(\hat{y}_n) = 0$ where $\hat{y}_n = [y_n, \ldots, s_i, \ldots]$. This combined nonlinear system can be solved using a modified Newton method as in (2.5) by solving the corrector equation
$$\hat{J}[\hat{y}_{n(m+1)} - \hat{y}_{n(m)}] = -\hat{G}(\hat{y}_{n(m)}) \tag{2.13}$$
at each iteration, where
$$\hat{J} = \begin{bmatrix} J & & & & \\ J_1 & J & & & \\ J_2 & 0 & J & & \\ \vdots & \vdots & \ddots & \ddots & \\ J_{N_s} & 0 & \ldots & 0 & J \end{bmatrix},$$
J is defined as in (2.6), and $J_i = (\partial/\partial y)\left[F_y s_i + F_{\dot{y}} \dot{s}_i + F_{p_i}\right]$. It can be shown that 2-step quadratic convergence can be retained by using only the block-diagonal portion of $\hat{J}$ in the corrector equation (2.13). This results in a decoupling that allows the reuse of J without additional matrix factorizations. However, the sum $F_y s_i + F_{\dot{y}} \dot{s}_i + F_{p_i}$ must still be reevaluated at each step of the iterative process (2.13) to update the sensitivity portions of the residual $\hat{G}$.

- *Staggered corrector* In this approach [16], as in the staggered direct method, the nonlinear system (2.4) is solved first using the Newton iteration (2.5). Then, for each sensitivity vector $\xi \equiv s_i$, a separate Newton iteration is used to solve the sensitivity system (2.12):
$$\begin{aligned} &J[\xi_{n(m+1)} - \xi_{n(m)}] = \\ &- \left[F_y(t_n, y_n, \dot{y}_n)\xi_{n(m)} + F_{\dot{y}}(t_n, y_n, \dot{y}_n) \cdot h_n^{-1} \left(\alpha_{n,0}\xi_{n(m)} + \sum_{i=1}^{q} \alpha_{n,i}\xi_{n-i} \right) + F_{p_i}(t_n, y_n, \dot{y}_n) \right]. \end{aligned} \tag{2.14}$$
In other words, a modified Newton iteration is used to solve a linear system. In this approach, the matrices $\partial F/\partial y$, $\partial F/\partial \dot{y}$ and vectors $\partial F/\partial p_i$ need be updated only once per integration step, after the state correction phase (2.5) has converged.

IDAS implements both the simultaneous corrector method and the staggered corrector method.

An important observation is that the staggered corrector method, combined with a Krylov linear solver, effectively results in a staggered direct method. Indeed, the Krylov solver requires only the action of the matrix J on a vector and this can be provided with the current Jacobian information. Therefore, the modified Newton procedure (2.14) will theoretically converge after one iteration.

2.5.2 Selection of the absolute tolerances for sensitivity variables

If the sensitivities are included in the error test, IDAS provides an automated estimation of absolute tolerances for the sensitivity variables based on the absolute tolerance for the corresponding state variable. The relative tolerance for sensitivity variables is set to be the same as for the state variables.

The selection of absolute tolerances for the sensitivity variables is based on the observation that the sensitivity vector s_i will have units of $[y]/[p_i]$. With this, the absolute tolerance for the j-th component of the sensitivity vector s_i is set to $\text{ATOL}_j/|\bar{p}_i|$, where ATOL_j are the absolute tolerances for the state variables and $\bar{p}$ is a vector of scaling factors that are dimensionally consistent with the model parameters p and give an indication of their order of magnitude. This choice of relative and absolute tolerances is equivalent to requiring that the weighted root-mean-square norm of the sensitivity vector s_i with weights based on s_i be the same as the weighted root-mean-square norm of the vector of scaled sensitivities $\bar{s}_i = |\bar{p}_i| s_i$ with weights based on the state variables (the scaled sensitivities $\bar{s}_i$ being dimensionally consistent with the state variables). However, this choice of tolerances for the s_i may be a poor one, and the user of IDAS can provide different values as an option.

2.5.3 Evaluation of the sensitivity right-hand side

There are several methods for evaluating the residual functions in the sensitivity systems (2.12): analytic evaluation, automatic differentiation, complex-step approximation, and finite differences (or directional derivatives). IDAS provides all the software hooks for implementing interfaces to automatic differentiation (AD) or complex-step approximation; future versions will include a generic interface to AD-generated functions. At the present time, besides the option for analytical sensitivity right-hand sides (user-provided), IDAS can evaluate these quantities using various finite difference-based approximations to evaluate the terms $(\partial F/\partial y)s_i + (\partial F/\partial \dot{y})\dot{s}_i$ and $(\partial F/\partial p_i)$, or using directional derivatives to evaluate $[(\partial F/\partial y)s_i + (\partial F/\partial \dot{y})\dot{s}_i + (\partial F/\partial p_i)]$. As is typical for finite differences, the proper choice of perturbations is a delicate matter. IDAS takes into account several problem-related features: the relative DAE error tolerance RTOL, the machine unit roundoff U, the scale factor $\bar{p}_i$, and the weighted root-mean-square norm of the sensitivity vector s_i.

Using central finite differences as an example, the two terms $(\partial F/\partial y)s_i + (\partial F/\partial \dot{y})\dot{s}_i$ and $\partial F/\partial p_i$ in (2.12) can be evaluated either separately:

$$\frac{\partial F}{\partial y}s_i + \frac{\partial F}{\partial \dot{y}}\dot{s}_i \approx \frac{F(t, y+\sigma_y s_i, \dot{y}+\sigma_y \dot{s}_i, p) - F(t, y-\sigma_y s_i, \dot{y}-\sigma_y \dot{s}_i, p)}{2\,\sigma_y}\,, \tag{2.15}$$

$$\frac{\partial F}{\partial p_i} \approx \frac{F(t, y, \dot{y}, p+\sigma_i e_i) - F(t, y, \dot{y}, p-\sigma_i e_i)}{2\,\sigma_i}\,, \tag{2.15'}$$

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

or simultaneously:

$$\frac{\partial F}{\partial y}s_i + \frac{\partial F}{\partial \dot{y}}\dot{s}_i + \frac{\partial F}{\partial p_i} \approx \frac{F(t, y+\sigma s_i, \dot{y}+\sigma \dot{s}_i, p+\sigma e_i) - F(t, y-\sigma s_i, \dot{y}-\sigma \dot{s}_i, p-\sigma e_i)}{2\,\sigma}\,, \tag{2.16}$$

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

or by adaptively switching between (2.15)+(2.15') and (2.16), depending on the relative size of the two finite difference increments σ_i and σ_y. In the adaptive scheme, if $\rho = \max(\sigma_i/\sigma_y, \sigma_y/\sigma_i)$, we use separate evaluations if $\rho > \rho_{\max}$ (an input value), and simultaneous evaluations otherwise.

These procedures for choosing the perturbations (σ_i, σ_y, σ) and switching between derivative formulas have also been implemented for one-sided difference formulas. Forward finite differences can be applied to $(\partial F/\partial y)s_i + (\partial F/\partial \dot{y})\dot{s}_i$ and $\frac{\partial F}{\partial p_i}$ separately, or the single directional derivative formula

$$\frac{\partial F}{\partial y}s_i + \frac{\partial F}{\partial \dot{y}}\dot{s}_i + \frac{\partial F}{\partial p_i} \approx \frac{F(t, y+\sigma s_i, \dot{y}+\sigma \dot{s}_i, p+\sigma e_i) - F(t, y, \dot{y}, p)}{\sigma}$$

can be used. In IDAS, the default value of $\rho_{\max} = 0$ indicates the use of the second-order centered directional derivative formula (2.16) exclusively. Otherwise, the magnitude of $\rho_{\max}$ and its sign (positive or negative) indicates whether this switching is done with regard to (centered or forward) finite differences, respectively.

2.5.4 Quadratures depending on forward sensitivities

If pure quadrature variables are also included in the problem definition (see §2.4), IDAS does *not* carry their sensitivities automatically. Instead, we provide a more general feature through which integrals depending on both the states y of (2.2) and the state sensitivities s_i of (2.12) can be evaluated. In other words, IDAS provides support for computing integrals of the form:

$$\bar{z}(t) = \int_{t_0}^{t} \bar{q}(\tau, y(\tau), \dot{y}(\tau), s_1(\tau), \ldots, s_{N_p}(\tau), p)\, d\tau\,.$$

If the sensitivities of the quadrature variables z of (2.10) are desired, these can then be computed by using:

$$\bar{q}_i = q_y s_i + q_{\dot{y}} \dot{s}_i + q_{p_i}\,, \quad i = 1, \ldots, N_p\,,$$

as integrands for $\bar{z}$, where q_y, $q_{\dot{y}}$, and q_p are the partial derivatives of the integrand function q of (2.10).

As with the quadrature variables z, the new variables $\bar{z}$ are also excluded from any nonlinear solver phase and "corrected" values $\bar{z}_n$ are obtained through explicit formulas.

2.6 Adjoint sensitivity analysis

In the *forward sensitivity approach* described in the previous section, obtaining sensitivities with respect to N_s parameters is roughly equivalent to solving an DAE system of size $(1 + N_s)N$. This can become prohibitively expensive, especially for large-scale problems, if sensitivities with respect to many parameters are desired. In this situation, the *adjoint sensitivity method* is a very attractive alternative, provided that we do not need the solution sensitivities s_i, but rather the gradients with respect to model parameters of a relatively few derived functionals of the solution. In other words, if $y(t)$ is the solution of (2.2), we wish to evaluate the gradient dG/dp of

$$G(p) = \int_{t_0}^{T} g(t, y, p) dt\,, \tag{2.17}$$

or, alternatively, the gradient dg/dp of the function $g(t, y, p)$ at the final time $t = T$. The function g must be smooth enough that $\partial g/\partial y$ and $\partial g/\partial p$ exist and are bounded.

In what follows, we only sketch the analysis for the sensitivity problem for both G and g. For details on the derivation see [10].

2.6.1 Sensitivity of $G(p)$

We focus first on solving the sensitivity problem for $G(p)$ defined by (2.17). Introducing a Lagrange multiplier λ, we form the augmented objective function

$$I(p) = G(p) - \int_{t_0}^{T} \lambda^* F(t, y, \dot{y}, p) dt.$$

Since $F(t, y, \dot{y}, p) = 0$, the sensitivity of G with respect to p is

$$\frac{dG}{dp} = \frac{dI}{dp} = \int_{t_0}^{T} (g_p + g_y y_p) dt - \int_{t_0}^{T} \lambda^* (F_p + F_y y_p + F_{\dot{y}} \dot{y}_p) dt, \tag{2.18}$$

where subscripts on functions such as F or g are used to denote partial derivatives. By integration by parts, we have

$$\int_{t_0}^{T} \lambda^* F_{\dot{y}} \dot{y}_p dt = (\lambda^* F_{\dot{y}} y_p)|_{t_0}^{T} - \int_{t_0}^{T} (\lambda^* F_{\dot{y}})' y_p dt,$$

where $(\cdots)'$ denotes the $t-$derivative. Thus equation (2.18) becomes

$$\frac{dG}{dp} = \int_{t_0}^{T} (g_p - \lambda^* F_p)\, dt - \int_{t_0}^{T} \left[-g_y + \lambda^* F_y - (\lambda^* F_{\dot y})' \right] y_p dt - (\lambda^* F_{\dot y} y_p)|_{t_0}^{T}. \tag{2.19}$$

Now by requiring λ to satisfy

$$(\lambda^* F_{\dot y})' - \lambda^* F_y = -g_y, \tag{2.20}$$

we obtain

$$\frac{dG}{dp} = \int_{t_0}^{T} (g_p - \lambda^* F_p)\, dt - (\lambda^* F_{\dot y} y_p)|_{t_0}^{T}. \tag{2.21}$$

Note that y_p at $t = t_0$ is the sensitivity of the initial conditions with respect to p, which is easily obtained. To find the initial conditions (at $t = T$) for the adjoint system, we must take into consideration the structure of the DAE system.

For index-0 and index-1 DAE systems, we can simply take

$$\lambda^* F_{\dot y}|_{t=T} = 0, \tag{2.22}$$

yielding the sensitivity equation for dG/dp

$$\frac{dG}{dp} = \int_{t_0}^{T} (g_p - \lambda^* F_p)\, dt + (\lambda^* F_{\dot y} y_p)|_{t=t_0}. \tag{2.23}$$

This choice will not suffice for a Hessenberg index-2 DAE system. For a derivation of proper final conditions in such cases, see [10].

The first thing to notice about the adjoint system (2.20) is that there is no explicit specification of the parameters p; this implies that, once the solution λ is found, the formula (2.21) can then be used to find the gradient of G with respect to any of the parameters p. The second important remark is that the adjoint system (2.20) is a terminal value problem which depends on the solution $y(t)$ of the original IVP (2.2). Therefore, a procedure is needed for providing the states y obtained during a forward integration phase of (2.2) to IDAS during the backward integration phase of (2.20). The approach adopted in IDAS, based on *checkpointing*, is described in §2.6.3 below.

2.6.2 Sensitivity of $g(T, p)$

Now let us consider the computation of $dg/dp(T)$. From $dg/dp(T) = (d/dT)(dG/dp)$ and equation (2.21), we have

$$\frac{dg}{dp} = (g_p - \lambda^* F_p)(T) - \int_{t_0}^{T} \lambda_T^* F_p dt + (\lambda_T^* F_{\dot y} y_p)|_{t=t_0} - \frac{d(\lambda^* F_{\dot y} y_p)}{dT} \tag{2.24}$$

where λ_T denotes $\partial\lambda/\partial T$. For index-0 and index-1 DAEs, we obtain

$$\frac{d(\lambda^* F_{\dot y} y_p)|_{t=T}}{dT} = 0,$$

while for a Hessenberg index-2 DAE system we have

$$\frac{d(\lambda^* F_{\dot y} y_p)|_{t=T}}{dT} = - \left. \frac{d(g_{y^a} (CB)^{-1} f_p^2)}{dt} \right|_{t=T}.$$

The corresponding adjoint equations are

$$(\lambda_T^* F_{\dot y})' - \lambda_T^* F_y = 0. \tag{2.25}$$

For index-0 and index-1 DAEs (as shown above, the index-2 case is different), to find the boundary condition for this equation we write λ as $\lambda(t, T)$ because it depends on both t and T. Then

$$\lambda^*(T, T) F_{\dot y}|_{t=T} = 0.$$

Taking the total derivative, we obtain

$$(\lambda_t + \lambda_T)^*(T,T) F_{\dot y}|_{t=T} + \lambda^*(T,T)\frac{dF_{\dot y}}{dt}|_{t=T} = 0.$$

Since λ_t is just $\dot\lambda$, we have the boundary condition

$$(\lambda_T^* F_{\dot y})|_{t=T} = -\left[\lambda^*(T,T)\frac{dF_{\dot y}}{dt} + \dot\lambda^* F_{\dot y}\right]|_{t=T}.$$

For the index-one DAE case, the above relation and (2.20) yield

$$(\lambda_T^* F_{\dot y})|_{t=T} = \left[g_y - \lambda^* F_y\right]|_{t=T}. \tag{2.26}$$

For the regular implicit ODE case, $F_{\dot y}$ is invertible; thus we have $\lambda(T,T) = 0$, which leads to $\lambda_T(T) = -\dot\lambda(T)$. As with the final conditions for $\lambda(T)$ in (2.20), the above selection for $\lambda_T(T)$ is not sufficient for index-two Hessenberg DAEs (see [10] for details).

2.6.3 Checkpointing scheme

During the backward integration, the evaluation of the right-hand side of the adjoint system requires, at the current time, the states y which were computed during the forward integration phase. Since IDAS implements variable-step integration formulas, it is unlikely that the states will be available at the desired time and so some form of interpolation is needed. The IDAS implementation being also variable-order, it is possible that during the forward integration phase the order may be reduced as low as first order, which means that there may be points in time where only y and $\dot y$ are available. These requirements therefore limit the choices for possible interpolation schemes. IDAS implements two interpolation methods: a cubic Hermite interpolation algorithm and a variable-degree polynomial interpolation method which attempts to mimic the BDF interpolant for the forward integration.

However, especially for large-scale problems and long integration intervals, the number and size of the vectors y and $\dot y$ that would need to be stored make this approach computationally intractable. Thus, IDAS settles for a compromise between storage space and execution time by implementing a so-called *checkpointing scheme*. At the cost of at most one additional forward integration, this approach offers the best possible estimate of memory requirements for adjoint sensitivity analysis. To begin with, based on the problem size N and the available memory, the user decides on the number N_d of data pairs (y, $\dot y$) if cubic Hermite interpolation is selected, or on the number N_d of y vectors in the case of variable-degree polynomial interpolation, that can be kept in memory for the purpose of interpolation. Then, during the first forward integration stage, after every N_d integration steps a checkpoint is formed by saving enough information (either in memory or on disk) to allow for a hot restart, that is a restart which will exactly reproduce the forward integration. In order to avoid storing Jacobian-related data at each checkpoint, a reevaluation of the iteration matrix is forced before each checkpoint. At the end of this stage, we are left with N_c checkpoints, including one at t_0. During the backward integration stage, the adjoint variables are integrated backwards from T to t_0, going from one checkpoint to the previous one. The backward integration from checkpoint $i+1$ to checkpoint i is preceded by a forward integration from i to $i+1$ during which the N_d vectors y (and, if necessary $\dot y$) are generated and stored in memory for interpolation[1]

This approach transfers the uncertainty in the number of integration steps in the forward integration phase to uncertainty in the final number of checkpoints. However, N_c is much smaller than the number of steps taken during the forward integration, and there is no major penalty for writing/reading the checkpoint data to/from a temporary file. Note that, at the end of the first forward

[1]The degree of the interpolation polynomial is always that of the current BDF order for the forward interpolation at the first point to the right of the time at which the interpolated value is sought (unless too close to the i-th checkpoint, in which case it uses the BDF order at the right-most relevant point). However, because of the FLC BDF implementation (see §2.1), the resulting interpolation polynomial is only an approximation to the underlying BDF interpolant.

The Hermite cubic interpolation option is present because it was implemented chronologically first and it is also used by other adjoint solvers (e.g. DASPKADJOINT). The variable-degree polynomial is more memory-efficient (it requires only half of the memory storage of the cubic Hermite interpolation) and is more accurate.

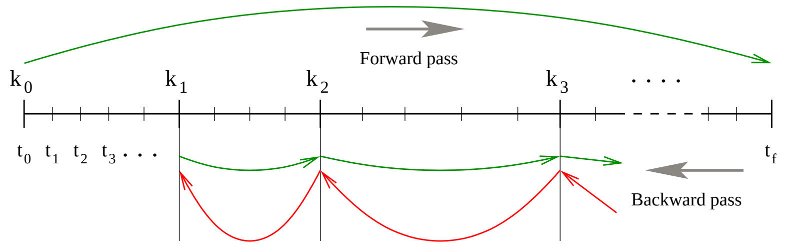


Figure 2.1: Illustration of the checkpointing algorithm for generation of the forward solution during the integration of the adjoint system.

integration stage, interpolation data are available from the last checkpoint to the end of the interval of integration. If no checkpoints are necessary (N_d is larger than the number of integration steps taken in the solution of (2.2)), the total cost of an adjoint sensitivity computation can be as low as one forward plus one backward integration. In addition, IDAS provides the capability of reusing a set of checkpoints for multiple backward integrations, thus allowing for efficient computation of gradients of several functionals (2.17).

Finally, we note that the adjoint sensitivity module in IDAS provides the necessary infrastructure to integrate backwards in time any DAE terminal value problem dependent on the solution of the IVP (2.2), including adjoint systems (2.20) or (2.25), as well as any other quadrature ODEs that may be needed in evaluating the integrals in (2.21). In particular, for DAE systems arising from semi-discretization of time-dependent PDEs, this feature allows for integration of either the discretized adjoint PDE system or the adjoint of the discretized PDE.

2.7 Second-order sensitivity analysis

In some applications (e.g., dynamically-constrained optimization) it may be desirable to compute second-order derivative information. Considering the DAE problem (2.2) and some model output functional[2] $g(y)$, the Hessian d^2g/dp^2 can be obtained in a forward sensitivity analysis setting as

$$\frac{d^2 g}{dp^2} = \left(g_y \otimes I_{N_p}\right) y_{pp} + y_p^T g_{yy} y_p \,,$$

where $\otimes$ is the Kronecker product. The second-order sensitivities are solution of the matrix DAE system:

$$\left(F_{\dot y} \otimes I_{N_p}\right) \cdot \dot y_{pp} + \left(F_y \otimes I_{N_p}\right) \cdot y_{pp} + \left(I_N \otimes \dot y_p^T\right) \cdot \left(F_{\dot y \dot y} \dot y_p + F_{y \dot y} y_p\right) + \left(I_N \otimes y_p^T\right) \cdot \left(F_{y\dot y} \dot y_p + F_{yy} y_p\right) = 0$$
$$y_{pp}(t_0) = \frac{\partial^2 y_0}{\partial p^2} \,, \quad \dot y_{pp}(t_0) = \frac{\partial^2 \dot y_0}{\partial p^2} \,,$$

where y_p denotes the first-order sensitivity matrix, the solution of N_p systems (2.12), and y_{pp} is a third-order tensor. It is easy to see that, except for situations in which the number of parameters N_p is very small, the computational cost of this so-called *forward-over-forward* approach is exorbitant as it requires the solution of $N_p + N_p^2$ additional DAE systems of the same dimension as (2.2).

A much more efficient alternative is to compute Hessian-vector products using a so-called *forward-over-adjoint* approach. This method is based on using the same "trick" as the one used in computing gradients of pointwise functionals with the adjoint method, namely applying a formal directional forward derivation to the gradient of (2.21) (or the equivalent one for a pointwise functional $g(T, y(T))$).

[2]For the sake of simplifity in presentation, we do not include explicit dependencies of g on time t or parameters p. Moreover, we only consider the case in which the dependency of the original DAE (2.2) on the parameters p is through its initial conditions only. For details on the derivation in the general case, see [27].

With that, the cost of computing a full Hessian is roughly equivalent to the cost of computing the gradient with forward sensitivity analysis. However, Hessian-vector products can be cheaply computed with one additional adjoint solve.

As an illustration[3], consider the ODE problem

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

depending on some parameters p through the initial conditions only and consider the model functional output $G(p) = \int_{t_0}^{t_f} g(t,y)\,dt$. It can be shown that the product between the Hessian of G (with respect to the parameters p) and some vector u can be computed as

$$\frac{\partial^2 G}{\partial p^2} u = \left[\left(\lambda^T \otimes I_{N_p} \right) y_{pp} u + y_p^T \mu \right]_{t=t_0} \,,$$

where λ and μ are solutions of

$$\begin{aligned} -\dot{\mu} &= f_y^T \mu + \left(\lambda^T \otimes I_n \right) f_{yy} s\,; \quad \mu(t_f) = 0 \\ -\dot{\lambda} &= f_y^T \lambda + g_y^T\,; \quad \lambda(t_f) = 0 \\ \dot{s} &= f_y s\,; \quad s(t_0) = y_{0p} u. \end{aligned} \tag{2.27}$$

In the above equation, $s = y_p u$ is a linear combination of the columns of the sensitivity matrix y_p. The *forward-over-adjoint* approach hinges crucially on the fact that s can be computed at the cost of a forward sensitivity analysis with respect to a single parameter (the last ODE problem above) which is possible due to the linearity of the forward sensitivity equations (2.12).

Therefore (and this is also valid for the DAE case), the cost of computing the Hessian-vector product is roughly that of two forward and two backward integrations of a system of DAEs of size N. For more details, including the corresponding formulas for a pointwise model functional output, see the work by Ozyurt and Barton [27] who discuss this problem for ODE initial value problems. As far as we know, there is no published equivalent work on DAE problems. However, the derivations given in [27] for ODE problems can be extended to DAEs with some careful consideration given to the derivation of proper final conditions on the adjoint systems, following the ideas presented in [10].

To allow the *foward-over-adjoint* approach described above, IDAS provides support for:

- the integration of multiple backward problems depending on the same underlying forward problem (2.2), and
- the integration of backward problems and computation of backward quadratures depending on both the states y and forward sensitivities (for this particular application, s) of the original problem (2.2).

[3]The derivation for the general DAE case is too involved for the purposes of this discussion.

Chapter 3

Code Organization

3.1 SUNDIALS organization

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

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

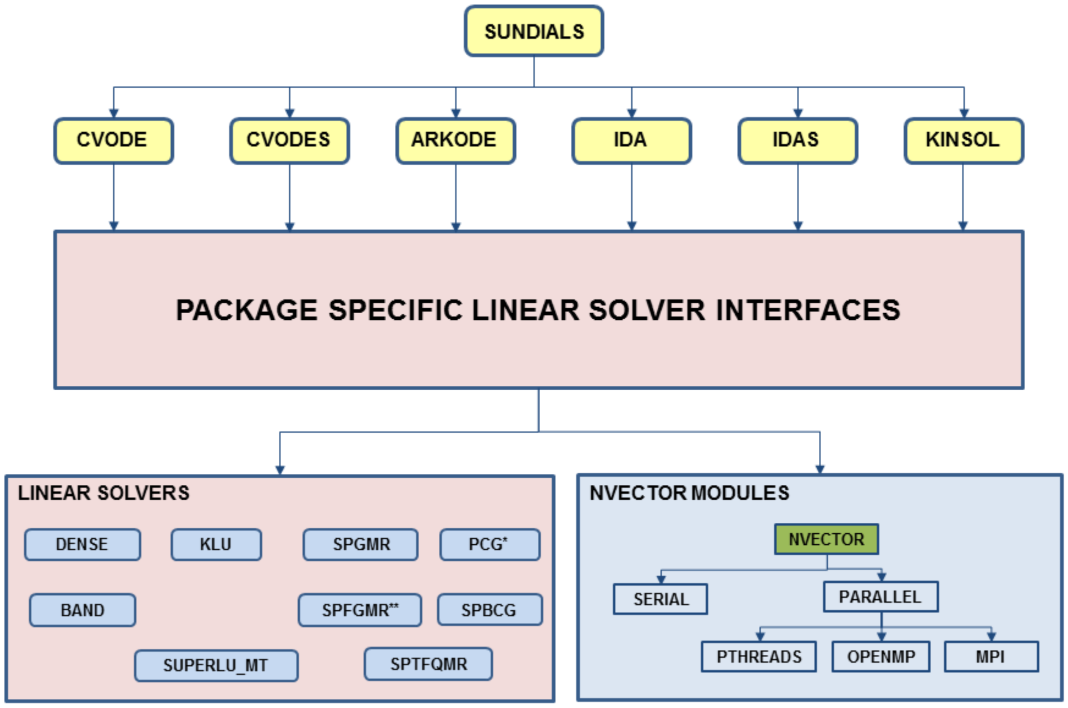
- CVODE, a solver for stiff and nonstiff ODE systems $dy/dt = f(t, y)$ based on Adams and BDF methods;
- CVODES, a solver for stiff and nonstiff ODE systems with sensitivity analysis capabilities;
- ARKODE, a solver for ODE systems $Mdy/dt = f(t, y)$ based on additive Runge-Kutta methods;
- IDA, a solver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

3.2 IDAS organization

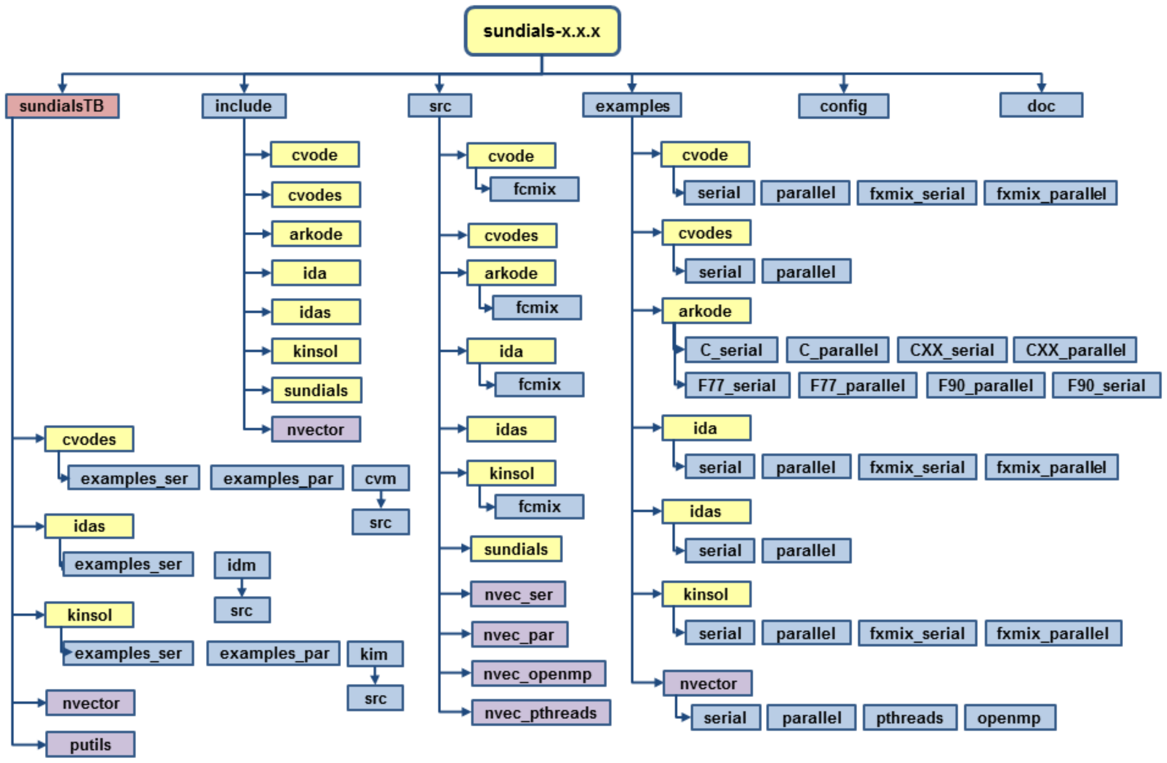
The IDAS package is written in the ANSI C language. The following summarizes the basic structure of the package, although knowledge of this structure is not necessary for its use.

The overall organization of the IDAS package is shown in Figure 3.2. The central integration module, implemented in the files `idas.h`, `idas_impl.h`, and `idas.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.

In addition, if forward sensitivity analysis is turned on, the main module will integrate the forward sensitivity equations simultaneously with the original IVP. The sensitivity variables may be included in the local error control mechanism of the main integrator. IDAS provides two different strategies for dealing with the correction stage for the sensitivity variables: `IDA_SIMULTANEOUS` `IDA_STAGGERED` (see §2.5). The IDAS package includes an algorithm for the approximation of the sensitivity equations residuals by difference quotients, but the user has the option of supplying these residual functions directly.



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



(b) Directory structure of the source tree

Figure 3.1: Organization of the SUNDIALS suite

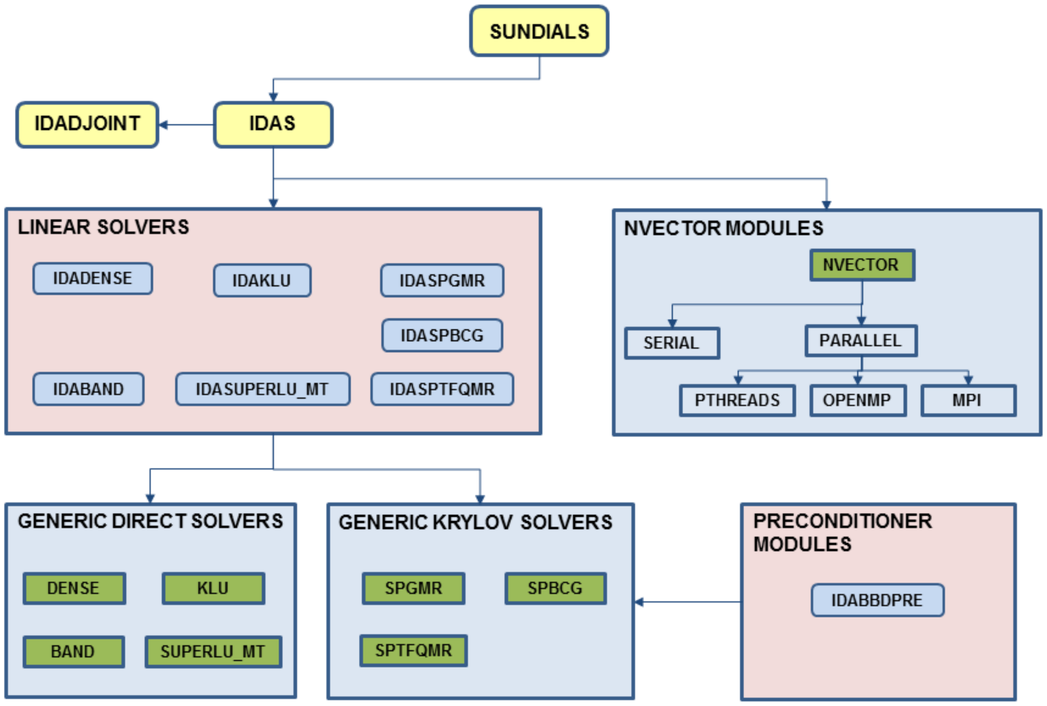


Figure 3.2: Overall structure diagram of the IDAS package. Modules specific to IDAS are distinguished by rounded boxes, while generic solver and auxiliary modules are in square boxes. Note that the direct linear solvers using Lapack implementations are not explicitly represented. Note also that the KLU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

The adjoint sensitivity module (file `idaa.c`) provides the infrastructure needed for the backward integration of any system of DAEs which depends on the solution of the original IVP, in particular the adjoint system and any quadratures required in evaluating the gradient of the objective functional. This module deals with the setup of the checkpoints, the interpolation of the forward solution during the backward integration, and the backward integration of the adjoint equations.

At present, the package includes the following seven IDAS linear algebra modules, organized into two families. The *direct* family of linear solvers provides solvers for the direct solution of linear systems with dense or banded matrices and includes:

- IDADENSE: LU factorization and backsolving with dense matrices (using either an internal implementation or Blas/Lapack);
- IDABAND: LU factorization and backsolving with banded matrices (using either an internal implementation or Blas/Lapack);
- IDAKLU: LU factorization and backsolving with compressed-sparse-column (CSC) matrices using the KLU linear solver library [14, 1] (KLU to be downloaded and compiled by user independent of IDA);
- IDASUPERLUMT: LU factorization and backsolving with compressed-sparse-column (CSC) matrices using the threaded SuperLU_MT linear solver library [25, 15, 2] (SuperLU_MT to be downloaded and compiled by user independent of IDA).

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

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

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

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

Each IDAS 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 IDAS 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 linear solver modules (IDADENSE etc.) consists of an interface built on top of a generic linear system solver (DENSE etc.). The interface deals with the use of the particular method in the IDAS context, whereas the generic solver is independent of the context. While some of the generic linear system solvers (DENSE, BAND, SPGMR, SPBCG, and SPTFQMR) were written with SUNDIALS in mind, they are intended to be usable anywhere as general-purpose solvers. This separation also allows for any generic solver to be replaced by an improved version, with no necessity to revise the IDAS package elsewhere.

IDAS 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 IDAS 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 IDAS 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 IDAS memory structure. The reentrancy of IDAS was motivated by the situation where two or more problems are solved by intermixed calls to the package from one user program.

Chapter 4

Using IDAS for IVP Solution

This chapter is concerned with the use of IDAS for the integration of DAEs. The following sections treat the header files, the layout of the user's main program, description of the IDAS user-callable functions, and description of user-supplied functions. This usage is essentially equivalent to using IDA [22].

The sample programs described in the companion document [30] 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 IDAS 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, direct band or direct sparse linear solvers, since these linear solver modules need to form the complete system Jacobian. The IDADENSE and IDABAND modules (using either the internal implementation or Lapack), as well as the IDAKLU and IDASUPERLUMT modules can only be used with NVECTOR_SERIAL, NVECTOR_OPENMP and NVECTOR_PTHREADS. It is not recommended to use a threaded vector module with SuperLU_MT unless it is the NVECTOR_OPENMP module and SuperLU_MT is also compiled with openMP. The preconditioner module IDABBDPRE can only be used with NVECTOR_PARALLEL.

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

4.1 Access to library and header files

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

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

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

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

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

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

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

4.2 Data types

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

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

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

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

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

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

4.3 Header files

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

- `idas.h`, the header file for IDAS, which defines the several types and various constants, and includes function prototypes.

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

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

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

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

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

- `idas_dense.h`, which is used with the dense direct linear solver;
- `idas_band.h`, which is used with the band direct linear solver;
- `idas_lapack.h`, which is used with Lapack implementations of dense or band direct linear solvers;
- `idas_klu.h`, which is used with the KLU sparse direct linear solver;
- `idas_superlumt.h`, which is used with the SuperLU_MT threaded sparse direct linear solver;
- `idas_spgmr.h`, which is used with the scaled, preconditioned GMRES Krylov linear solver SPGMR;
- `idas_spbcgs.h`, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver SPBCG;
- `idas_sptfqmr.h`, which is used with the scaled, preconditioned TFQMR Krylov solver SPTFQMR.

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

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

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

Other headers may be needed, according to the choice of preconditioner, etc. For example, in the `idasFoodWeb_kry_p` example (see [30]), preconditioning is done with a block-diagonal matrix. For this, even though the IDASPGMR linear solver is used, the header `sundials_dense.h` is included for access to the underlying generic dense linear solver.

4.4 A skeleton of the user's main program

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

1. **[P] Initialize MPI**

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

2. **Set problem dimensions**

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

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

[**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.

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

3. **Set vectors of initial values**

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

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

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

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

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

   Otherwise, make the calls:

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

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

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

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

   and load initial values into the structure defined by:

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

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

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

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

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

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

4. **Create IDAS object**

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

5. **Initialize IDAS solver**

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

6. **Specify integration tolerances**

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

7. **Set optional inputs**

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

8. **Attach linear solver module**

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

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

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

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

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

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

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

   `flag = IDASpgmr(...);`

   `flag = IDASpbcg(...);`

   `flag = IDASptfqmr(...);`

9. **Set linear solver optional inputs**

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

10. **Correct initial values**

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

11. **Specify rootfinding problem**

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

12. **Advance solution in time**

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

13. **Get optional outputs**

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

14. **Deallocate memory for solution vectors**

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

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

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

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

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

    and similarly for `ypret`.

15. **Free solver memory**

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

16. **[P] Finalize MPI**

    Call `MPI_Finalize()` to terminate MPI.

4.5 User-callable functions

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

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

4.5.1 IDAS 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 IDAS memory block created and allocated by the first two calls.

`IDACreate`

Call	`ida_mem = IDACreate();`
Description	The function `IDACreate` instantiates an IDAS solver object.
Arguments	`IDACreate` has no arguments.
Return value	If successful, `IDACreate` returns a pointer to the newly created IDAS memory block (of type `void *`). Otherwise it returns `NULL`.

`IDAInit`

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

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

Arguments

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

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

`IDA_SUCCESS`	The call to `IDAInit` was successful.
`IDA_MEM_NULL`	The IDAS memory block was not initialized through a previous call to `IDACreate`.
`IDA_MEM_FAIL`	A memory allocation request has failed.
`IDA_ILL_INPUT`	An input argument to `IDAInit` has an illegal value.

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

`IDAFree`

Call	`IDAFree(&ida_mem);`
Description	The function `IDAFree` frees the pointer allocated by a previous call to `IDACreate`.
Arguments	The argument is the pointer to the IDAS memory block (of type `void *`).
Return value	The function `IDAFree` has no return value.

4.5.2 IDAS tolerance specification functions

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

`IDASStolerances`

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

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

Arguments

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

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

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

`IDASVtolerances`

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

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

Arguments

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

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

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

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

`IDAWFtolerances`

Call	`flag = IDAWFtolerances(ida_mem, efun);`
Description	The function `IDAWFtolerances` specifies a user-supplied function `efun` that sets the multiplicative error weights W_i for use in the weighted RMS norm, which are normally defined by Eq. (2.7).
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block returned by `IDACreate`.
	`efun` (`IDAEwtFn`) is the C function which defines the `ewt` vector (see §4.6.3).
Return value	The return value `flag` (of type `int`) will be one of the following:
	`IDA_SUCCESS` The call to `IDAWFtolerances` was successful.
	`IDA_MEM_NULL` The IDAS memory block was not initialized through a previous call to `IDACreate`.
	`IDA_NO_MALLOC` The allocation function `IDAInit` has not been called.

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

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

(2) The absolute tolerances `abstol` (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector `y` may be so small that pure relative error control is meaningless. For example, if `y[i]` starts at some nonzero value, but in time decays to zero, then pure relative error control on `y[i]` makes no sense (and is overly costly) after `y[i]` is below some noise level. Then `abstol` (if scalar) or `abstol[i]` (if a vector) needs to be set to that noise level. If the different components have different noise levels, then `abstol` should be a vector. See the example `idasRoberts_dns` in the IDAS package, and the discussion of it in the IDAS Examples document [30]. In that problem, the three components vary between 0 and 1, and have different noise levels; hence the `abstol` vector. It is impossible to give any general advice on `abstol` values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.

(3) Finally, it is important to pick all the tolerance values conservatively, because they control the error committed on each individual time step. The final (global) errors are a sort of accumulation of those per-step errors. A good rule of thumb is to reduce the tolerances by a factor of .01 from the actual desired limits on errors. So if you want .01% accuracy (globally), a good choice is `reltol`$= 10^{-6}$. But in any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

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

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

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

(3) The user's residual routine `res` should never change a negative value in the solution vector `yy` to a non-negative value, as a "solution" to this problem. This can cause instability. If the `res` routine cannot tolerate a zero or negative value (e.g. because there is a square root or log of it), then the

offending value should be changed to zero or a tiny positive number in a temporary variable (not in the input yy vector) for the purposes of computing $F(t, y, \dot{y})$.

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

4.5.3 Linear solver specification functions

As previously explained, Newton iteration requires the solution of linear systems of the form (2.5). There are seven IDAS linear solvers currently available for this task: IDADENSE, IDABAND, IDAKLU, IDASUPERLUMT, IDASPGMR, IDASPBCG, and IDASPTFQMR.

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

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

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

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

To specify an IDAS linear solver, after the call to `IDACreate` but before any calls to `IDASolve`, the user's program must call one of the functions `IDADense`/`IDALapackDense`, `IDABand`/`IDALapackBand`, `IDAKLU`, `IDASuperLUMT`, `IDASpgmr`, `IDASpbcg`, or `IDASptfqmr`, as documented below. The first argument passed to these functions is the IDAS memory pointer returned by `IDACreate`. A call to one of these functions links the main IDAS 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 IDAS is actually built on top of a generic linear system solver, which may be of interest in itself. These generic solvers, denoted DENSE, BAND, KLU, SUPERLUMT, SPGMR, SPBCG, and SPTFQMR, are described separately in Chapter 9.

`IDADense`

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

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

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

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

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

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

`IDADLS_SUCCESS` The IDADENSE initialization was successful.

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

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

`IDALapackDense`

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

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

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

Arguments

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

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

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

`IDABand`

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

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

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

Arguments

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

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

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

Notes The IDABAND linear solver is not compatible with all implementations of the NVECTOR module. Of the NVECTOR modules provided with SUNDIALS, only NVECTOR_SERIAL, NVECTOR_OPENMP and NVECTOR_PTHREADS are 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 $-\texttt{mlower} \leq j - i \leq \texttt{mupper}$.

`IDALapackBand`

Call	`flag = IDALapackBand(ida_mem, N, mupper, mlower);`
Description	The function `IDALapackBand` selects the IDABAND linear solver and indicates the use of Lapack functions. The user's main program must include the `idas_lapack.h` header file.
Arguments	The input arguments are identical to those of `IDABand`, except that `N`, `mupper`, and `mlower` are of type `int` here.
Return value	The values of the returned `flag` (of type `int`) are identical to those of `IDABand`.
Notes	Note that `N`, `mupper`, and `mlower` are restricted to be of type `int` here, because of the corresponding type restriction in the Lapack solvers.

`IDAKLU`

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

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

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

Arguments

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

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

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

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

`IDASuperLUMT`

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

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

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

Arguments

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

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

`IDASLS_SUCCESS`	The IDASUPERLUMT initialization was successful.
`IDASLS_MEM_NULL`	The `ida_mem` pointer is `NULL`.

IDASLS_ILL_INPUT	The IDASUPERLUMT solver is not compatible with the current NVECTOR module.
IDASLS_MEM_FAIL	A memory allocation request failed.
IDASLS_PACKAGE_FAIL	A call to the SuperLU_MT library returned a failure flag.

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

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

`IDASpgmr`

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

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

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

Arguments
`ida_mem` (`void *`) pointer to the IDAS memory block.
`maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `IDA_SPILS_MAXL`= 5.

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

IDASPILS_SUCCESS	The IDASPGMR initialization was successful.
IDASPILS_MEM_NULL	The `ida_mem` pointer is `NULL`.
IDASPILS_MEM_FAIL	A memory allocation request failed.

`IDASpbcg`

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

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

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

Arguments
`ida_mem` (`void *`) pointer to the IDAS memory block.
`maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `IDA_SPILS_MAXL`= 5.

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

IDASPILS_SUCCESS	The IDASPBCG initialization was successful.
IDASPILS_MEM_NULL	The `ida_mem` pointer is `NULL`.
IDASPILS_MEM_FAIL	A memory allocation request failed.

`IDASptfqmr`

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

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

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

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

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

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

`IDASPILS_SUCCESS`	The IDASPTFQMR initialization was successful.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_MEM_FAIL`	A memory allocation request failed.

4.5.4 Initial condition calculation function

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

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

`IDACalcIC`

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

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

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`icopt`	(`int`) is one of the following two options for the initial condition calculation. `icopt`=`IDA_YA_YDP_INIT` directs `IDACalcIC` to compute the algebraic components of y and differential components of $\dot{y}$, given the differential components of y. This option requires that the `N_Vector id` was set through `IDASetId`, specifying the differential and algebraic components. `icopt`=`IDA_Y_INIT` directs `IDACalcIC` to compute all components of y, given $\dot{y}$. In this case, `id` is not required.
`tout1`	(`realtype`) is the first value of t at which a solution will be requested (from `IDASolve`). This value is needed here only to determine the direction of integration and rough scale in the independent variable t.

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

`IDA_SUCCESS`	`IDASolve` succeeded.
`IDA_MEM_NULL`	The argument `ida_mem` was `NULL`.
`IDA_NO_MALLOC`	The allocation function `IDAInit` has not been called.
`IDA_ILL_INPUT`	One of the input arguments was illegal.
`IDA_LSETUP_FAIL`	The linear solver's setup function failed in an unrecoverable manner.
`IDA_LINIT_FAIL`	The linear solver's initialization function failed.
`IDA_LSOLVE_FAIL`	The linear solver's solve function failed in an unrecoverable manner.
`IDA_BAD_EWT`	Some component of the error weight vector is zero (illegal), either for the input value of `y0` or a corrected value.
`IDA_FIRST_RES_FAIL`	The user's residual function returned a recoverable error flag on the first call, but `IDACalcIC` was unable to recover.
`IDA_RES_FAIL`	The user's residual function returned a nonrecoverable error flag.
`IDA_NO_RECOVERY`	The user's residual function, or the linear solver's setup or solve function had a recoverable error, but `IDACalcIC` was unable to recover.

IDA_CONSTR_FAIL	IDACalcIC was unable to find a solution satisfying the inequality constraints.
IDA_LINESEARCH_FAIL	The linesearch algorithm failed to find a solution with a step larger than steptol in weighted RMS norm.
IDA_CONV_FAIL	IDACalcIC failed to get convergence of the Newton iterations.

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

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

4.5.5 Rootfinding initialization function

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

IDARootInit

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

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

Arguments

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

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

IDA_SUCCESS	The call to IDARootInit was successful.
IDA_MEM_NULL	The ida_mem argument was NULL.
IDA_MEM_FAIL	A memory allocation failed.
IDA_ILL_INPUT	The function g is NULL, but nrtfn>0.

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

4.5.6 IDAS solver function

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

IDASolve

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

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

Arguments

ida_mem	(void *) pointer to the IDAS memory block.
tout	(realtype) the next time at which a computed solution is desired.
tret	(realtype) the time reached by the solver (output).
yret	(N_Vector) the computed solution vector y.
ypret	(N_Vector) the computed solution vector $\dot{y}$.

itask	(int) a flag indicating the job of the solver for the next user step. The IDA_NORMAL task is to have the solver take internal steps until it has reached or just passed the user specified tout parameter. The solver then interpolates in order to return approximate values of y(tout) and $\dot{y}$(tout). The IDA_ONE_STEP option tells the solver to just take one internal step and return the solution at the point reached by that step.

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

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

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

IDA_SUCCESS	IDASolve succeeded.
IDA_TSTOP_RETURN	IDASolve succeeded by reaching the stop point specified through the optional input function IDASetStopTime.
IDA_ROOT_RETURN	IDASolve succeeded and found one or more roots. In this case, tret is the location of the root. If nrtfn > 1, call IDAGetRootInfo to see which g_i were found to have a root. See §4.5.9.3 for more information.
IDA_MEM_NULL	The ida_mem argument was NULL.
IDA_ILL_INPUT	One of the inputs to IDASolve was illegal, or some other input to the solver was either illegal or missing. The latter category includes the following situations: (a) The tolerances have not been set. (b) A component of the error weight vector became zero during internal time-stepping. (c) The linear solver initialization function (called by the user after calling IDACreate) failed to set the linear solver-specific lsolve field in ida_mem. (d) A root of one of the root functions was found both at a point t and also very near t. In any case, the user should see the printed error message for details.
IDA_TOO_MUCH_WORK	The solver took mxstep internal steps but could not reach tout. The default value for mxstep is MXSTEP_DEFAULT = 500.
IDA_TOO_MUCH_ACC	The solver could not satisfy the accuracy demanded by the user for some internal step.
IDA_ERR_FAIL	Error test failures occurred too many times (MXNEF = 10) during one internal time step or occurred with $\|h\| = h_{min}$.
IDA_CONV_FAIL	Convergence test failures occurred too many times (MXNCF = 10) during one internal time step or occurred with $\|h\| = h_{min}$.
IDA_LINIT_FAIL	The linear solver's initialization function failed.
IDA_LSETUP_FAIL	The linear solver's setup function failed in an unrecoverable manner.
IDA_LSOLVE_FAIL	The linear solver's solve function failed in an unrecoverable manner.
IDA_CONSTR_FAIL	The inequality constraints were violated and the solver was unable to recover.
IDA_REP_RES_ERR	The user's residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.
IDA_RES_FAIL	The user's residual function returned a nonrecoverable error flag.
IDA_RTFUNC_FAIL	The rootfinding function failed.

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

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

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

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

4.5.7 Optional input functions

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

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

4.5.7.1 Main solver optional input functions

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

`IDASetErrFile`

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

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

Arguments

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

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

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

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

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

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

`IDASetErrHandlerFn`

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

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

Arguments

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

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

`IDA_SUCCESS`	The function `ehfun` and data pointer `eh_data` have been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.

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

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

[(a)] Only for IDASPGMR
[(b)] Only for IDASPBCG and IDASPTFQMR

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

`IDASetUserData`

Call	`flag = IDASetUserData(ida_mem, user_data);`
Description	The function `IDASetUserData` specifies the user data block `user_data` and attaches it to the main IDAS memory block.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `user_data` (`void *`) pointer to the user data.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
Notes	If specified, the pointer to `user_data` is passed to all user-supplied functions that have it as an argument. Otherwise, a `NULL` pointer is passed. ! If `user_data` is needed in user preconditioner functions, the call to `IDASetUserData` must be made *before* the call to specify the linear solver.

`IDASetMaxOrd`

Call	`flag = IDASetMaxOrd(ida_mem, maxord);`
Description	The function `IDASetMaxOrd` specifies the maximum order of the linear multistep method.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `maxord` (`int`) value of the maximum method order. This must be positive.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDA_ILL_INPUT` The input value `maxord` is ≤ 0, or larger than its previous value.
Notes	The default value is 5. If the input value exceeds 5, the value 5 will be used. Since `maxord` affects the memory requirements for the internal IDAS memory block, its value cannot be increased past its previous value.

`IDASetMaxNumSteps`

Call	`flag = IDASetMaxNumSteps(ida_mem, mxsteps);`
Description	The function `IDASetMaxNumSteps` specifies the maximum number of steps to be taken by the solver in its attempt to reach the next output time.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `mxsteps` (`long int`) maximum allowed number of steps.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
Notes	Passing `mxsteps` $= 0$ results in IDAS using the default value (500). Passing `mxsteps` < 0 disables the test (*not recommended).*

`IDASetInitStep`

Call	`flag = IDASetInitStep(ida_mem, hin);`
Description	The function `IDASetInitStep` specifies the initial step size.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `hin` (`realtype`) value of the initial step size to be attempted. Pass 0.0 to have IDAS use the default value.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
Notes	By default, IDAS estimates the initial step as the solution of $\|h\dot{y}\|_{\mathrm{WRMS}} = 1/2$, with an added restriction that $|h| \leq .001|$`tout - t0`$|$.

`IDASetMaxStep`

Call	`flag = IDASetMaxStep(ida_mem, hmax);`
Description	The function `IDASetMaxStep` specifies the maximum absolute value of the step size.
Arguments	`ida_mem` (`void *`) pointer to the IDAS 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 IDAS memory block. `tstop` (`realtype`) value of the independent variable past which the solution should not proceed.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDA_ILL_INPUT` The value of `tstop` is not beyond the current t value, t_n.
Notes	The default, if this routine is not called, is that no stop time is imposed.

`IDASetMaxErrTestFails`

Call	`flag = IDASetMaxErrTestFails(ida_mem, maxnef);`
Description	The function `IDASetMaxErrTestFails` specifies the maximum number of error test failures in attempting one step.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `maxnef` (`int`) maximum number of error test failures allowed on one step (> 0).
Return value	The return value `flag` (of type `int`) is one of

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

Notes: The default value is 7.

`IDASetMaxNonlinIters`

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

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

Arguments:

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

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

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

Notes: The default value is 3.

`IDASetMaxConvFails`

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

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

Arguments:

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`maxncf`	(`int`) maximum number of allowable nonlinear solver convergence failures on one step (> 0).

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

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

Notes: The default value is 10.

`IDASetNonlinConvCoef`

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

Description: The function `IDASetNonlinConvCoef` specifies the safety factor in the nonlinear convergence test; see Chapter 2, Eq. (2.8).

Arguments:

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`nlscoef`	(`realtype`) coefficient in nonlinear convergence test (> 0.0).

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

`IDA_SUCCESS`	The optional value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT`	The value of `nlscoef` is $<= 0.0$.

Notes: The default value is 0.33.

`IDASetSuppressAlg`

Call	`flag = IDASetSuppressAlg(ida_mem, suppressalg);`
Description	The function `IDASetSuppressAlg` indicates whether or not to suppress algebraic variables in the local error test.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `suppressalg` (`booleantype`) indicates whether to suppress (`TRUE`) or not (`FALSE`) the algebraic variables in the local error test.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
Notes	The default value is `FALSE`. If `suppressalg`=`TRUE` is selected, then the `id` vector must be set (through `IDASetId`) to specify the algebraic components. In general, the use of this option (with `suppressalg = TRUE`) is *discouraged* when solving DAE systems of index 1, whereas it is generally *encouraged* for systems of index 2 or more. See pp. 146-147 of Ref. [3] for more on this issue.

`IDASetId`

Call	`flag = IDASetId(ida_mem, id);`
Description	The function `IDASetId` specifies algebraic/differential components in the y vector.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `id` (`N_Vector`) state vector. A value of 1.0 indicates a differential variable, while 0.0 indicates an algebraic variable.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
Notes	The vector `id` is required if the algebraic variables are to be suppressed from the local error test (see `IDASetSuppressAlg`) or if `IDACalcIC` is to be called with `icopt` = `IDA_YA_YDP_INIT` (see §4.5.4).

`IDASetConstraints`

Call	`flag = IDASetConstraints(ida_mem, constraints);`
Description	The function `IDASetConstraints` specifies a vector defining inequality constraints for each component of the solution vector y.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `constraints` (`N_Vector`) vector of constraint flags. If `constraints[i]` is 0.0 then no constraint is imposed on y_i. 1.0 then y_i will be constrained to be $y_i \geq 0.0$. -1.0 then y_i will be constrained to be $y_i \leq 0.0$. 2.0 then y_i will be constrained to be $y_i > 0.0$. -2.0 then y_i will be constrained to be $y_i < 0.0$.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDA_ILL_INPUT` The constraints vector contains illegal values.

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

4.5.7.2 Dense/band direct linear solvers optional input functions

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

`IDADlsSetDenseJacFn`

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

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

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

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

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

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

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

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

`IDADlsSetBandJacFn`

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

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

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

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

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

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

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

4.5.7.3 Sparse direct linear solvers optional input functions

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

`IDASlsSetSparseJacFn`

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

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

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

`sjac` (`IDASlsSparseJacFn`) user-defined sparse Jacobian approximation function.

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

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

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

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

`IDAKLUReInit`

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

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

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

`n` (`int`) number of state variables in the system.

`nnz` (`int`) number of nonzeroes in the Jacobian matrix.

`reinit_type` (`int`) type of reinitialization:

1 The Jacobian matrix will be destroyed and a new one will be allocated based on the `nnz` value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.

2 Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of `nnz` given in the prior call to IDAKLU.

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

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

Notes The default value for `reinit_type` is 2.

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

`IDAKLUSetOrdering`

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

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

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

`ordering_choice` (`int`) flag denoting algorithm choice:

0	`AMD`
1	`COLAMD`
2	natural ordering

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

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

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

`IDASuperLUMTSetOrdering`

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

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

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

`ordering_choice` (`int`) flag denoting algorithm choice:

0	natural ordering
1	minimal degree ordering on $J^T J$
2	minimal degree ordering on $J^T + J$
3	`COLAMD`

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

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

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

4.5.7.4 Iterative linear solvers optional input functions

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

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

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

`IDASpilsSetPreconditioner`

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

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

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

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

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

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

`IDASpilsSetJacTimesVecFn`

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

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

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

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

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

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

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

`IDASpilsSetGSType`

Call	`flag = IDASpilsSetGSType(ida_mem, gstype);`
Description	The function `IDASpilsSetGSType` specifies the Gram-Schmidt orthogonalization to be used. This must be one of the enumeration constants `MODIFIED_GS` or `CLASSICAL_GS`. These correspond to using modified Gram-Schmidt and classical Gram-Schmidt, respectively.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `gstype` (`int`) type of Gram-Schmidt orthogonalization.
Return value	The return value `flag` (of type `int`) is one of `IDASPILS_SUCCESS` The optional value has been successfully set. `IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDASPILS_LMEM_NULL` The IDASPILS linear solver has not been initialized. `IDASPILS_ILL_INPUT` The value of `gstype` is not valid.
Notes	The default value is `MODIFIED_GS`. This option is available only for the IDASPGMR linear solver.

`IDASpilsSetMaxRestarts`

Call	`flag = IDASpilsSetMaxRestarts(ida_mem, maxrs);`
Description	The function `IDASpilsSetMaxRestarts` specifies the maximum number of restarts to be used in the GMRES algorithm.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `maxrs` (`int`) maximum number of restarts.
Return value	The return value `flag` (of type `int`) is one of `IDASPILS_SUCCESS` The optional value has been successfully set. `IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDASPILS_LMEM_NULL` The IDASPILS linear solver has not been initialized. `IDASPILS_ILL_INPUT` The `maxrs` argument is negative.
Notes	The default value is 5. Pass `maxrs` $= 0$ to specify no restarts. This option is available only for the IDASPGMR linear solver.

`IDASpilsSetEpsLin`

Call	`flag = IDASpilsSetEpsLin(ida_mem, eplifac);`
Description	The function `IDASpilsSetEpsLin` specifies the factor by which the Krylov linear solver's convergence test constant is reduced from the Newton iteration test constant. (See §2.1).
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `eplifac` (`realtype`) linear convergence safety factor ($>=$ 0.0).
Return value	The return value `flag` (of type `int`) is one of `IDASPILS_SUCCESS` The optional value has been successfully set. `IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDASPILS_LMEM_NULL` The IDASPILS linear solver has not been initialized. `IDASPILS_ILL_INPUT` The value of `eplifac` is negative.
Notes	The default value is 0.05. Passing a value `eplifac`$=$ 0.0 also indicates using the default value.

`IDASpilsSetIncrementFactor`

Call	`flag = IDASpilsSetIncrementFactor(ida_mem, dqincfac);`
Description	The function `IDASpilsSetIncrementFactor` specifies a factor in the increments to y used in the difference quotient approximations to the Jacobian-vector products. (See §2.1). The increment used to approximate Jv will be $\sigma =$ `dqincfac`$/\|v\|$.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `dqincfac` (`realtype`) difference quotient increment factor.
Return value	The return value `flag` (of type `int`) is one of

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

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

`IDASpilsSetMaxl`

Call	`flag = IDASpilsSetMaxl(ida_mem, maxl);`
Description	The function `IDASpilsSetMaxl` resets the maximum Krylov subspace dimension for the Bi-CGStab or TFQMR methods.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `maxl` (`int`) maximum dimension of the Krylov subspace.
Return value	The return value `flag` (of type `int`) is one of

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

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

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

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

4.5.7.5 Initial condition calculation optional input functions

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

`IDASetNonlinConvCoefIC`

Call	`flag = IDASetNonlinConvCoefIC(ida_mem, epiccon);`
Description	The function `IDASetNonlinConvCoefIC` specifies the positive constant in the Newton iteration convergence test within the initial condition calculation.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `epiccon` (`realtype`) coefficient in the Newton convergence test (> 0).
Return value	The return value `flag` (of type `int`) is one of

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

Notes	The default value is $0.01 \cdot 0.33$. This test uses a weighted RMS norm (with weights defined by the tolerances). For new initial value vectors y and $\dot{y}$ to be accepted, the norm of $J^{-1}F(t_0, y, \dot{y})$ must be $\leq$ `epiccon`, where J is the system Jacobian.

`IDASetMaxNumStepsIC`

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

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

Arguments

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

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

`IDA_SUCCESS`	The optional value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT`	`maxnh` is non-positive.

Notes The default value is 5.

`IDASetMaxNumJacsIC`

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

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

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`maxnj`	(`int`) maximum allowed number of Jacobian or preconditioner evaluations.

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

`IDA_SUCCESS`	The optional value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT`	`maxnj` is non-positive.

Notes The default value is 4.

`IDASetMaxNumItersIC`

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

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

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`maxnit`	(`int`) maximum number of Newton iterations.

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

`IDA_SUCCESS`	The optional value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT`	`maxnit` is non-positive.

Notes The default value is 10.

`IDASetLineSearchOffIC`

Call	`flag = IDASetLineSearchOffIC(ida_mem, lsoff);`
Description	The function `IDASetLineSearchOffIC` specifies whether to turn on or off the linesearch algorithm.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `lsoff` (`booleantype`) a flag to turn off (`TRUE`) or keep (`FALSE`) the linesearch algorithm.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
Notes	The default value is `FALSE`.

`IDASetStepToleranceIC`

Call	`flag = IDASetStepToleranceIC(ida_mem, steptol);`
Description	The function `IDASetStepToleranceIC` specifies a positive lower bound on the Newton step.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `steptol` (`int`) Minimum allowed WRMS-norm of the Newton step (> 0.0).
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDA_ILL_INPUT` The `steptol` tolerance is $<= 0.0$.
Notes	The default value is (unit roundoff)$^{2/3}$.

4.5.7.6 Rootfinding optional input functions

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

`IDASetRootDirection`

Call	`flag = IDASetRootDirection(ida_mem, rootdir);`
Description	The function `IDASetRootDirection` specifies the direction of zero-crossings to be located and returned to the user.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `rootdir` (`int *`) state array of length `nrtfn`, the number of root functions g_i, as specified in the call to the function `IDARootInit`. A value of 0 for `rootdir[i]` indicates that crossing in either direction should be reported for g_i. A value of $+1$ or -1 indicates that the solver should report only zero-crossings where g_i is increasing or decreasing, respectively.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDA_ILL_INPUT` rootfinding has not been activated through a call to `IDARootInit`.
Notes	The default behavior is to locate both zero-crossing directions.

`IDASetNoInactiveRootWarn`

Call	`flag = IDASetNoInactiveRootWarn(ida_mem);`
Description	The function `IDASetNoInactiveRootWarn` disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
Notes	IDAS will not report the initial conditions as a possible zero-crossing (assuming that one or more components g_i are zero at the initial time). However, if it appears that some g_i is identically zero at the initial time (i.e., g_i is zero at the initial time and after the first step), IDAS will issue a warning which can be disabled with this optional input function.

4.5.8 Interpolated output function

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

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

`IDAGetDky`

Call	`flag = IDAGetDky(ida_mem, t, k, dky);`
Description	The function `IDAGetDky` computes the interpolated values of the k^{th} derivative of y for any value of t in the last internal step taken by IDAS. The value of k must be non-negative and smaller than the last internal order used. A value of 0 for k means that the y is interpolated. The value of t must satisfy $t_n - h_u \le t \le t_n$, 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 IDAS memory block. `t` (`realtype`) time at which to interpolate. `k` (`int`) integer specifying the order of the derivative of y wanted. `dky` (`N_Vector`) vector containing the interpolated k^{th} derivative of $y(t)$.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` `IDAGetDky` succeeded. `IDA_MEM_NULL` The `ida_mem` argument was `NULL`. `IDA_BAD_T` `t` is not in the interval $[t_n - h_u, t_n]$. `IDA_BAD_K` `k` is not one of $\{0, 1, \ldots, klast\}$. `IDA_BAD_DKY` `dky` is `NULL`.
Notes	It is only legal to call the function `IDAGetDky` after a successful return from `IDASolve`. Functions `IDAGetCurrentTime`, `IDAGetLastStep` and `IDAGetLastOrder` (see §4.5.9.1) can be used to access t_n, h_u and $klast$.

4.5.9 Optional output functions

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

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

4.5.9.1 Main solver optional output functions

IDAS 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 IDAS 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 IDAS 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 IDAS real and integer workspace sizes.

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

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

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

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

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

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

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

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

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

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

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

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

Note that additional memory is allocated if quadratures and/or forward sensitivity integration is enabled. See §4.7.1 and §5.2.1 for more details.

`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 IDAS memory block.
`nsteps`	(`long int`) number of steps taken by IDAS.

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

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

`IDAGetNumResEvals`

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

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

Arguments

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

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

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

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

`IDAGetNumLinSolvSetups`

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

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

Arguments

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

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

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

`IDAGetNumErrTestFails`

Call	`flag = IDAGetNumErrTestFails(ida_mem, &netfails);`
Description	The function `IDAGetNumErrTestFails` returns the cumulative number of local error test failures that have occurred (total so far).
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `netfails` (`long int`) number of error test failures.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetLastOrder`

Call	`flag = IDAGetLastOrder(ida_mem, &klast);`
Description	The function `IDAGetLastOrder` returns the integration method order used during the last internal step.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `klast` (`int`) method order used on the last internal step.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetCurrentOrder`

Call	`flag = IDAGetCurrentOrder(ida_mem, &kcur);`
Description	The function `IDAGetCurrentOrder` returns the integration method order to be used on the next internal step.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `kcur` (`int`) method order to be used on the next internal step.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetLastStep`

Call	`flag = IDAGetLastStep(ida_mem, &hlast);`
Description	The function `IDAGetLastStep` returns the integration step size taken on the last internal step.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `hlast` (`realtype`) step size taken on the last internal step by IDA, or last artificial step size used in `IDACalcIC`, whichever was called last.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetCurrentStep`

Call	`flag = IDAGetCurrentStep(ida_mem, &hcur);`
Description	The function `IDAGetCurrentStep` returns the integration step size to be attempted on the next internal step.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `hcur` (`realtype`) step size to be attempted on the next internal step.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetActualInitStep`

Call	`flag = IDAGetActualInitStep(ida_mem, &hinused);`
Description	The function `IDAGetActualInitStep` returns the value of the integration step size used on the first step.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `hinused` (`realtype`) actual value of initial step size.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
Notes	Even if the value of the initial integration step size was specified by the user through a call to `IDASetInitStep`, this value might have been changed by IDAS to ensure that the step size is within the prescribed bounds ($h_{\min} \leq h_0 \leq h_{\max}$), 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 IDAS memory block. `tcur` (`realtype`) current internal time reached.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetTolScaleFactor`

Call	`flag = IDAGetTolScaleFactor(ida_mem, &tolsfac);`
Description	The function `IDAGetTolScaleFactor` returns a suggested factor by which the user's tolerances should be scaled when too much accuracy has been requested for some internal step.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `tolsfac` (`realtype`) suggested scaling factor for user tolerances.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetErrWeights`

Call	`flag = IDAGetErrWeights(ida_mem, eweight);`
Description	The function `IDAGetErrWeights` returns the solution error weights at the current time. These are the W_i given by Eq. (2.7) (or by the user's `IDAEwtFn`).
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `eweight` (`N_Vector`) solution error weights at the current time.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
Notes	! The user must allocate space for `eweight`.

`IDAGetEstLocalErrors`

Call	`flag = IDAGetEstLocalErrors(ida_mem, ele);`
Description	The function `IDAGetEstLocalErrors` returns the estimated local errors.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `ele` (`N_Vector`) estimated local errors at the current time.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
Notes	! The user must allocate space for `ele`. The values returned in `ele` are only valid if `IDASolve` returned a non-negative value. The `ele` vector, togther with the `eweight` vector from `IDAGetErrWeights`, can be used to determine how the various components of the system contributed to the estimated local error test. Specifically, that error test uses the RMS norm of a vector whose components are the products of the components of these two vectors. Thus, for example, if there were recent error test failures, the components causing the failures are those with largest values for the products, denoted loosely as `eweight[i]*ele[i]`.

`IDAGetIntegratorStats`

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

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

Arguments

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

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

`IDAGetNumNonlinSolvIters`

Call	`flag = IDAGetNumNonlinSolvIters(ida_mem, &nniters);`
Description	The function `IDAGetNumNonlinSolvIters` returns the cumulative number of nonlinear (functional or Newton) iterations performed.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block.
	`nniters` (`long int`) number of nonlinear iterations performed.
Return value	The return value `flag` (of type `int`) is one of
	`IDA_SUCCESS` The optional output value has been successfully set.
	`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetNumNonlinSolvConvFails`

Call	`flag = IDAGetNumNonlinSolvConvFails(ida_mem, &nncfails);`
Description	The function `IDAGetNumNonlinSolvConvFails` returns the cumulative number of nonlinear convergence failures that have occurred.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block.
	`nncfails` (`long int`) number of nonlinear convergence failures.
Return value	The return value `flag` (of type `int`) is one of
	`IDA_SUCCESS` The optional output value has been successfully set.
	`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetNonlinSolvStats`

Call	`flag = IDAGetNonlinSolvStats(ida_mem, &nniters, &nncfails);`
Description	The function `IDAGetNonlinSolvStats` returns the IDAS nonlinear solver statistics as a group.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block.
	`nniters` (`long int`) cumulative number of nonlinear iterations performed.
	`nncfails` (`long int`) cumulative number of nonlinear convergence failures.
Return value	The return value `flag` (of type `int`) is one of
	`IDA_SUCCESS` The optional output value has been successfully set.
	`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetReturnFlagName`

Call	`name = IDAGetReturnFlagName(flag);`
Description	The function `IDAGetReturnFlagName` returns the name of the IDAS constant corresponding to `flag`.
Arguments	The only argument, of type `int`, is a return flag from an IDAS function.
Return value	The return value is a string containing the name of the corresponding constant.

4.5.9.2 Initial condition calculation optional output functions

`IDAGetNumBcktrackOps`

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

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

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

`nbacktr` (`long int`) the cumulative number of backtrack operations.

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

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

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

`IDAGetConsistentIC`

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

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

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

`yy0_mod` (`N_Vector`) consistent solution vector.

`yp0_mod` (`N_Vector`) consistent derivative vector.

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

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

`IDA_ILL_INPUT` The function was not called before the first call to `IDASolve`.

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

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

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

4.5.9.3 Rootfinding optional output functions

There are two optional output functions associated with rootfinding.

`IDAGetRootInfo`

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

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

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

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

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

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

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

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

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

`IDAGetNumGEvals`

Call	`flag = IDAGetNumGEvals(ida_mem, &ngevals);`
Description	The function `IDAGetNumGEvals` returns the cumulative number of calls to the user root function g.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `ngevals` (`long int`) number of calls to the user's function `g` so far.
Return value	The return value `flag` (of type `int`) is one of `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

4.5.9.4 Dense/band direct linear solvers optional output functions

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

`IDADlsGetWorkSpace`

Call	`flag = IDADlsGetWorkSpace(ida_mem, &lenrwLS, &leniwLS);`
Description	The function `IDADlsGetWorkSpace` returns the sizes of the real and integer workspaces used by an IDADLS linear solver (IDADENSE or IDABAND).
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `lenrwLS` (`long int`) the number of real values in the IDADLS workspace. `leniwLS` (`long int`) the number of integer values in the IDADLS workspace.
Return value	The return value `flag` (of type `int`) is one of `IDADLS_SUCCESS` The optional output value has been successfully set. `IDADLS_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDADLS_LMEM_NULL` The IDADLS linear solver has not been initialized.
Notes	For the IDADENSE linear solver, in terms of the problem size N, the actual size of the real workspace is $2N^2$ `realtype` words, while the actual size of the integer workspace is N integer words. For the IDABAND linear solver, in terms of N and Jacobian half-bandwidths, the actual size of the real workspace is $N\,(2$ `mupper`$+3$ `mlower` $+2)$ `realtype` words, while the actual size of the integer workspace is N integer words.

`IDADlsGetNumJacEvals`

Call	`flag = IDADlsGetNumJacEvals(ida_mem, &njevals);`
Description	The function `IDADlsGetNumJacEvals` returns the cumulative number of calls to the IDADLS (dense or banded) Jacobian approximation function.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `njevals` (`long int`) the cumulative number of calls to the Jacobian function (total so far).
Return value	The return value `flag` (of type `int`) is one of `IDADLS_SUCCESS` The optional output value has been successfully set. `IDADLS_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDADLS_LMEM_NULL` The IDADENSE linear solver has not been initialized.

`IDADlsGetNumResEvals`

Call	`flag = IDADlsGetNumResEvals(ida_mem, &nrevalsLS);`
Description	The function `IDADlsGetNumResEvals` returns the cumulative number of calls to the user residual function due to the finite difference (dense or band) Jacobian approximation.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `nrevalsLS` (`long int`) the cumulative number of calls to the user residual function.
Return value	The return value `flag` (of type `int`) is one of `IDADLS_SUCCESS` The optional output value has been successfully set. `IDADLS_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDADLS_LMEM_NULL` The IDADENSE linear solver has not been initialized.
Notes	The value `nrevalsLS` is incremented only if the default internal difference quotient function is used.

`IDADlsGetLastFlag`

Call	`flag = IDADlsGetLastFlag(ida_mem, &lsflag);`
Description	The function `IDADlsGetLastFlag` returns the last return value from an IDADLS routine.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `lsflag` (`long int`) the value of the last return flag from an IDADLS function.
Return value	The return value `flag` (of type `int`) is one of `IDADLS_SUCCESS` The optional output value has been successfully set. `IDADLS_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDADLS_LMEM_NULL` The IDADENSE linear solver has not been initialized.
Notes	If the IDADENSE setup function failed (i.e., `IDASolve` returned `IDA_LSETUP_FAIL`), the value `lsflag` is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or band) Jacobian matrix. For all other failures, the value of `lsflag` is negative.

`IDADlsGetReturnFlagName`

Call	`name = IDADlsGetReturnFlagName(lsflag);`
Description	The function `IDADlsGetReturnFlagName` returns the name of the IDADLS constant corresponding to `lsflag`.
Arguments	The only argument, of type `long int`, is a return flag from an IDADLS function.
Return value	The return value is a string containing the name of the corresponding constant. If $1 \leq$ `lsflag` $\leq N$ (LU factorization failed), this function returns "NONE".

4.5.9.5 Sparse direct linear solvers optional output functions

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

`IDASlsGetNumJacEvals`

Call	`flag = IDASlsGetNumJacEvals(ida_mem, &njevals);`
Description	The function `IDASlsGetNumJacEvals` returns the cumulative number of calls to the IDASLS sparse Jacobian approximation function.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block.

	`njevals` (`long int`) the cumulative number of calls to the Jacobian function (total so far).
Return value	The return value `flag` (of type `int`) is one of
	`IDASLS_SUCCESS` The optional output value has been successfully set.
	`IDASLS_MEM_NULL` The `ida_mem` pointer is `NULL`.
	`IDASLS_LMEM_NULL` The IDASLS linear solver has not been initialized.

`IDASlsGetLastFlag`

Call	`flag = IDASlsGetLastFlag(ida_mem, &lsflag);`
Description	The function `IDASlsGetLastFlag` returns the last return value from an IDASLS routine.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block.
	`lsflag` (`long int`) the value of the last return flag from an IDASLS function.
Return value	The return value `flag` (of type `int`) is one of
	`IDASLS_SUCCESS` The optional output value has been successfully set.
	`IDASLS_MEM_NULL` The `ida_mem` pointer is `NULL`.
	`IDASLS_LMEM_NULL` The IDASLS linear solver has not been initialized.
Notes	

`IDASlsGetReturnFlagName`

Call	`name = IDASlsGetReturnFlagName(lsflag);`
Description	The function `IDASlsGetReturnFlagName` returns the name of the IDASLS constant corresponding to `lsflag`.
Arguments	The only argument, of type `long int`, is a return flag from an IDASLS function.
Return value	The return value is a string containing the name of the corresponding constant.

4.5.9.6 Iterative linear solvers optional output functions

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

`IDASpilsGetWorkSpace`

Call	`flag = IDASpilsGetWorkSpace(ida_mem, &lenrwLS, &leniwLS);`
Description	The function `IDASpilsGetWorkSpace` returns the global sizes of the IDASPILS real and integer workspaces.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block.
	`lenrwLS` (`long int`) global number of real values in the IDASPILS workspace.
	`leniwLS` (`long int`) global number of integer values in the IDASPILS workspace.
Return value	The return value `flag` (of type `int`) is one of
	`IDASPILS_SUCCESS` The optional output value has been successfully set.
	`IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`.
	`IDASPILS_LMEM_NULL` The IDASPILS linear solver has not been initialized.

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

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

`IDASpilsGetNumLinIters`

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

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

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

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

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

`IDASpilsGetNumConvFails`

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

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

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

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

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

`IDASpilsGetNumPrecEvals`

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

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

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

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

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

`IDASpilsGetNumPrecSolves`

Call	`flag = IDASpilsGetNumPrecSolves(ida_mem, &npsolves);`
Description	The function `IDASpilsGetNumPrecSolves` returns the cumulative number of calls made to the preconditioner solve function, `psolve`.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `npsolves` (`long int`) the cumulative number of calls to `psolve`.
Return value	The return value `flag` (of type `int`) is one of `IDASPILS_SUCCESS` The optional output value has been successfully set. `IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDASPILS_LMEM_NULL` The IDASPILS linear solver has not been initialized.

`IDASpilsGetNumJtimesEvals`

Call	`flag = IDASpilsGetNumJtimesEvals(ida_mem, &njvevals);`
Description	The function `IDASpilsGetNumJtimesEvals` returns the cumulative number of calls made to the Jacobian-vector function, `jtimes`.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `njvevals` (`long int`) the cumulative number of calls to `jtimes`.
Return value	The return value `flag` (of type `int`) is one of `IDASPILS_SUCCESS` The optional output value has been successfully set. `IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDASPILS_LMEM_NULL` The IDASPILS linear solver has not been initialized.

`IDASpilsGetNumResEvals`

Call	`flag = IDASpilsGetNumResEvals(ida_mem, &nrevalsLS);`
Description	The function `IDASpilsGetNumResEvals` returns the cumulative number of calls to the user residual function for finite difference Jacobian-vector product approximation.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `nrevalsLS` (`long int`) the cumulative number of calls to the user residual function.
Return value	The return value `flag` (of type `int`) is one of `IDASPILS_SUCCESS` The optional output value has been successfully set. `IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDASPILS_LMEM_NULL` The IDASPILS linear solver has not been initialized.
Notes	The value `nrevalsLS` is incremented only if the default `IDASpilsDQJtimes` difference quotient function is used.

`IDASpilsGetLastFlag`

Call	`flag = IDASpilsGetLastFlag(ida_mem, &lsflag);`
Description	The function `IDASpilsGetLastFlag` returns the last return value from an IDASPILS routine.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `lsflag` (`long int`) the value of the last return flag from an IDASPILS function.
Return value	The return value `flag` (of type `int`) is one of `IDASPILS_SUCCESS` The optional output value has been successfully set. `IDASPILS_MEM_NULL` The `ida_mem` pointer is `NULL`.

	`IDASPILS_LMEM_NULL` The IDASPILS linear solver has not been initialized.
Notes	If the IDASPILS setup function failed (`IDASolve` returned `IDA_LSETUP_FAIL`), `lsflag` will be `SPGMR_PSET_FAIL_UNREC`, `SPBCG_PSET_FAIL_UNREC`, or `SPTFQMR_PSET_FAIL_UNREC`.
	If the IDASPGMR solve function failed (`IDASolve` returned `IDA_LSOLVE_FAIL`), `lsflag` contains the error return flag from `SpgmrSolve` and will be one of: `SPGMR_MEM_NULL`, indicating that the SPGMR memory is `NULL`; `SPGMR_ATIMES_FAIL_UNREC`, indicating an unrecoverable failure in the $J * v$ function; `SPGMR_PSOLVE_FAIL_UNREC`, indicating that the preconditioner solve function `psolve` failed unrecoverably; `SPGMR_GS_FAIL`, indicating a failure in the Gram-Schmidt procedure; or `SPGMR_QRSOL_FAIL`, indicating that the matrix R was found to be singular during the QR solve phase.
	If the IDASPBCG solve function failed (`IDASolve` returned `IDA_LSOLVE_FAIL`), `lsflag` contains the error return flag from `SpbcgSolve` and will be one of: `SPBCG_MEM_NULL`, indicating that the SPBCG memory is `NULL`; `SPBCG_ATIMES_FAIL_UNREC`, indicating an unrecoverable failure in the $J*v$ function; or `SPBCG_PSOLVE_FAIL_UNREC`, indicating that the preconditioner solve function `psolve` failed unrecoverably.
	If the IDASPTFQMR solve function failed (`IDASolve` returned `IDA_LSOLVE_FAIL`), `lsflag` contains the error flag from `SptfqmrSolve` and will be one of: `SPTFQMR_MEM_NULL`, indicating that the SPTFQMR memory is `NULL`; `SPTFQMR_ATIMES_FAIL_UNREC`, indicating an unrecoverable failure in the $J*v$ function; or `SPTFQMR_PSOLVE_FAIL_UNREC`, indicating that the preconditioner solve function `psolve` failed unrecoverably.

`IDASpilsGetReturnFlagName`

Call	`name = IDASpilsGetReturnFlagName(lsflag);`
Description	The function `IDASpilsGetReturnFlagName` returns the name of the IDASPILS constant corresponding to `lsflag`.
Arguments	The only argument, of type `long int`, is a return flag from an IDASPILS function.
Return value	The return value is a string containing the name of the corresponding constant.

4.5.10 IDAS reinitialization function

The function `IDAReInit` reinitializes the main IDAS solver for the solution of a problem, where a prior call to `IDAInit` has been made. The new problem must have the same size as the previous one. `IDAReInit` performs the same input checking and initializations that `IDAInit` does, but does no memory allocation, assuming that the existing internal memory is sufficient for the new problem. A call to `IDAReInit` deletes the solution history that was stored internally during the previous integration.

The use of `IDAReInit` requires that the maximum method order, `maxord`, is no larger for the new problem than for the problem specified in the last call to `IDAInit`. In addition, the same NVECTOR module set for the previous problem will be reused for the new problem.

If there are changes to the linear solver specifications, make the appropriate `IDA***` calls, as described in §4.5.3. If there are changes to any optional inputs, make the appropriate `IDASet***` calls, as described in §4.5.7.

`IDAReInit`

Call	`flag = IDAReInit(ida_mem, t0, y0, yp0);`
Description	The function `IDAReInit` provides required problem specifications and reinitializes IDAS.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block.
	`t0` (`realtype`) is the initial value of t.
	`y0` (`N_Vector`) is the initial value of y.
	`yp0` (`N_Vector`) is the initial value of $\dot{y}$.

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

`IDA_SUCCESS`	The call to `IDAReInit` was successful.
`IDA_MEM_NULL`	The IDAS memory block was not initialized through a previous call to `IDACreate`.
`IDA_NO_MALLOC`	Memory space for the IDAS memory block was not allocated through a previous call to `IDAInit`.
`IDA_ILL_INPUT`	An input argument to `IDAReInit` has an illegal value.

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

4.6 User-supplied functions

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

4.6.1 Residual function

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

`IDAResFn`

Definition
```
typedef int (*IDAResFn)(realtype tt, N_Vector yy, N_Vector yp,
                        N_Vector rr, void *user_data);
```

Purpose This function computes the problem residual for given values of the independent variable t, state vector y, and derivative $\dot{y}$.

Arguments

`tt`	is the current value of the independent variable.
`yy`	is the current value of the dependent variable vector, $y(t)$.
`yp`	is the current value of $\dot{y}(t)$.
`rr`	is the output residual vector $F(t, y, \dot{y})$.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.

Return value An `IDAResFn` function type should return a value of 0 if successful, a positive value if a recoverable error occurred (e.g. `yy` has an illegal value), or a negative value if a nonrecoverable error occurred. In the last case, the integrator halts. If a recoverable error occurred, the integrator will attempt to correct and retry.

Notes A recoverable failure error return from the `IDAResFn` is typically used to flag a value of the dependent variable `y` that is “illegal” in some way (e.g., negative where only a non-negative value is physically meaningful). If such a return is made, IDAS will attempt to recover (possibly repeating the Newton iteration, or reducing the step size) in order to avoid this recoverable error return.

For efficiency reasons, the DAE residual function is not evaluated at the converged solution of the nonlinear solver. Therefore, in general, a recoverable error in that converged value cannot be corrected. (It may be detected when the right-hand side function is called the first time during the following integration step, but a successful step cannot be undone.) However, if the user program also includes quadrature integration, the state variables can be checked for legality in the call to `IDAQuadRhsFn`, which is called at the converged solution of the nonlinear system, and therefore IDAS can be flagged to attempt to recover from such a situation. Also, if sensitivity analysis is performed with the staggered method, the DAE residual function is called at the converged solution of the nonlinear system, and a recoverable error at that point can be flagged, and IDAS will then try to correct it.

Allocation of memory for `yp` is handled within IDAS.

4.6.2 Error message handler function

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

`IDAErrHandlerFn`

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

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

Arguments

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

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

Notes `error_code` is negative for errors and positive (`IDA_WARNING`) for warnings. If a function that returns a pointer to memory encounters an error, it sets `error_code` to 0.

4.6.3 Error weight function

As an alternative to providing the relative and absolute tolerances, the user may provide a function of type `IDAEwtFn` to compute a vector `ewt` containing the multiplicative weights W_i used in the WRMS norm $\| v\|_{\mathrm{WRMS}} = \sqrt{(1/N)\sum_1^N (W_i \cdot v_i)^2}$. These weights will used in place of those defined by Eq. (2.7). The function type `IDAEwtFn` is defined as follows:

`IDAEwtFn`

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

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

Arguments

`y`	is the value of the dependent variable vector at which the weight vector is to be computed.
`ewt`	is the output vector containing the error weights.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.

Return value An `IDAEwtFn` function type must return 0 if it successfully set the error weights and -1 otherwise.

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

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

4.6.4 Rootfinding function

If a rootfinding problem is to be solved during the integration of the DAE system, the user must supply a C function of type `IDARootFn`, defined as follows:

`IDARootFn`

Definition	`typedef int (*IDARootFn)(realtype t, N_Vector y, N_Vector yp, realtype *gout, void *user_data);`
Purpose	This function computes a vector-valued function $g(t, y, \dot{y})$ such that the roots of the `nrtfn` components $g_i(t, y, \dot{y})$ are to be found during the integration.
Arguments	`t` is the current value of the independent variable. `y` is the current value of the dependent variable vector, $y(t)$. `yp` is the current value of $\dot{y}(t)$, the $t-$derivative of y. `gout` is the output array, of length `nrtfn`, with components $g_i(t, y, \dot{y})$. `user_data` is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.
Return value	An `IDARootFn` should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and `IDASolve` returns `IDA_RTFUNC_FAIL`).
Notes	Allocation of memory for `gout` is handled within IDAS.

4.6.5 Jacobian information (direct method with dense Jacobian)

If the direct linear solver with dense treatment of the Jacobian is used (i.e. either `IDADense` or `IDALapackDense` is called in Step 8 of §4.4), the user may provide a function of type `IDADlsDenseJacFn` defined by

`IDADlsDenseJacFn`

Definition
```
typedef int (*IDADlsDenseJacFn)(long int Neq, realtype tt, realtype cj,
                                N_Vector yy, N_Vector yp, N_Vector rr,
                                DlsMat Jac, void *user_data,
                                N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
```

Purpose: This function computes the dense Jacobian J of the DAE system (or an approximation to it), defined by Eq. (2.6).

Arguments	
`Neq`	is the problem size (number of equations).
`tt`	is the current value of the independent variable t.
`cj`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`yy`	is the current value of the dependent variable vector, $y(t)$.
`yp`	is the current value of $\dot{y}(t)$.
`rr`	is the current value of the residual vector $F(t, y, \dot{y})$.
`Jac`	is the output (approximate) Jacobian matrix, $J = \partial F/\partial y + cj\ \partial F/\partial \dot{y}$.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.
`tmp1`	
`tmp2`	
`tmp3`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDADlsDenseJacFn` as temporary storage or work space.

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

In the case of a recoverable eror return, the integrator will attempt to recover by reducing the stepsize, and hence changing α in (2.6).

Notes A user-supplied dense Jacobian function must load the `Neq` $\times$ `Neq` dense matrix `Jac` with an approximation to the Jacobian matrix $J(t, y, \dot{y})$ at the point (`tt`, `yy`, `yp`). Only nonzero elements need to be loaded into `Jac` because `Jac` is set to the zero matrix before the call to the Jacobian function. The type of `Jac` is `DlsMat` (described below and in §9.1).

The accessor macros `DENSE_ELEM` and `DENSE_COL` allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the `DlsMat` type. `DENSE_ELEM(Jac, i, j)` references the (`i`, `j`)-th element of the dense matrix `Jac` (`i`, `j`$= 0 \ldots$ `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 `DlsMat` type and the accessor macros `DENSE_ELEM` and `DENSE_COL` are documented in §9.1.

If the user's `IDADlsDenseJacFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to `ida_mem` to `user_data` and then use the `IDAGet*` functions described in §4.5.9.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

For the sake of uniformity, the argument `Neq` is of type `long int`, even in the case that the Lapack dense solver is to be used.

4.6.6 Jacobian information (direct method with banded Jacobian)

If the direct linear solver with banded treatment of the Jacobian is used (i.e. either `IDABand` or `IDALapackBand` is called in Step 8 of §4.4), the user may provide a function of type `IDADlsBandJacFn` defined as follows:

`IDADlsBandJacFn`

Definition
```
typedef int (*IDADlsBandJacFn)(long int Neq, long int mupper,
                               long int mlower, realtype tt, realtype cj,
                               N_Vector yy, N_Vector yp, N_Vector rr,
                               DlsMat Jac, void *user_data,
                               N_Vector tmp1, N_Vector tmp2,N_Vector tmp3);
```

Purpose This function computes the banded Jacobian J of the DAE system (or a banded approximation to it), defined by Eq. (2.6).

Arguments

`Neq`	is the problem size.
`mupper`	
`mlower`	are the upper and lower half bandwidth of the Jacobian.
`tt`	is the current value of the independent variable.
`yy`	is the current value of the dependent variable vector, $y(t)$.

`yp`	is the current value of $\dot{y}(t)$.
`rr`	is the current value of the residual vector $F(t, y, \dot{y})$.
`cj`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`Jac`	is the output (approximate) Jacobian matrix, $J = \partial F/\partial y + cj\ \partial F/\partial \dot{y}$.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.
`tmp1`	
`tmp2`	
`tmp3`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDADlsBandJacFn` as temporary storage or work space.

Return value A `IDADlsBandJacFn` function type should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if a nonrecoverable error occurred.

In the case of a recoverable eror return, the integrator will attempt to recover by reducing the stepsize, and hence changing α in (2.6).

Notes A user-supplied band Jacobian function must load the band matrix `Jac` of type `DlsMat` with the elements of the Jacobian $J(t, y, \dot{y})$ at the point (`tt`, `yy`, `yp`). Only nonzero elements need to be loaded into `Jac` because `Jac` is preset to zero before the call to the Jacobian function.

The accessor macros `BAND_ELEM`, `BAND_COL`, and `BAND_COL_ELEM` allow the user to read and write band matrix elements without making specific references to the underlying representation of the `DlsMat` type. `BAND_ELEM(Jac, i, j)` references the (`i`, `j`)th element of the band matrix `Jac`, counting from 0. This macro is for use in small problems in which efficiency of access is not a major concern. Thus, in terms of indices m and n running from 1 to `Neq` with (m, n) within the band defined by `mupper` and `mlower`, the Jacobian element $J_{m,n}$ can be loaded with the statement `BAND_ELEM(Jac, m-1, n-1) =` $J_{m,n}$. The elements within the band are those with `-mupper` $\leq$ `m-n` $\leq$ `mlower`. Alternatively, `BAND_COL(Jac, j)` returns a pointer to the diagonal element of the `j`th column of `Jac`, and if we assign this address to `realtype *col_j`, then the `i`th element of the `j`th column is given by `BAND_COL_ELEM(col_j, i, j)`, counting from 0. Thus for (m, n) within the band, $J_{m,n}$ can be loaded by setting `col_n = BAND_COL(Jac, n-1); BAND_COL_ELEM(col_n, m-1, n-1) =` $J_{m,n}$. The elements of the `j`th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type `DlsMat`. The array `col_n` can be indexed from $-$`mupper` to `mlower`. For large problems, it is more efficient to use the combination of `BAND_COL` and `BAND_COL_ELEM` than to use the `BAND_ELEM`. As in the dense case, these macros all number rows and columns starting from 0, not 1.

The `DlsMat` type and the accessor macros `BAND_ELEM`, `BAND_COL`, and `BAND_COL_ELEM` are documented in §9.1.

If the user's `IDADlsBandJacFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to `ida_mem` to `user_data` and then use the `IDAGet*` functions described in §4.5.9.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

For the sake of uniformity, the arguments `Neq`, `mlower`, and `mupper` are of type `long int`, even in the case that the Lapack band solver is to be used.

4.6.7 Jacobian information (direct method with sparse Jacobian)

If the direct linear solver with sparse treatment of the Jacobian is used (i.e. either `IDAKLU` or `IDASuperLUMT` is called in Step 8 of §4.4), the user must provide a function of type `IDASlsSparseJacFn` defined as follows:

`IDASlsSparseJacFn`

Definition
```
typedef int (*IDASlsSparseJacFn)(realtype t, realtype c_j,
                                 N_Vector y, N_Vector yp, N_Vector r,
                                 SlsMat Jac, void *user_data,
                                 N_Vector tmp1, N_Vector tmp2,N_Vector tmp3);
```

Purpose — This function computes the sparse Jacobian J of the DAE system (or an approximation to it), defined by Eq. (2.6).

Arguments

`t`	is the current value of the independent variable.
`y`	is the current value of the dependent variable vector, $y(t)$.
`yp`	is the current value of $\dot{y}(t)$.
`r`	is the current value of the residual vector $F(t, y, \dot{y})$.
`c_j`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`Jac`	is the output (approximate) Jacobian matrix, $J = \partial F/\partial y + cj\ \partial F/\partial \dot{y}$.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.
`tmp1`	
`tmp2`	
`tmp3`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDASlsSparseJacFn` as temporary storage or work space.

Return value — A `IDASlsSparseJacFn` function type should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if a nonrecoverable error occurred.

In the case of a recoverable error return, the integrator will attempt to recover by reducing the stepsize, and hence changing α in (2.6).

Notes — A user-supplied sparse Jacobian function must load the compressed-sparse-column matrix `Jac` with the elements of the Jacobian $J(t, y, \dot{y})$ at the point (`t`, `y`, `yp`). Storage for `Jac` already exists on entry to this function, although the user should ensure that sufficient space is allocated in `Jac` to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and row index arrays as needed. The type of `Jac` is `SlsMat`, and the amount of allocated space is available within the `SlsMat` structure as `NNZ`. The `SlsMat` type is further documented in the Section §9.2.

If the user's `IDASlsSparseJacFn` function uses difference quotient approximations to set the specific nonzero matrix entries, then it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to `ida_mem` to `user_data` and then use the `IDAGet*` functions described in §4.5.9.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

4.6.8 Jacobian information (matrix-vector product)

If one of the Krylov iterative linear solvers SPGMR, SPBCG, or SPTFQMR is selected (`IDASp*` is called in step 8 of §4.4), the user may provide a function of type `IDASpilsJacTimesVecFn`, described below, to compute matrix-vector products Jv. If such a function is not supplied, the default is a difference quotient approximation to these products.

`IDASpilsJacTimesVecFn`

Definition
```
typedef int (*IDASpilsJacTimesVecFn)(realtype tt, N_Vector yy,
                                      N_Vector yp, N_Vector rr,
                                      N_Vector v, N_Vector Jv,
                                      realtype cj, void *user_data,
                                      N_Vector tmp1, N_Vector tmp2);
```

Purpose This function computes the product Jv of the DAE system Jacobian J (or an approximation to it) and a given vector v, where J is defined by Eq. (2.6).

Arguments

`tt`	is the current value of the independent variable.
`yy`	is the current value of the dependent variable vector, $y(t)$.
`yp`	is the current value of $\dot{y}(t)$.
`rr`	is the current value of the residual vector $F(t, y, \dot{y})$.
`v`	is the vector by which the Jacobian must be multiplied to the right.
`Jv`	is the computed output vector.
`cj`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.
`tmp1`	
`tmp2`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDASpilsJacTimesVecFn` as temporary storage or work space.

Return value The value to be returned by the Jacobian-times-vector function should be 0 if successful. A nonzero value indicates that a nonrecoverable error occurred.

Notes If the user's `IDASpilsJacTimesVecFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to `ida_mem` to `user_data` and then use the `IDAGet*` functions described in §4.5.9.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

4.6.9 Preconditioning (linear system solution)

If preconditioning is used, then the user must provide a C function to solve the linear system $Pz = r$ where P is a left preconditioner matrix which approximates (at least crudely) the Jacobian matrix $J = \partial F/\partial y + cj\ \partial F/\partial \dot{y}$. This function must be of type `IDASpilsPrecSolveFn`, defined as follows:

`IDASpilsPrecSolveFn`

Definition
```
typedef int (*IDASpilsPrecSolveFn)(realtype tt, N_Vector yy,
                                    N_Vector yp, N_Vector rr,
                                    N_Vector rvec, N_Vector zvec,
                                    realtype cj, realtype delta,
                                    void *user_data, N_Vector tmp);
```

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

Arguments

`tt`	is the current value of the independent variable.
`yy`	is the current value of the dependent variable vector, $y(t)$.
`yp`	is the current value of $\dot{y}(t)$.
`rr`	is the current value of the residual vector $F(t, y, \dot{y})$.
`rvec`	is the right-hand side vector r of the linear system to be solved.
`zvec`	is the computed output vector.
`cj`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).

`delta`	is an input tolerance to be used if an iterative method is employed in the solution. In that case, the residual vector $Res = r - Pz$ of the system should be made less than `delta` in weighted l_2 norm, i.e., $\sqrt{\sum_i (Res_i \cdot ewt_i)^2} <$ `delta`. To obtain the `N_Vector` `ewt`, call `IDAGetErrWeights` (see §4.5.9.1).
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to the function `IDASetUserData`.
`tmp`	is a pointer to memory allocated for a variable of type `N_Vector` which can be used for work space.

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

4.6.10 Preconditioning (Jacobian data)

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

`IDASpilsPrecSetupFn`

Definition
```
typedef int (*IDASpilsPrecSetupFn)(realtype tt, N_Vector yy,
                                   N_Vector yp, N_Vector rr,
                                   realtype cj, void *user_data,
                                   N_Vector tmp1, N_Vector tmp2,
                                   N_Vector tmp3);
```

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

Arguments The arguments of an `IDASpilsPrecSetupFn` are as follows:

`tt`	is the current value of the independent variable.
`yy`	is the current value of the dependent variable vector, $y(t)$.
`yp`	is the current value of $\dot{y}(t)$.
`rr`	is the current value of the residual vector $F(t, y, \dot{y})$.
`cj`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to the function `IDASetUserData`.
`tmp1`	
`tmp2`	
`tmp3`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDASpilsPrecSetupFn` as temporary storage or work space.

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

Notes The operations performed by this function might include forming a crude approximate Jacobian, and performing an LU factorization on the resulting approximation.

Each call to the preconditioner setup function is preceded by a call to the `IDAResFn` user function with the same (`tt`, `yy`, `yp`) arguments. Thus the preconditioner setup function can use any auxiliary data that is computed and saved during the evaluation of the DAE residual.

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

If the user's `IDASpilsPrecSetupFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to `ida_mem` to `user_data` and then use the `IDAGet*` functions described in §4.5.9.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

4.7 Integration of pure quadrature equations

IDAS allows the DAE system to include *pure quadratures*. In this case, it is more efficient to treat the quadratures separately by excluding them from the nonlinear solution stage. To do this, begin by excluding the quadrature variables from the vectors `yy` and `yp` and the quadrature equations from within `res`. Thus a separate vector `yQ` of quadrature variables is to satisfy $(d/dt)\texttt{yQ} = f_Q(t, y, \dot{y})$. The following is an overview of the sequence of calls in a user's main program in this situation. Steps that are unchanged from the skeleton program presented in §4.4 are grayed out.

1. **[P] Initialize MPI**
2. **Set problem dimensions**

   [**S**] Set `N` to the problem size N (excluding quadrature variables), and `Nq` to the number of quadrature variables.

   [**P**] Set `Nlocal` to the local vector length (excluding quadrature variables), and `Nqlocal` to the local number of quadrature variables.
3. **Set vectors of initial values**
4. **Create IDAS object**
5. **Allocate internal memory**
6. **Set optional inputs**
7. **Attach linear solver module**
8. **Set linear solver optional inputs**
9. **Set vector of initial values for quadrature variables**

   Typically, the quadrature variables should be initialized to 0.
10. **Initialize quadrature integration**

    Call `IDAQuadInit` to specify the quadrature equation right-hand side function and to allocate internal memory related to quadrature integration. See §4.7.1 for details.
11. **Set optional inputs for quadrature integration**

    Call `IDASetQuadErrCon` to indicate whether or not quadrature variables should be used in the step size control mechanism. If so, one of the `IDAQuad*tolerances` functions must be called to specify the integration tolerances for quadrature variables. See §4.7.4 for details.
12. **Advance solution in time**
13. **Extract quadrature variables**

    Call `IDAGetQuad` or `IDAGetQuadDky` to obtain the values of the quadrature variables or their derivatives at the current time. See §4.7.3 for details.

14. **Get optional outputs**

15. **Get quadrature optional outputs**

    Call `IDAGetQuad*` functions to obtain optional output related to the integration of quadratures. See §4.7.5 for details.

16. **Deallocate memory for solution vectors and for the vector of quadrature variables**

17. **Free solver memory**

18. **[P] Finalize MPI**

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

4.7.1 Quadrature initialization and deallocation functions

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

`IDAQuadInit`

Call `flag = IDAQuadInit(ida_mem, rhsQ, yQ0);`

Description The function `IDAQuadInit` provides required problem specifications, allocates internal memory, and initializes quadrature integration.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block returned by `IDACreate`.
`rhsQ`	(`IDAQuadRhsFn`) is the C function which computes f_Q, the right-hand side of the quadrature equations. This function has the form `fQ(t, yy, yp, rhsQ, user_data)` (for full details see §4.7.6).
`yQ0`	(`N_Vector`) is the initial value of y_Q.

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

`IDA_SUCCESS`	The call to `IDAQuadInit` was successful.
`IDA_MEM_NULL`	The IDAS memory was not initialized by a prior call to `IDACreate`.
`IDA_MEM_FAIL`	A memory allocation request failed.

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

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

- Base value: `lenrw` = `lenrw` + (`maxord`+5)N_q
- If `IDAQuadSVtolerances` is called: `lenrw` = `lenrw` +N_q

and the size of the integer workspace is increased as follows:

- Base value: `leniw` = `leniw` + (`maxord`+5)N_q
- If `IDAQuadSVtolerances` is called: `leniw` = `leniw` +N_q

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

`IDAQuadReInit`

Call `flag = IDAQuadReInit(ida_mem, yQ0);`

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

Arguments

`ida_mem`	(`void *`)	pointer to the IDAS memory block.
`yQ0`	(`N_Vector`)	is the initial value of y_Q.

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

`IDA_SUCCESS`	The call to `IDAReInit` was successful.
`IDA_MEM_NULL`	The IDAS memory was not initialized by a prior call to `IDACreate`.
`IDA_NO_QUAD`	Memory space for the quadrature integration was not allocated by a prior call to `IDAQuadInit`.

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

`IDAQuadFree`

Call `IDAQuadFree(ida_mem);`

Description The function `IDAQuadFree` frees the memory allocated for quadrature integration.

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

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

Notes In general, `IDAQuadFree` need not be called by the user as it is invoked automatically by `IDAFree`.

4.7.2 IDAS solver function

Even if quadrature integration was enabled, the call to the main solver function `IDASolve` is exactly the same as in §4.5.6. However, in this case the return value `flag` can also be one of the following:

`IDA_QRHS_FAIL`	The quadrature right-hand side function failed in an unrecoverable manner.
`IDA_FIRST_QRHS_ERR`	The quadrature right-hand side function failed at the first call.
`IDA_REP_QRHS_ERR`	Convergence test failures occurred too many times due to repeated recoverable errors in the quadrature right-hand side function. This value will also be returned if the quadrature right-hand side function had repeated recoverable errors during the estimation of an initial step size (assuming the quadrature variables are included in the error tests).

4.7.3 Quadrature extraction functions

If quadrature integration has been initialized by a call to `IDAQuadInit`, or reinitialized by a call to `IDAQuadReInit`, then IDAS computes both a solution and quadratures at time `t`. However, `IDASolve` will still return only the solution y in `y`. Solution quadratures can be obtained using the following function:

`IDAGetQuad`

Call `flag = IDAGetQuad(ida_mem, &tret, yQ);`

Description The function `IDAGetQuad` returns the quadrature solution vector after a successful return from `IDASolve`.

Arguments

`ida_mem`	(`void *`)	pointer to the memory previously allocated by `IDAInit`.
`tret`	(`realtype`)	the time reached by the solver (output).

yQ	(N_Vector) the computed quadrature vector.

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

IDA_SUCCESS	IDAGetQuad was successful.
IDA_MEM_NULL	ida_mem was NULL.
IDA_NO_QUAD	Quadrature integration was not initialized.
IDA_BAD_DKY	yQ is NULL.

The function `IDAGetQuadDky` computes the `k`-th derivatives of the interpolating polynomials for the quadrature variables at time `t`. This function is called by `IDAGetQuad` with `k = 0` and with the current time at which `IDASolve` has returned, but may also be called directly by the user.

`IDAGetQuadDky`

Call `flag = IDAGetQuadDky(ida_mem, t, k, dkyQ);`

Description The function `IDAGetQuadDky` returns derivatives of the quadrature solution vector after a successful return from `IDASolve`.

Arguments

ida_mem	(void *) pointer to the memory previously allocated by IDAInit.
t	(realtype) the time at which quadrature information is requested. The time t must fall within the interval defined by the last successful step taken by IDAS.
k	(int) order of the requested derivative. This must be $\leq$ *klast*.
dkyQ	(N_Vector) the vector containing the derivative. This vector must be allocated by the user.

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

IDA_SUCCESS	IDAGetQuadDky succeeded.
IDA_MEM_NULL	The pointer to ida_mem was NULL.
IDA_NO_QUAD	Quadrature integration was not initialized.
IDA_BAD_DKY	The vector dkyQ is NULL.
IDA_BAD_K	k is not in the range $0, 1, ..., klast$.
IDA_BAD_T	The time t is not in the allowed range.

4.7.4 Optional inputs for quadrature integration

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

`IDASetQuadErrCon`

Call `flag = IDASetQuadErrCon(ida_mem, errconQ);`

Description The function `IDASetQuadErrCon` specifies whether or not the quadrature variables are to be used in the step size control mechanism within IDAS. If they are, the user must call either `IDAQuadSStolerances` or `IDAQuadSVtolerances` to specify the integration tolerances for the quadrature variables.

Arguments

ida_mem	(void *) pointer to the IDAS memory block.
errconQ	(booleantype) specifies whether quadrature variables are included (TRUE) or not (FALSE) in the error control mechanism.

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_NO_QUAD	Quadrature integration has not been initialized.

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

It is illegal to call `IDASetQuadErrCon` before a call to `IDAQuadInit`.

If the quadrature variables are part of the step size control mechanism, one of the following functions must be called to specify the integration tolerances for quadrature variables.

`IDAQuadSStolerances`

Call `flag = IDAQuadSVtolerances(ida_mem, reltolQ, abstolQ);`

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

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

`reltolQ` (`realtype`) is the scalar relative error tolerance.

`abstolQ` (`realtype`) is the scalar absolute error tolerance.

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

`IDA_SUCCESS`	The optional value has been successfully set.
`IDA_NO_QUAD`	Quadrature integration was not initialized.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT`	One of the input tolerances was negative.

`IDAQuadSVtolerances`

Call `flag = IDAQuadSVtolerances(ida_mem, reltolQ, abstolQ);`

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

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

`reltolQ` (`realtype`) is the scalar relative error tolerance.

`abstolQ` (`N_Vector`) is the vector absolute error tolerance.

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

`IDA_SUCCESS`	The optional value has been successfully set.
`IDA_NO_QUAD`	Quadrature integration was not initialized.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT`	One of the input tolerances was negative.

4.7.5 Optional outputs for quadrature integration

IDAS provides the following functions that can be used to obtain solver performance information related to quadrature integration.

`IDAGetQuadNumRhsEvals`

Call `flag = IDAGetQuadNumRhsEvals(ida_mem, &nrhsQevals);`

Description The function `IDAGetQuadNumRhsEvals` returns the number of calls made to the user's quadrature right-hand side function.

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

`nrhsQevals` (`long int`) number of calls made to the user's `rhsQ` function.

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

`IDA_SUCCESS`	The optional output value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_NO_QUAD`	Quadrature integration has not been initialized.

`IDAGetQuadNumErrTestFails`

Call `flag = IDAGetQuadNumErrTestFails(ida_mem, &nQetfails);`

Description The function `IDAGetQuadNumErrTestFails` returns the number of local error test failures due to quadrature variables.

Arguments `ida_mem` (`void *`) pointer to the IDAS memory block.
`nQetfails` (`long int`) number of error test failures due to quadrature variables.

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

`IDA_SUCCESS`	The optional output value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_NO_QUAD`	Quadrature integration has not been initialized.

`IDAGetQuadErrWeights`

Call `flag = IDAGetQuadErrWeights(ida_mem, eQweight);`

Description The function `IDAGetQuadErrWeights` returns the quadrature error weights at the current time.

Arguments `ida_mem` (`void *`) pointer to the IDAS memory block.
`eQweight` (`N_Vector`) quadrature error weights at the current time.

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

`IDA_SUCCESS`	The optional output value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_NO_QUAD`	Quadrature integration has not been initialized.

Notes The user must allocate memory for `eQweight`.

If quadratures were not included in the error control mechanism (through a call to `IDASetQuadErrCon` with `errconQ = TRUE`), `IDAGetQuadErrWeights` does not set the `eQweight` vector.

`IDAGetQuadStats`

Call `flag = IDAGetQuadStats(ida_mem, &nrhsQevals, &nQetfails);`

Description The function `IDAGetQuadStats` returns the IDAS integrator statistics as a group.

Arguments `ida_mem` (`void *`) pointer to the IDAS memory block.
`nrhsQevals` (`long int`) number of calls to the user's `rhsQ` function.
`nQetfails` (`long int`) number of error test failures due to quadrature variables.

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

`IDA_SUCCESS`	the optional output values have been successfully set.
`IDA_MEM_NULL`	the `ida_mem` pointer is `NULL`.
`IDA_NO_QUAD`	Quadrature integration has not been initialized.

4.7.6 User-supplied function for quadrature integration

For integration of quadrature equations, the user must provide a function that defines the right-hand side of the quadrature equations (in other words, the integrand function of the integral that must be evaluated). This function must be of type `IDAQuadRhsFn` defined as follows:

`IDAQuadRhsFn`

Definition
```
typedef int (*IDAQuadRhsFn)(realtype t, N_Vector yy, N_Vector yp,
                            N_Vector rhsQ, void *user_data);
```

Purpose This function computes the quadrature equation right-hand side for a given value of the independent variable t and state vectors y and $\dot{y}$.

Arguments

`t`	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 the dependent variable derivative vector, $\dot{y}(t)$.
`rhsQ`	is the output vector $f_Q(t, y, \dot{y})$.
`user_data`	is the `user_data` pointer passed to `IDASetUserData`.

Return value A `IDAQuadRhsFn` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `IDA_QRHS_FAIL` is returned).

Notes Allocation of memory for `rhsQ` is automatically handled within IDAS.

Both `y` and `rhsQ` are of type `N_Vector`, but they typically have different internal representations. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with IDAS do not perform any consistency checks with respect to their `N_Vector` arguments (see §7.1 and §7.2).

There is one situation in which recovery is not possible even if `IDAQuadRhsFn` function returns a recoverable error flag. This is when this occurs at the very first call to the `IDAQuadRhsFn` (in which case IDAS returns `IDA_FIRST_QRHS_ERR`).

4.8 A parallel band-block-diagonal preconditioner module

A principal reason for using a parallel DAE solver such as IDAS lies in the solution of partial differential equations (PDEs). Moreover, the use of a Krylov iterative method for the solution of many such problems is motivated by the nature of the underlying linear system of equations (2.5) that must be solved at each time step. The linear algebraic system is large, sparse, and structured. However, if a Krylov iterative method is to be effective in this setting, then a nontrivial preconditioner needs to be used. Otherwise, the rate of convergence of the Krylov iterative method is usually unacceptably slow. Unfortunately, an effective preconditioner tends to be problem-specific.

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used for several realistic, large-scale problems [23] and is included in a software module within the IDAS package. This module works with the parallel vector module NVECTOR_PARALLEL and generates a preconditioner that is a block-diagonal matrix with each block being a band matrix. The blocks need not have the same number of super- and sub-diagonals and these numbers may vary from block to block. This Band-Block-Diagonal Preconditioner module is called IDABBDPRE.

One way to envision these preconditioners is to think of the domain of the computational PDE problem as being subdivided into M non-overlapping sub-domains. Each of these sub-domains is then assigned to one of the M processors to be used to solve the DAE system. The basic idea is to isolate the preconditioning so that it is local to each processor, and also to use a (possibly cheaper) approximate residual function. This requires the definition of a new function $G(t, y, \dot{y})$ which approximates the function $F(t, y, \dot{y})$ in the definition of the DAE system (2.1). However, the user may set $G = F$. Corresponding to the domain decomposition, there is a decomposition of the solution vectors y and $\dot{y}$ into M disjoint blocks y_m and $\dot{y}_m$, and a decomposition of G into blocks G_m. The block G_m depends on y_m and $\dot{y}_m$, and also on components of $y_{m'}$ and $\dot{y}_{m'}$ associated with neighboring sub-domains (so-called ghost-cell data). Let $\bar{y}_m$ and $\bar{\dot{y}}_m$ denote y_m and $\dot{y}_m$ (respectively) augmented with those other components on which G_m depends. Then we have

$$G(t, y, \dot{y}) = [G_1(t, \bar{y}_1, \bar{\dot{y}}_1), G_2(t, \bar{y}_2, \bar{\dot{y}}_2), \ldots, G_M(t, \bar{y}_M, \bar{\dot{y}}_M)]^T \ , \tag{4.1}$$

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

The preconditioner associated with this decomposition has the form

$$P = diag[P_1, P_2, \ldots, P_M] \tag{4.2}$$

where

$$P_m \approx \partial G_m/\partial y_m + \alpha \partial G_m/\partial \dot{y}_m \tag{4.3}$$

This matrix is taken to be banded, with upper and lower half-bandwidths `mudq` and `mldq` defined as the number of non-zero diagonals above and below the main diagonal, respectively. The difference quotient approximation is computed using `mudq` + `mldq` +2 evaluations of G_m, but only a matrix of bandwidth `mukeep` + `mlkeep` +1 is retained.

Neither pair of parameters need be the true half-bandwidths of the Jacobians of the local block of G, if smaller values provide a more efficient preconditioner. Such an efficiency gain may occur if the couplings in the DAE system outside a certain bandwidth are considerably weaker than those within the band. Reducing `mukeep` and `mlkeep` while keeping `mudq` and `mldq` at their true values, discards the elements outside the narrower band. Reducing both pairs has the additional effect of lumping the outer Jacobian elements into the computed elements within the band, and requires more caution and experimentation.

The solution of the complete linear system

$$Px = b \tag{4.4}$$

reduces to solving each of the equations

$$P_m x_m = b_m \tag{4.5}$$

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

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

The IDABBDPRE module calls two user-provided functions to construct P: a required function `Gres` (of type `IDABBDLocalFn`) which approximates the residual function $G(t, y, \dot{y}) \approx F(t, y, \dot{y})$ and which is computed locally, and an optional function `Gcomm` (of type `IDABBDCommFn`) which performs all inter-process communication necessary to evaluate the approximate residual G. These are in addition to the user-supplied residual function `res`. Both functions take as input the same pointer `user_data` as passed by the user to `IDASetUserData` and passed to the user's function `res`. The user is responsible for providing space (presumably within `user_data`) for components of `yy` and `yp` that are communicated by `Gcomm` from the other processors, and that are then used by `Gres`, which should not do any communication.

`IDABBDLocalFn`

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

Purpose This `Gres` function computes $G(t, y, \dot{y})$. It loads the vector `gval` as a function of `tt`, `yy`, and `yp`.

Arguments

`Nlocal`	is the local vector length.
`tt`	is the value of the independent variable.
`yy`	is the dependent variable.
`yp`	is the derivative of the dependent variable.

`gval`	is the output vector.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.

Return value An `IDABBDLocalFn` function type should return 0 to indicate success, 1 for a recoverable error, or -1 for a non-recoverable error.

Notes This function must assume that all inter-processor communication of data needed to calculate `gval` has already been done, and this data is accessible within `user_data`.

The case where G is mathematically identical to F is allowed.

`IDABBDCommFn`

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

Purpose This `Gcomm` function performs all inter-processor communications necessary for the execution of the `Gres` function above, using the input vectors `yy` and `yp`.

Arguments

`Nlocal`	is the local vector length.
`tt`	is the value of the independent variable.
`yy`	is the dependent variable.
`yp`	is the derivative of the dependent variable.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.

Return value An `IDABBDCommFn` function type should return 0 to indicate success, 1 for a recoverable error, or -1 for a non-recoverable error.

Notes The `Gcomm` function is expected to save communicated data in space defined within the structure `user_data`.

Each call to the `Gcomm` function is preceded by a call to the residual function `res` with the same (`tt`, `yy`, `yp`) arguments. Thus `Gcomm` can omit any communications done by `res` if relevant to the evaluation of `Gres`. If all necessary communication was done in `res`, then `Gcomm` = `NULL` can be passed in the call to `IDABBDPrecInit` (see below).

Besides the header files required for the integration of the DAE problem (see §4.3), to use the IDABBDPRE module, the main program must include the header file `idas_bbdpre.h` which declares the needed function prototypes.

The following is a summary of the usage of this module and describes the sequence of calls in the user main program. Steps that are unchanged from the user main program presented in §4.4 are grayed-out.

1. **Initialize MPI**
2. **Set problem dimensions**
3. **Set vector of initial values**
4. **Create IDAS object**
5. **Allocate internal memory**
6. **Set optional inputs**
7. **Attach iterative linear solver, one of:**
   (a) `flag = IDASpgmr(ida_mem, maxl);`
   (b) `flag = IDASpbcg(ida_mem, maxl);`

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

8. **Initialize the IDABBDPRE preconditioner module**

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

   ```
   flag = IDABBDPrecInit(ida_mem, Nlocal, mudq, mldq,
                         mukeep, mlkeep, dq_rel_yy, Gres, Gcomm);
   ```

   to allocate memory and initialize the internal preconditioner data. The last two arguments of `IDABBDPrecInit` are the two user-supplied functions described above.

9. **Set linear solver optional inputs**

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

10. **Correct initial values**

11. **Specify rootfinding problem**

12. **Advance solution in time**

13. **Get optional outputs**

    Additional optional outputs associated with IDABBDPRE are available by way of two routines described below, `IDABBDPrecGetWorkSpace` and `IDABBDPrecGetNumGfnEvals`.

14. **Deallocate memory for solution vector**

15. **Free solver memory**

16. **Finalize MPI**

The user-callable functions that initialize (step 8 above) or re-initialize the IDABBDPRE preconditioner module are described next.

`IDABBDPrecInit`

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

Description The function `IDABBDPrecInit` initializes and allocates (internal) memory for the IDABBDPRE preconditioner.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`Nlocal`	(`long int`) local vector dimension.
`mudq`	(`long int`) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
`mldq`	(`long int`) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
`mukeep`	(`long int`) upper half-bandwidth of the retained banded approximate Jacobian block.
`mlkeep`	(`long int`) lower half-bandwidth of the retained banded approximate Jacobian block.
`dq_rel_yy`	(`realtype`) the relative increment in components of `y` used in the difference quotient approximations. The default is `dq_rel_yy`$=\sqrt{\text{unit roundoff}}$, which can be specified by passing `dq_rel_yy`= 0.0.
`Gres`	(`IDABBDLocalFn`) the C function which computes the local residual approximation $G(t, y, \dot{y})$.

`Gcomm`	(`IDABBDCommFn`) the optional C function which performs all inter-process communication required for the computation of $G(t, y, \dot{y})$.

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

`IDASPILS_SUCCESS`	The call to `IDABBDPrecInit` was successful.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer was `NULL`.
`IDASPILS_MEM_FAIL`	A memory allocation request has failed.
`IDASPILS_LMEM_NULL`	An IDASPILS linear solver memory was not attached.
`IDASPILS_ILL_INPUT`	The supplied vector implementation was not compatible with block band preconditioner.

Notes If one of the half-bandwidths `mudq` or `mldq` to be used in the difference-quotient calculation of the approximate Jacobian is negative or exceeds the value `Nlocal`-1, it is replaced by 0 or `Nlocal`-1 accordingly.

The half-bandwidths `mudq` and `mldq` need not be the true half-bandwidths of the Jacobian of the local block of G, when smaller values may provide a greater efficiency.

Also, the half-bandwidths `mukeep` and `mlkeep` of the retained banded approximate Jacobian block may be even smaller, to reduce storage and computation costs further.

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

The IDABBDPRE module also provides a reinitialization function to allow for a sequence of problems of the same size with IDASPGMR/IDABBDPRE, IDASPBCG/IDABBDPRE, or IDASPTFQMR/IDABBDPRE, provided there is no change in `local_N`, `mukeep`, or `mlkeep`. After solving one problem, and after calling `IDAReInit` to re-initialize IDAS for a subsequent problem, a call to `IDABBDPrecReInit` can be made to change any of the following: the half-bandwidths `mudq` and `mldq` used in the difference-quotient Jacobian approximations, the relative increment `dq_rel_yy`, or one of the user-supplied functions `Gres` and `Gcomm`.

`IDABBDPrecReInit`

Call `flag = IDABBDPrecReInit(ida_mem, mudq, mldq, dq_rel_yy);`

Description The function `IDABBDPrecReInit` reinitializes the IDABBDPRE preconditioner.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`mudq`	(`long int`) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
`mldq`	(`long int`) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
`dq_rel_yy`	(`realtype`) the relative increment in components of `y` used in the difference quotient approximations. The default is `dq_rel_yy` $= \sqrt{\text{unit roundoff}}$, which can be specified by passing `dq_rel_yy` $= 0.0$.

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

`IDASPILS_SUCCESS`	The call to `IDABBDPrecReInit` was successful.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer was `NULL`.
`IDASPILS_LMEM_NULL`	An IDASPILS linear solver memory was not attached.
`IDASPILS_PMEM_NULL`	The function `IDABBDPrecInit` was not previously called.

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

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

`IDABBDPrecGetWorkSpace`

Call	`flag = IDABBDPrecGetWorkSpace(ida_mem, &lenrwBBDP, &leniwBBDP);`
Description	The function `IDABBDPrecGetWorkSpace` returns the local sizes of the IDABBDPRE real and integer workspaces.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `lenrwBBDP` (`long int`) local number of real values in the IDABBDPRE workspace. `leniwBBDP` (`long int`) local number of integer values in the IDABBDPRE workspace.
Return value	The return value `flag` (of type `int`) is one of `IDASPILS_SUCCESS` The optional output value has been successfully set. `IDASPILS_MEM_NULL` The `ida_mem` pointer was `NULL`. `IDASPILS_PMEM_NULL` The IDABBDPRE preconditioner has not been initialized.
Notes	In terms of the local vector dimension N_l, and `smu` $= \min(N_l - 1,$ `mukeep` + `mlkeep`$)$, the actual size of the real workspace is $N_l\,(2$ `mlkeep` + `mukeep` + `smu` $+2)$ `realtype` words. The actual size of the integer workspace is N_l integer words.

`IDABBDPrecGetNumGfnEvals`

Call	`flag = IDABBDPrecGetNumGfnEvals(ida_mem, &ngevalsBBDP);`
Description	The function `IDABBDPrecGetNumGfnEvals` returns the cumulative number of calls to the user `Gres` function due to the finite difference approximation of the Jacobian blocks used within IDABBDPRE's preconditioner setup function.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `ngevalsBBDP` (`long int`) the cumulative number of calls to the user `Gres` function.
Return value	The return value `flag` (of type `int`) is one of `IDASPILS_SUCCESS` The optional output value has been successfully set. `IDASPILS_MEM_NULL` The `ida_mem` pointer was `NULL`. `IDASPILS_PMEM_NULL` The IDABBDPRE preconditioner has not been initialized.

In addition to the `ngevalsBBDP` `Gres` evaluations, the costs associated with IDABBDPRE also include `nlinsetups` LU factorizations, `nlinsetups` calls to `Gcomm`, `npsolves` banded backsolve calls, and `nrevalsLS` residual function evaluations, where `nlinsetups` is an optional IDAS output (see §4.5.9.1), and `npsolves` and `nrevalsLS` are linear solver optional outputs (see §4.5.9.6).

Chapter 5

Using IDAS for Forward Sensitivity Analysis

This chapter describes the use of IDAS to compute solution sensitivities using forward sensitivity analysis. One of our main guiding principles was to design the IDAS user interface for forward sensitivity analysis as an extension of that for IVP integration. Assuming a user main program and user-defined support routines for IVP integration have already been defined, in order to perform forward sensitivity analysis the user only has to insert a few more calls into the main program and (optionally) define an additional routine which computes the residuals for sensitivity systems (2.12). The only departure from this philosophy is due to the `IDAResFn` type definition (§4.6.1). Without changing the definition of this type, the only way to pass values of the problem parameters to the DAE residual function is to require the user data structure `user_data` to contain a pointer to the array of real parameters p.

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

We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable routines and of the user-supplied routines that were not already described in Chapter 4.

5.1 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) as an application of IDAS. The user program is to have these steps in the order indicated, unless otherwise noted. For the sake of brevity, we defer many of the details to the later sections. As in §4.4, most steps are independent of the NVECTOR implementation used; where this is not the case, usage specifications are given for the four implementations provided with IDAS: steps marked [**P**] correspond to NVECTOR_PARALLEL, steps marked [**O**] correspond to NVECTOR_OPENMP, steps marked [**T**] correspond to NVECTOR_PTHREADS, while steps marked [**S**] correspond to NVECTOR_SERIAL. Differences between the user main program in §4.4 and the one below start only at step (11). Steps that are unchanged from the skeleton program presented in §4.4 are grayed out.

First, note that no additional header files need be included for forward sensitivity analysis beyond those for IVP solution (§4.4).

1. **[P] Initialize MPI**

2. **Set problem dimensions**

3. **Set initial values**

4. **Create IDAS object**

5. **Allocate internal memory**

6. **Specify integration tolerances**

7. **Set optional inputs**

8. **Attach linear solver module**

9. **Set linear solver optional inputs**

10. **Initialize quadrature problem, if not sensitivity-dependent**

11. **Define the sensitivity problem**

    - Number of sensitivities (required)

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

    - Problem parameters (optional)

      If IDAS is to evaluate the residuals of the sensitivity systems, set `p`, an array of `Np` real parameters upon which the IVP depends. Only parameters with respect to which sensitivities are (potentially) desired need to be included. Attach `p` to the user data structure `user_data`. For example, `user_data->p = p;`

      If the user provides a function to evaluate the sensitivity residuals, `p` need not be specified.

    - Parameter list (optional)

      If IDAS is to evaluate the sensitivity residuals, set `plist`, an array of `Ns` integers to specify the parameters `p` with respect to which solution sensitivities are to be computed. If sensitivities with respect to the j-th parameter `p[j]` ($0 \leq$ `j` $<$ `Np`) are desired, set $\text{plist}_i = j$, for some $i = 0, \ldots, N_s - 1$.

      If `plist` is not specified, IDAS will compute sensitivities with respect to the first `Ns` parameters; i.e., $\text{plist}_i = i \;\; (i = 0, \ldots, N_s - 1)$.

      If the user provides a function to evaluate the sensitivity residuals, `plist` need not be specified.

    - Parameter scaling factors (optional)

      If IDAS is to estimate tolerances for the sensitivity solution vectors (based on tolerances for the state solution vector) or if IDAS is to evaluate the residuals of the sensitivity systems using the internal difference-quotient function, the results will be more accurate if order of magnitude information is provided.

      Set `pbar`, an array of `Ns` positive scaling factors. Typically, if $p_i \neq 0$, the value $\bar{p}_i = |p_{\text{plist}_i}|$ can be used.

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

      If the user provides a function to evaluate the sensitivity residual and specifies tolerances for the sensitivity variables, `pbar` need not be specified.

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

12. **Set sensitivity initial conditions**

    To set the sensitivities vectors `yS0` and `ypS0` to initial values, use functions defined by a particular NVECTOR implementation.

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

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

    [**S**] `yS0 = N_VCloneVectorArray_Serial(Ns, y0);`

    [**O**] `yS0 = N_VCloneVectorArray_OpenMP(Ns, y0);`

[**T**] `yS0 = N_VCloneVectorArray_Pthreads(Ns, y0);`

[**P**] `yS0 = N_VCloneVectorArray_Parallel(Ns, y0);`

and, for each $i = 0, \ldots,$Ns-1, load initial values for the i-th sensitivity vector into the structure defined by:

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

[**O**] `NV_DATA_OMP(yS0[i])`

[**T**] `NV_DATA_PT(yS0[i])`

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

Here the argument `y0` serves only to provide the `N_Vector` type for cloning.

Alternatively, if the initial values for the sensitivity variables are already available in `realtype` arrays, create an array of `Ns` "empty" vectors by making the call

[**S**] `yS0 = N_VCloneEmptyVectorArray_Serial(Ns, y0);`

[**O**] `yS0 = N_VCloneEmptyVectorArray_OpenMP(Ns, y0);`

[**T**] `yS0 = N_VCloneEmptyVectorArray_Pthreads(Ns, y0);`

[**P**] `yS0 = N_VCloneEmptyVectorArray_Parallel(Ns, y0);`

and then attach the `realtype` array `yS0_i` containing the initial values of the i-th sensitivity vector using

[**S**] `N_VSetArrayPointer_Serial(yS0_i, yS0[i]);`

[**O**] `N_VSetArrayPointer_OpenMP(yS0_i, yS0[i]);`

[**T**] `N_VSetArrayPointer_Pthreads(yS0_i, yS0[i]);`

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

for $i = 0, \cdots$ `Ns` -1.

The initial conditions for the sensitivity derivatives `ypS0` of $\dot{y}$ are set similarly.

13. **Activate sensitivity calculations**

    Call `flag = IDASensInit(...);` to activate forward sensitivity computations and allocate internal memory for IDAS related to sensitivity calculations (see §5.2.1).

14. **Set sensitivity tolerances**

    Call `IDASensSStolerances`, `IDASensSVtolerances`, or `IDASensEEtolerances`. See §5.2.2.

15. **Set sensitivity analysis optional inputs**

    Call `IDASetSens*` routines to change from their default values any optional inputs that control the behavior of IDAS in computing forward sensitivities. See §5.2.6.

16. **Correct initial values**

17. **Specify rootfinding problem**

18. **Advance solution in time**

19. **Extract sensitivity solution**

    After each successful return from `IDASolve`, the solution of the original IVP is available in the `y` argument of `IDASolve`, while the sensitivity solution can be extracted into `yS` and `ypS` (which can be the same as `yS0` and `ypS0`, respectively) by calling one of the following routines: `IDAGetSens`, `IDAGetSens1`, `IDAGetSensDky` or `IDAGetSensDky1` (see §5.2.5).

20. Deallocate memory for solutions vector

21. **Deallocate memory for sensitivity vectors**

    Upon completion of the integration, deallocate memory for the vectors contained in `yS0` and `ypS0`:

    [**S**] `N_VDestroyVectorArray_Serial(yS0, Ns);`

    [**O**] `N_VDestroyVectorArray_OpenMP(yS0, Ns);`

    [**T**] `N_VDestroyVectorArray_Pthreads(yS0, Ns);`

    [**P**] `N_VDestroyVectorArray_Parallel(yS0, Ns);`

    and similarly for `ypS0`.

    If `yS` was created from `realtype` arrays `yS_i`, it is the user's responsibility to also free the space for the arrays `yS_i`, and likewise for `ypS`.

22. **Free user data structure**

23. Free solver memory

24. Free vector specification memory

5.2 User-callable routines for forward sensitivity analysis

This section describes the IDAS functions, in addition to those presented in §4.5, that are called by the user to set up and solve a forward sensitivity problem.

5.2.1 Forward sensitivity initialization and deallocation functions

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

`IDASensInit`

Call `flag = IDASensInit(ida_mem, Ns, ism, resS, yS0, ypS0);`

Description The routine `IDASensInit` activates forward sensitivity computations and allocates internal memory related to sensitivity calculations.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block returned by `IDACreate`.
`Ns`	(`int`) the number of sensitivities to be computed.
`ism`	(`int`) a flag used to select the sensitivity solution method. Its value can be either `IDA_SIMULTANEOUS` or `IDA_STAGGERED`: • In the `IDA_SIMULTANEOUS` approach, the state and sensitivity variables are corrected at the same time. If `IDA_NEWTON` was selected as the nonlinear system solution method, this amounts to performing a modified Newton iteration on the combined nonlinear system; • In the `IDA_STAGGERED` approach, the correction step for the sensitivity variables takes place at the same time for all sensitivity equations, but only after the correction of the state variables has converged and the state variables have passed the local error test;
`resS`	(`IDASensResFn`) is the C function which computes the residual of the sensitivity DAE. For full details see §5.3.
`yS0`	(`N_Vector *`) a pointer to an array of `Ns` vectors containing the initial values of the sensitivities of y.
`ypS0`	(`N_Vector *`) a pointer to an array of `Ns` vectors containing the initial values of the sensitivities of $\dot{y}$.

Return value	The return value `flag` (of type `int`) will be one of the following:	
	`IDA_SUCCESS`	The call to `IDASensInit` was successful.
	`IDA_MEM_NULL`	The IDAS 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 `IDASensInit` has an illegal value.
Notes	Passing `resS=NULL` indicates using the default internal difference quotient sensitivity residual routine.	
	If an error occurred, `IDASensInit` also prints an error message to the file specified by the optional input `errfp`.	

In terms of the problem size N, number of sensitivity vectors N_s, and maximum method order `maxord`, the size of the real workspace is increased as follows:

- Base value: `lenrw` = `lenrw` + (`maxord+5`)$N_s N$
- With `IDASensSVtolerances`: `lenrw` = `lenrw` $+N_s N$

the size of the integer workspace is increased as follows:

- Base value: `leniw` = `leniw` + (`maxord+5`)$N_s N_i$
- With `IDASensSVtolerances`: `leniw` = `leniw` $+N_s N_i$,

where N_i is the number of integer words in one `N_Vector`.

The routine `IDASensReInit`, useful during the solution of a sequence of problems of same size, reinitializes the sensitivity-related internal memory and must follow a call to `IDASensInit` (and maybe a call to `IDAReInit`). The number `Ns` of sensitivities is assumed to be unchanged since the call to `IDASensInit`. The call to the `IDASensReInit` function has the form:

`IDASensReInit`

Call	`flag = IDASensReInit(ida_mem, ism, yS0, ypS0);`	
Description	The routine `IDASensReInit` reinitializes forward sensitivity computations.	
Arguments	`ida_mem`	(`void *`) pointer to the IDAS memory block returned by `IDACreate`.
	`ism`	(`int`) a flag used to select the sensitivity solution method. Its value can be either `IDA_SIMULTANEOUS` or `IDA_STAGGERED`.
	`yS0`	(`N_Vector *`) a pointer to an array of `Ns` variables of type `N_Vector` containing the initial values of the sensitivities of y.
	`ypS0`	(`N_Vector *`) a pointer to an array of `Ns` variables of type `N_Vector` containing the initial values of the sensitivities of $\dot{y}$.
Return value	The return value `flag` (of type `int`) will be one of the following:	
	`IDA_SUCCESS`	The call to `IDAReInit` was successful.
	`IDA_MEM_NULL`	The IDAS memory block was not initialized through a previous call to `IDACreate`.
	`IDA_NO_SENS`	Memory space for sensitivity integration was not allocated through a previous call to `IDASensInit`.
	`IDA_ILL_INPUT`	An input argument to `IDASensReInit` has an illegal value.
	`IDA_MEM_FAIL`	A memory allocation request has failed.
Notes	All arguments of `IDASensReInit` are the same as those of `IDASensInit`.	
	If an error occurred, `IDASensReInit` also prints an error message to the file specified by the optional input `errfp`.	

To deallocate all forward sensitivity-related memory (allocated in a prior call to `IDASensInit`), the user must call

`IDASensFree`

Call	`IDASensFree(ida_mem);`
Description	The function `IDASensFree` frees the memory allocated for forward sensitivity computations by a previous call to `IDASensInit`.
Arguments	The argument is the pointer to the IDAS memory block (of type `void *`).
Return value	The function `IDASensFree` has no return value.
Notes	In general, `IDASensFree` need not be called by the user as it is invoked automatically by `IDAFree`. After a call to `IDASensFree`, forward sensitivity computations can be reactivated only by calling `IDASensInit` again.

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

`IDASensToggleOff`

Call	`IDASensToggleOff(ida_mem);`
Description	The function `IDASensToggleOff` deactivates forward sensitivity calculations. It does *not* deallocate sensitivity-related memory.
Arguments	`ida_mem` (`void *`) pointer to the memory previously allocated by `IDAInit`.
Return value	The return value `flag` of `IDASensToggle` is one of:

`IDA_SUCCESS`	`IDASensToggleOff` was successful.
`IDA_MEM_NULL`	`ida_mem` was `NULL`.

Notes: Since sensitivity-related memory is not deallocated, sensitivities can be reactivated at a later time (using `IDASensReInit`).

5.2.2 Forward sensitivity tolerance specification functions

One of the following three functions must be called to specify the integration tolerances for sensitivities. Note that this call must be made after the call to `IDASensInit`.

`IDASensSStolerances`

Call	`flag = IDASensSStolerances(ida_mem, reltolS, abstolS);`
Description	The function `IDASensSStolerances` specifies scalar relative and absolute tolerances.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block returned by `IDACreate`.
	`reltolS` (`realtype`) is the scalar relative error tolerance.
	`abstolS` (`realtype*`) is a pointer to an array of length `Ns` containing the scalar absolute error tolerances.
Return value	The return flag `flag` (of type `int`) will be one of the following:

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

`IDASensSVtolerances`

Call `flag = IDASensSVtolerances(ida_mem, reltolS, abstolS);`

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

Arguments
`ida_mem` (`void *`) pointer to the IDAS memory block returned by `IDACreate`.
`reltolS` (`realtype`) is the scalar relative error tolerance.
`abstolS` (`N_Vector*`) is an array of `Ns` variables of type `N_Vector`. The `N_Vector` from `abstolS[is]` specifies the vector tolerances for `is`-th sensitivity.

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

`IDA_SUCCESS`	The call to `IDASVtolerances` was successful.
`IDA_MEM_NULL`	The IDAS memory block was not initialized through a previous call to `IDACreate`.
`IDA_NO_SENS`	The sensitivity allocation function `IDASensInit` has not been called.
`IDA_ILL_INPUT`	The relative error tolerance was negative or one of the absolute tolerance vectors had a negative component.

Notes This choice of tolerances is important when the absolute error tolerance needs to be different for each component of any vector `yS[i]`.

`IDASensEEtolerances`

Call `flag = IDASensEEtolerances(ida_mem);`

Description When `IDASensEEtolerances` is called, IDAS will estimate tolerances for sensitivity variables based on the tolerances supplied for states variables and the scaling factors $\bar{p}$.

Arguments `ida_mem` (`void *`) pointer to the IDAS memory block returned by `IDACreate`.

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

`IDA_SUCCESS`	The call to `IDASensEEtolerances` was successful.
`IDA_MEM_NULL`	The IDAS memory block was not initialized through a previous call to `IDACreate`.
`IDA_NO_SENS`	The sensitivity allocation function `IDASensInit` has not been called.

5.2.3 Forward sensitivity initial condition calculation function

`IDACalcIC` also calculates corrected initial conditions for sensitivity variables of a DAE system. When used for initial conditions calculation of the forward sensitivities, `IDACalcIC` must be preceded by successful calls to `IDASensInit` (or `IDASensReInit`) and should precede the call(s) to `IDASolve`. For restrictions that apply for initial conditions calculation of the state variables, see §4.5.4.

Calling `IDACalcIC` is optional. It is only necessary when the initial conditions do not satisfy the sensitivity systems. Even if forward sensitivity analysis was enabled, the call to the initial conditions calculation function `IDACalcIC` is exactly the same as for state variables.

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

See §4.5.4 for a list of possible return values.

5.2.4 IDAS solver function

Even if forward sensitivity analysis was enabled, the call to the main solver function `IDASolve` is exactly the same as in §4.5.6. However, in this case the return value `flag` can also be one of the following:

`IDA_SRES_FAIL`	The sensitivity residual function failed in an unrecoverable manner.
`IDA_REP_SRES_ERR`	The user's residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.

5.2.5 Forward sensitivity extraction functions

If forward sensitivity computations have been initialized by a call to `IDASensInit`, or reinitialized by a call to `IDASensReInit`, then IDAS computes both a solution and sensitivities at time `t`. However, `IDASolve` will still return only the solutions y and $\dot{y}$ in `yret` and `ypret`, respectively. Solution sensitivities can be obtained through one of the following functions:

`IDAGetSens`

Call	`flag = IDAGetSens(ida_mem, &tret, yS);`
Description	The function `IDAGetSens` returns the sensitivity solution vectors after a successful return from `IDASolve`.
Arguments	`ida_mem` (`void *`) pointer to the memory previously allocated by `IDAInit`.
	`tret` (`realtype`) the time reached by the solver (output).
	`yS` (`N_Vector *`) the array of `Ns` computed forward sensitivity vectors.
Return value	The return value `flag` of `IDAGetSens` is one of:

`IDA_SUCCESS`	`IDAGetSens` was successful.
`IDA_MEM_NULL`	`ida_mem` was `NULL`.
`IDA_NO_SENS`	Forward sensitivity analysis was not initialized.
`IDA_BAD_DKY`	`yS` is `NULL`.

Notes: Note that the argument `tret` is an output for this function. Its value will be the same as that returned at the last `IDASolve` call.

The function `IDAGetSensDky` computes the `k`-th derivatives of the interpolating polynomials for the sensitivity variables at time `t`. This function is called by `IDAGetSens` with `k` $= 0$, but may also be called directly by the user.

`IDAGetSensDky`

Call	`flag = IDAGetSensDky(ida_mem, t, k, dkyS);`
Description	The function `IDAGetSensDky` returns derivatives of the sensitivity solution vectors after a successful return from `IDASolve`.
Arguments	`ida_mem` (`void *`) pointer to the memory previously allocated by `IDAInit`.
	`t` (`realtype`) specifies the time at which sensitivity information is requested. The time `t` must fall within the interval defined by the last successful step taken by IDAS.
	`k` (`int`) order of derivatives.
	`dkyS` (`N_Vector *`) array of `Ns` vectors containing the derivatives on output. The space for `dkyS` must be allocated by the user.
Return value	The return value `flag` of `IDAGetSensDky` is one of:

`IDA_SUCCESS`	`IDAGetSensDky` succeeded.
`IDA_MEM_NULL`	`ida_mem` was `NULL`.
`IDA_NO_SENS`	Forward sensitivity analysis was not initialized.
`IDA_BAD_DKY`	`dkyS` or one of the vectors `dkyS[i]` is `NULL`.
`IDA_BAD_K`	`k` is not in the range $0, 1, ..., klast$.
`IDA_BAD_T`	The time `t` is not in the allowed range.

Forward sensitivity solution vectors can also be extracted separately for each parameter in turn through the functions `IDAGetSens1` and `IDAGetSensDky1`, defined as follows:

`IDAGetSens1`

Call	`flag = IDAGetSens1(ida_mem, &tret, is, yS);`
Description	The function `IDAGetSens1` returns the `is`-th sensitivity solution vector after a successful return from `IDASolve`.
Arguments	`ida_mem` (`void *`) pointer to the memory previously allocated by `IDAInit`.
	`tret` (`realtype *`) the time reached by the solver (output).
	`is` (`int`) specifies which sensitivity vector is to be returned ($0 \leq$`is`$< N_s$).
	`yS` (`N_Vector`) the computed forward sensitivity vector.
Return value	The return value `flag` of `IDAGetSens1` is one of:

`IDA_SUCCESS`	`IDAGetSens1` was successful.
`IDA_MEM_NULL`	`ida_mem` was `NULL`.
`IDA_NO_SENS`	Forward sensitivity analysis was not initialized.
`IDA_BAD_IS`	The index `is` is not in the allowed range.
`IDA_BAD_DKY`	`yS` is `NULL`.
`IDA_BAD_T`	The time `t` is not in the allowed range.

Notes: Note that the argument `tret` is an output for this function. Its value will be the same as that returned at the last `IDASolve` call.

`IDAGetSensDky1`

Call	`flag = IDAGetSensDky1(ida_mem, t, k, is, dkyS);`
Description	The function `IDAGetSensDky1` returns the `k`-th derivative of the `is`-th sensitivity solution vector after a successful return from `IDASolve`.
Arguments	`ida_mem` (`void *`) pointer to the memory previously allocated by `IDAInit`.
	`t` (`realtype`) specifies the time at which sensitivity information is requested. The time `t` must fall within the interval defined by the last successful step taken by IDAS.
	`k` (`int`) order of derivative.
	`is` (`int`) specifies the sensitivity derivative vector to be returned ($0 \leq$`is`$< N_s$).
	`dkyS` (`N_Vector`) the vector containing the derivative on output. The space for `dkyS` must be allocated by the user.
Return value	The return value `flag` of `IDAGetSensDky1` is one of:

`IDA_SUCCESS`	`IDAGetQuadDky1` succeeded.
`IDA_MEM_NULL`	`ida_mem` was `NULL`.
`IDA_NO_SENS`	Forward sensitivity analysis was not initialized.
`IDA_BAD_DKY`	`dkyS` is `NULL`.
`IDA_BAD_IS`	The index `is` is not in the allowed range.
`IDA_BAD_K`	`k` is not in the range $0, 1, ..., klast$.
`IDA_BAD_T`	The time `t` is not in the allowed range.

5.2.6 Optional inputs for forward sensitivity analysis

Optional input variables that control the computation of sensitivities can be changed from their default values through calls to `IDASetSens*` functions. Table 5.1 lists all forward sensitivity optional input functions in IDAS which are described in detail in the remainder of this section.

`IDASetSensParams`

Call `flag = IDASetSensParams(ida_mem, p, pbar, plist);`

Description The function `IDASetSensParams` specifies problem parameter information for sensitivity calculations.

Arguments
- `ida_mem` (`void *`) pointer to the IDAS memory block.
- `p` (`realtype *`) a pointer to the array of real problem parameters used to evaluate $F(t, y, \dot{y}, p)$. If non-`NULL`, `p` must point to a field in the user's data structure `user_data` passed to the user's residual function. (See §5.1).
- `pbar` (`realtype *`) an array of `Ns` positive scaling factors. If non-`NULL`, `pbar` must have all its components > 0.0. (See §5.1).
- `plist` (`int *`) an array of `Ns` non-negative indices to specify which components of `p` to use in estimating the sensitivity equations. If non-`NULL`, `plist` must have all components ≥ 0. (See §5.1).

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_NO_SENS`	Forward sensitivity analysis was not initialized.
`IDA_ILL_INPUT`	An argument has an illegal value.

Notes This function must be preceded by a call to `IDASensInit`.

`IDASetSensDQMethod`

Call `flag = IDASetSensDQMethod(ida_mem, DQtype, DQrhomax);`

Description The function `IDASetSensDQMethod` specifies the difference quotient strategy in the case in which the residual of the sensitivity equations are to be computed by IDAS.

Arguments
- `ida_mem` (`void *`) pointer to the IDAS memory block.
- `DQtype` (`int`) specifies the difference quotient type and can be either `IDA_CENTERED` or `IDA_FORWARD`.
- `DQrhomax` (`realtype`) positive value of the selection parameter used in deciding switching between a simultaneous or separate approximation of the two terms in the sensitivity residual.

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`	An argument has an illegal value.

Notes If `DQrhomax` $= 0.0$, then no switching is performed. The approximation is done simultaneously using either centered or forward finite differences, depending on the value of `DQtype`. For values of `DQrhomax` ≥ 1.0, the simultaneous approximation is used whenever the estimated finite difference perturbations for states and parameters are within

Table 5.1: Forward sensitivity optional inputs

Optional input	**Routine name**	**Default**
Sensitivity scaling factors	`IDASetSensParams`	`NULL`
DQ approximation method	`IDASetSensDQMethod`	centered,0.0
Error control strategy	`IDASetSensErrCon`	`FALSE`
Maximum no. of nonlinear iterations	`IDASetSensMaxNonlinIters`	3

a factor of `DQrhomax`, and the separate approximation is used otherwise. Note that a value `DQrhomax` < 1.0 will effectively disable switching. See §2.5 for more details.

The default value are `DQtype`=`IDA_CENTERED` and `DQrhomax`= 0.0.

`IDASetSensErrCon`

Call `flag = IDASetSensErrCon(ida_mem, errconS);`

Description The function `IDASetSensErrCon` specifies the error control strategy for sensitivity variables.

Arguments `ida_mem` (`void *`) pointer to the IDAS memory block.
`errconS` (`booleantype`) specifies whether sensitivity variables are included (`TRUE`) or not (`FALSE`) in the error control mechanism.

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

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

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

`IDASetSensMaxNonlinIters`

Call `flag = IDASetSensMaxNonlinIters(ida_mem, maxcorS);`

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

Arguments `ida_mem` (`void *`) pointer to the IDAS memory block.
`maxcorS` (`int`) maximum number of nonlinear solver iterations allowed per step (> 0).

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

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

Notes The default value is 3.

5.2.7 Optional outputs for forward sensitivity analysis

5.2.7.1 Main solver optional output functions

Optional output functions that return statistics and solver performance information related to forward sensitivity computations are listed in Table 5.2 and described in detail in the remainder of this section.

`IDAGetSensNumResEvals`

Call `flag = IDAGetSensNumResEvals(ida_mem, &nfSevals);`

Description The function `IDAGetSensNumResEvals` returns the number of calls to the sensitivity residual function.

Arguments `ida_mem` (`void *`) pointer to the IDAS memory block.
`nfSevals` (`long int`) number of calls to the sensitivity residual function.

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

`IDA_SUCCESS`	The optional output value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_NO_SENS`	Forward sensitivity analysis was not initialized.

`IDAGetNumResEvalsSens`

Call	`flag = IDAGetNumResEvalsSens(ida_mem, &nfevalsS);`
Description	The function `IDAGetNumResEvalsSEns` returns the number of calls to the user's residual function due to the internal finite difference approximation of the sensitivity residuals.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `nfevalsS` (`long int`) number of calls to the user residual function for sensitivity residuals.
Return value	The return value `flag` (of type `int`) is one of: `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDA_NO_SENS` Forward sensitivity analysis was not initialized.
Notes	This counter is incremented only if the internal finite difference approximation routines are used for the evaluation of the sensitivity residuals.

`IDAGetSensNumErrTestFails`

Call	`flag = IDAGetSensNumErrTestFails(ida_mem, &nSetfails);`
Description	The function `IDAGetSensNumErrTestFails` returns the number of local error test failures for the sensitivity variables that have occurred.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `nSetfails` (`long int`) number of error test failures.
Return value	The return value `flag` (of type `int`) is one of: `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDA_NO_SENS` Forward sensitivity analysis was not initialized.
Notes	This counter is incremented only if the sensitivity variables have been included in the error test (see `IDASetSensErrCon` in §5.2.6). Even in that case, this counter is not incremented if the `ism=IDA_SIMULTANEOUS` sensitivity solution method has been used.

`IDAGetSensNumLinSolvSetups`

Call	`flag = IDAGetSensNumLinSolvSetups(ida_mem, &nlinsetupsS);`
Description	The function `IDAGetSensNumLinSolvSetups` returns the number of calls to the linear solver setup function due to forward sensitivity calculations.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block.

Table 5.2: Forward sensitivity optional outputs

Optional output	**Routine name**
No. of calls to sensitivity residual function	`IDAGetSensNumResEvals`
No. of calls to residual function for sensitivity	`IDAGetNumResEvalsSens`
No. of sensitivity local error test failures	`IDAGetSensNumErrTestFails`
No. of calls to lin. solv. setup routine for sens.	`IDAGetSensNumLinSolvSetups`
Sensitivity-related statistics as a group	`IDAGetSensStats`
Error weight vector for sensitivity variables	`IDAGetSensErrWeights`
No. of sens. nonlinear solver iterations	`IDAGetSensNumNonlinSolvIters`
No. of sens. convergence failures	`IDAGetSensNumNonlinSolvConvFails`
Sens. nonlinear solver statistics as a group	`IDAGetSensNonlinSolvStats`

	`nlinsetupsS` (`long int`) number of calls to the linear solver setup function.
Return value	The return value `flag` (of type `int`) is one of: `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDA_NO_SENS` Forward sensitivity analysis was not initialized.
Notes	This counter is incremented only if Newton iteration has been used and staggered sensitivity solution method (`ism`=`IDA_STAGGERED`) was specified in the call to `IDASensInit` (see §5.2.1).

`IDAGetSensStats`

Call	`flag = IDAGetSensStats(ida_mem, &nfSevals, &nfevalsS, &nSetfails, &nlinsetupsS);`
Description	The function `IDAGetSensStats` returns all of the above sensitivity-related solver statistics as a group.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `nfSevals` (`long int`) number of calls to the sensitivity residual function. `nfevalsS` (`long int`) number of calls to the user-supplied residual function. `nSetfails` (`long int`) number of error test failures. `nlinsetupsS` (`long int`) number of calls to the linear solver setup function.
Return value	The return value `flag` (of type `int`) is one of: `IDA_SUCCESS` The optional output values have been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDA_NO_SENS` Forward sensitivity analysis was not initialized.

`IDAGetSensErrWeights`

Call	`flag = IDAGetSensErrWeights(ida_mem, eSweight);`
Description	The function `IDAGetSensErrWeights` returns the sensitivity error weight vectors at the current time. These are the reciprocals of the W_i of (2.7) for the sensitivity variables.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `eSweight` (`N_Vector_S`) pointer to the array of error weight vectors.
Return value	The return value `flag` (of type `int`) is one of: `IDA_SUCCESS` The optional output value has been successfully set. `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`. `IDA_NO_SENS` Forward sensitivity analysis was not initialized.
Notes	The user must allocate memory for `eweightS`.

`IDAGetSensNumNonlinSolvIters`

Call	`flag = IDAGetSensNumNonlinSolvIters(ida_mem, &nSniters);`
Description	The function `IDAGetSensNumNonlinSolvIters` returns the number of nonlinear iterations performed for sensitivity calculations.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `nSniters` (`long int`) number of nonlinear iterations performed.
Return value	The return value `flag` (of type `int`) is one of: `IDA_SUCCESS` The optional output value has been successfully set.

`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_NO_SENS`	Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if `ism` was `IDA_STAGGERED` in the call to `IDASensInit` (see §5.2.1).

`IDAGetSensNumNonlinSolvConvFails`

Call `flag = IDAGetSensNumNonlinSolvConvFails(ida_mem, &nSncfails);`

Description The function `IDAGetSensNumNonlinSolvConvFails` returns the number of nonlinear convergence failures that have occurred for sensitivity calculations.

Arguments `ida_mem` (`void *`) pointer to the IDAS memory block.
`nSncfails` (`long int`) number of nonlinear convergence failures.

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

`IDA_SUCCESS`	The optional output value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_NO_SENS`	Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if `ism` was `IDA_STAGGERED` in the call to `IDASensInit` (see §5.2.1).

`IDAGetSensNonlinSolvStats`

Call `flag = IDAGetSensNonlinSolvStats(ida_mem, &nSniters, &nSncfails);`

Description The function `IDAGetSensNonlinSolvStats` returns the sensitivity-related nonlinear solver statistics as a group.

Arguments `ida_mem` (`void *`) pointer to the IDAS memory block.
`nSniters` (`long int`) number of nonlinear iterations performed.
`nSncfails` (`long int`) number of nonlinear convergence failures.

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

`IDA_SUCCESS`	The optional output values have been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_NO_SENS`	Forward sensitivity analysis was not initialized.

5.2.7.2 Initial condition calculation optional output functions

The sensitivity consistent initial conditions found by IDAS (after a successful call to `IDACalcIC`) can be obtained by calling the following function:

`IDAGetSensConsistentIC`

Call `flag = IDAGetSensConsistentIC(ida_mem, yyS0_mod, ypS0_mod);`

Description The function `IDAGetSensConsistentIC` returns the corrected initial conditions calculated by `IDACalcIC` for sensitivities variables.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`yyS0_mod`	(`N_Vector *`) a pointer to an array of `Ns` vectors containing consistent sensitivity vectors.
`ypS0_mod`	(`N_Vector *`) a pointer to an array of `Ns` vectors containing consistent sensitivity derivative vectors.

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

`IDA_SUCCESS`	`IDAGetSensConsistentIC` succeeded.

`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_NO_SENS`	The function `IDASensInit` has not been previously called.
`IDA_ILL_INPUT`	`IDASolve` has been already called.

Notes: If the consistent sensitivity vectors or consistent derivative vectors are not desired, pass `NULL` for the corresponding argument.

!

The user must allocate space for `yyS0_mod` and `ypS0_mod` (if not `NULL`).

5.3 User-supplied routines for forward sensitivity analysis

In addition to the required and optional user-supplied routines described in §4.6, when using IDAS for forward sensitivity analysis, the user has the option of providing a routine that calculates the residual of the sensitivity equations (2.12).

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

The user may provide the residuals of the sensitivity equations (2.12), for all sensitivity parameters at once, through a function of type `IDASensResFn` defined by:

`IDASensResFn`

Definition
```
typedef int (*IDASensResFn)(int Ns, realtype t,
                            N_Vector yy, N_Vector yp, N_Vector resval,
                            N_Vector *yS, N_Vector *ypS,
                            N_Vector *resvalS, void *user_data,
                            N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
```

Purpose: This function computes the sensitivity residual for all sensitivity equations. It must compute the vectors $(\partial F/\partial y)s_i(t)+(\partial F/\partial \dot{y})\dot{s}_i(t)+(\partial F/\partial p_i)$ and store them in `resvalS[i]`.

Arguments:

`t`	is the current value of the independent variable.
`yy`	is the current value of the state vector, $y(t)$.
`yp`	is the current value of $\dot{y}(t)$.
`resval`	contains the current value F of the original DAE residual.
`yS`	contains the current values of the sensitivities s_i.
`ypS`	contains the current values of the sensitivity derivatives $\dot{s}_i$.
`resvalS`	contains the output sensitivity residual vectors.
`user_data`	is a pointer to user data.
`tmp1`	
`tmp2`	
`tmp3`	are `N_Vector`s of length N which can be used as temporary storage.

Return value: An `IDASensResFn` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `IDA_SRES_FAIL` is returned).

Notes: There is one situation in which recovery is not possible even if `IDASensResFn` function returns a recoverable error flag. That is when this occurs at the very first call to the `IDASensResFn`, in which case IDAS returns `IDA_FIRST_RES_FAIL`.

5.4 Integration of quadrature equations depending on forward sensitivities

IDAS provides support for integration of quadrature equations that depends not only on the state variables but also on forward sensitivities.

The following is an overview of the sequence of calls in a user's main program in this situation. Steps that are unchanged from the skeleton program presented in §5.1 are grayed out.

1. **[P] Initialize MPI**
2. **Set problem dimensions**
3. **Set vectors of initial values**
4. **Create IDAS object**
5. **Allocate internal memory**
6. **Set optional inputs**
7. **Attach linear solver module**
8. **Set linear solver optional inputs**
9. **Initialize sensitivity-independent quadrature problem**
10. **Define the sensitivity problem**
11. **Set sensitivity initial conditions**
12. **Activate sensitivity calculations**
13. **Set sensitivity analysis optional inputs**
14. **Set vector of initial values for quadrature variables**

    Typically, the quadrature variables should be initialized to 0.
15. **Initialize sensitivity-dependent quadrature integration**

    Call `IDAQuadSensInit` to specify the quadrature equation right-hand side function and to allocate internal memory related to quadrature integration. See §5.4.1 for details.
16. **Set optional inputs for sensitivity-dependent quadrature integration**

    Call `IDASetQuadSensErrCon` to indicate whether or not quadrature variables should be used in the step size control mechanism. If so, one of the `IDAQuadSens*tolerances` functions must be called to specify the integration tolerances for quadrature variables. See §5.4.4 for details.
17. **Advance solution in time**
18. **Extract sensitivity-dependent quadrature variables**

    Call `IDAGetQuadSens`, `IDAGetQuadSens1`, `IDAGetQuadSensDky` or `IDAGetQuadSensDky1` to obtain the values of the quadrature variables or their derivatives at the current time. See §5.4.3 for details.
19. **Get optional outputs**
20. **Extract sensitivity solution**
21. **Get sensitivity-dependent quadrature optional outputs**

    Call `IDAGetQuadSens*` functions to obtain optional output related to the integration of sensitivity-dependent quadratures. See §5.4.5 for details.
22. **Deallocate memory for solutions vector**
23. **Deallocate memory for sensitivity vectors**

24. **Deallocate memory for sensitivity-dependent quadrature variables**

25. **Free solver memory**

26. **[P] Finalize MPI**

Note: `IDAQuadSensInit` (step 15 above) can be called and quadrature-related optional inputs (step 16 above) can be set, anywhere between steps 10 and 17.

5.4.1 Sensitivity-dependent quadrature initialization and deallocation

The function `IDAQuadSensInit` activates integration of quadrature equations depending on sensitivities and allocates internal memory related to these calculations. If `rhsQS` is input as `NULL`, then IDAS uses an internal function that computes difference quotient approximations to the functions $\bar{q}_i = (\partial q/\partial y)s_i + (\partial q/\partial \dot{y})\dot{s}_i + \partial q/\partial p_i$, in the notation of (2.10). The form of the call to this function is as follows:

`IDAQuadSensInit`

Call	`flag = IDAQuadSensInit(ida_mem, rhsQS, yQS0);`
Description	The function `IDAQuadSensInit` provides required problem specifications, allocates internal memory, and initializes quadrature integration.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block returned by `IDACreate`. `rhsQS` (`IDAQuadSensRhsFn`) is the C function which computes f_{QS}, the right-hand side of the sensitivity-dependent quadrature equations (for full details see §5.4.6). `yQS0` (`N_Vector *`) contains the initial values of sensitivity-dependent quadratures.
Return value	The return value `flag` (of type `int`) will be one of the following: `IDA_SUCCESS` The call to `IDAQuadSensInit` was successful. `IDA_MEM_NULL` The IDAS memory was not initialized by a prior call to `IDACreate`. `IDA_MEM_FAIL` A memory allocation request failed. `IDA_NO_SENS` The sensitivities were not initialized by a prior call to `IDASensInit`. `IDA_ILL_INPUT` The parameter `yQS0` is `NULL`.
Notes	Before calling `IDAQuadSensInit`, the user must enable the sensitivites by calling `IDASensInit`. If an error occurred, `IDAQuadSensInit` also sends an error message to the error handler function.

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

- Base value: `lenrw` = `lenrw` + (`maxord+5`)N_q
- If `IDAQuadSensSVtolerances` is called: `lenrw` = `lenrw` $+N_q N_s$

and the size of the integer workspace is increased as follows:

- Base value: `leniw` = `leniw` + (`maxord+5`)N_q
- If `IDAQuadSensSVtolerances` is called: `leniw` = `leniw` $+N_q N_s$

The function `IDAQuadSensReInit`, useful during the solution of a sequence of problems of same size, reinitializes the quadrature related internal memory and must follow a call to `IDAQuadSensInit`. The number `Nq` of quadratures as well as the number `Ns` of sensitivities are assumed to be unchanged from the prior call to `IDAQuadSensInit`. The call to the `IDAQuadSensReInit` function has the form:

`IDAQuadSensReInit`

Call	`flag = IDAQuadSensReInit(ida_mem, yQS0);`
Description	The function `IDAQuadSensReInit` provides required problem specifications and reinitializes the sensitivity-dependent quadrature integration.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block. `yQS0` (`N_Vector *`) contains the initial values of sensitivity-dependent quadratures.
Return value	The return value `flag` (of type `int`) will be one of the following:

`IDA_SUCCESS`	The call to `IDAQuadSensReInit` was successful.
`IDA_MEM_NULL`	The IDAS memory was not initialized by a prior call to `IDACreate`.
`IDA_NO_SENS`	Memory space for the sensitivity calculation was not allocated by a prior call to `IDASensInit`.
`IDA_NO_QUADSENS`	Memory space for the sensitivity quadratures integration was not allocated by a prior call to `IDAQuadSensInit`.
`IDA_ILL_INPUT`	The parameter `yQS0` is `NULL`.

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

`IDAQuadSensFree`

Call	`IDAQuadSensFree(ida_mem);`
Description	The function `IDAQuadSensFree` frees the memory allocated for sensitivity quadrature integration.
Arguments	The argument is the pointer to the IDAS memory block (of type `void *`).
Return value	The function `IDAQuadSensFree` has no return value.
Notes	In general, `IDAQuadSensFree` need not be called by the user as it is called automatically by `IDAFree`.

5.4.2 IDAS solver function

Even if quadrature integration was enabled, the call to the main solver function `IDASolve` is exactly the same as in §4.5.6. However, in this case the return value `flag` can also be one of the following:

`IDA_QSRHS_FAIL`	The sensitivity quadrature right-hand side function failed in an unrecoverable manner.
`IDA_FIRST_QSRHS_ERR`	The sensitivity quadrature right-hand side function failed at the first call.
`IDA_REP_QSRHS_ERR`	Convergence test failures occurred too many times due to repeated recoverable errors in the quadrature right-hand side function. The `IDA_REP_RES_ERR` will also be returned if the quadrature right-hand side function had repeated recoverable errors during the estimation of an initial step size (assuming the sensitivity quadrature variables are included in the error tests).

5.4.3 Sensitivity-dependent quadrature extraction functions

If sensitivity-dependent quadratures have been initialized by a call to `IDAQuadSensInit`, or reinitialized by a call to `IDAQuadSensReInit`, then IDAS computes a solution, sensitivities, and quadratures depending on sensitivities at time `t`. However, `IDASolve` will still return only the solutions y and $\dot{y}$. Sensitivity-dependent quadratures can be obtained using one of the following functions:

`IDAGetQuadSens`

Call	`flag = IDAGetQuadSens(ida_mem, &tret, yQS);`
Description	The function `IDAGetQuadSens` returns the quadrature sensitivity solution vectors after a successful return from `IDASolve`.
Arguments	`ida_mem` (`void *`) pointer to the memory previously allocated by `IDAInit`.
	`tret` (`realtype`) the time reached by the solver (output).
	`yQS` (`N_Vector *`) array of `Ns` computed sensitivity-dependent quadrature vectors.
Return value	The return value `flag` of `IDAGetQuadSens` is one of:

`IDA_SUCCESS`	`IDAGetQuadSens` was successful.
`IDA_MEM_NULL`	`ida_mem` was NULL.
`IDA_NO_SENS`	Sensitivities were not activated.
`IDA_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.
`IDA_BAD_DKY`	`yQS` or one of the `yQS[i]` is `NULL`.

The function `IDAGetQuadSensDky` computes the `k`-th derivatives of the interpolating polynomials for the sensitivity-dependent quadrature variables at time `t`. This function is called by `IDAGetQuadSens` with `k = 0`, but may also be called directly by the user.

`IDAGetQuadSensDky`

Call	`flag = IDAGetQuadSensDky(ida_mem, t, k, dkyQS);`
Description	The function `IDAGetQuadSensDky` returns derivatives of the quadrature sensitivities solution vectors after a successful return from `IDASolve`.
Arguments	`ida_mem` (`void *`) pointer to the memory previously allocated by `IDAInit`.
	`t` (`realtype`) the time at which information is requested. The time `t` must fall within the interval defined by the last successful step taken by IDAS.
	`k` (`int`) order of the requested derivative.
	`dkyQS` (`N_Vector *`) array of `Ns` vectors containing the derivatives. This vector array must be allocated by the user.
Return value	The return value `flag` of `IDAGetQuadSensDky` is one of:

`IDA_SUCCESS`	`IDAGetQuadSensDky` succeeded.
`IDA_MEM_NULL`	`ida_mem` was `NULL`.
`IDA_NO_SENS`	Sensitivities were not activated.
`IDA_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.
`IDA_BAD_DKY`	`dkyQS` or one of the vectors `dkyQS[i]` is `NULL`.
`IDA_BAD_K`	`k` is not in the range $0, 1, ..., klast$.
`IDA_BAD_T`	The time `t` is not in the allowed range.

Quadrature sensitivity solution vectors can also be extracted separately for each parameter in turn through the functions `IDAGetQuadSens1` and `IDAGetQuadSensDky1`, defined as follows:

`IDAGetQuadSens1`

Call	`flag = IDAGetQuadSens1(ida_mem, &tret, is, yQS);`
Description	The function `IDAGetQuadSens1` returns the `is`-th sensitivity of quadratures after a successful return from `IDASolve`.
Arguments	`ida_mem` (`void *`) pointer to the memory previously allocated by `IDAInit`.
	`tret` (`realtype`) the time reached by the solver (output).
	`is` (`int`) specifies which sensitivity vector is to be returned ($0 \leq$`is`$< N_s$).
	`yQS` (`N_Vector`) the computed sensitivity-dependent quadrature vector.

Return value The return value `flag` of `IDAGetQuadSens1` is one of:

`IDA_SUCCESS`	`IDAGetQuadSens1` was successful.
`IDA_MEM_NULL`	`ida_mem` was `NULL`.
`IDA_NO_SENS`	Forward sensitivity analysis was not initialized.
`IDA_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.
`IDA_BAD_IS`	The index `is` is not in the allowed range.
`IDA_BAD_DKY`	`yQS` is `NULL`.

`IDAGetQuadSensDky1`

Call `flag = IDAGetQuadSensDky1(ida_mem, t, k, is, dkyQS);`

Description The function `IDAGetQuadSensDky1` returns the `k`-th derivative of the `is`-th sensitivity solution vector after a successful return from `IDASolve`.

Arguments

`ida_mem`	(`void *`) pointer to the memory previously allocated by `IDAInit`.
`t`	(`realtype`) specifies the time at which sensitivity information is requested. The time `t` must fall within the interval defined by the last successful step taken by IDAS.
`k`	(`int`) order of derivative.
`is`	(`int`) specifies the sensitivity derivative vector to be returned ($0 \leq$`is`$< N_s$).
`dkyQS`	(`N_Vector`) the vector containing the derivative. The space for `dkyQS` must be allocated by the user.

Return value The return value `flag` of `IDAGetQuadSensDky1` is one of:

`IDA_SUCCESS`	`IDAGetQuadDky1` succeeded.
`IDA_MEM_NULL`	`ida_mem` was `NULL`.
`IDA_NO_SENS`	Forward sensitivity analysis was not initialized.
`IDA_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.
`IDA_BAD_DKY`	`dkyQS` is `NULL`.
`IDA_BAD_IS`	The index `is` is not in the allowed range.
`IDA_BAD_K`	`k` is not in the range $0, 1, ..., klast$.
`IDA_BAD_T`	The time `t` is not in the allowed range.

5.4.4 Optional inputs for sensitivity-dependent quadrature integration

IDAS provides the following optional input functions to control the integration of sensitivity-dependent quadrature equations.

`IDASetQuadSensErrCon`

Call `flag = IDASetQuadSensErrCon(ida_mem, errconQS)`

Description The function `IDASetQuadSensErrCon` specifies whether or not the quadrature variables are to be used in the local error control mechanism. If they are, the user must specify the error tolerances for the quadrature variables by calling `IDAQuadSensSStolerances`, `IDAQuadSensSVtolerances`, or `IDAQuadSensEEtolerances`.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`errconQS`	(`booleantype`) specifies whether sensitivity quadrature variables are included (`TRUE`) or not (`FALSE`) in the error control mechanism.

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_NO_SENS	Sensitivities were not activated.
IDA_NO_QUADSENS	Quadratures depending on the sensitivities were not activated.

Notes By default, errconQS is set to FALSE.

It is illegal to call IDASetQuadSensErrCon before a call to IDAQuadSensInit. ⚠

If the quadrature variables are part of the step size control mechanism, one of the following functions must be called to specify the integration tolerances for quadrature variables.

`IDAQuadSensSStolerances`

Call `flag = IDAQuadSensSVtolerances(ida_mem, reltolQS, abstolQS);`

Description The function IDAQuadSensSStolerances specifies scalar relative and absolute tolerances.

Arguments

ida_mem	(void *) pointer to the IDAS memory block.
reltolQS	(realtype) is the scalar relative error tolerance.
abstolQS	(realtype*) is a pointer to an array containing the Ns scalar absolute error tolerances.

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_NO_SENS	Sensitivities were not activated.
IDA_NO_QUADSENS	Quadratures depending on the sensitivities were not activated.
IDA_ILL_INPUT	One of the input tolerances was negative.

`IDAQuadSensSVtolerances`

Call `flag = IDAQuadSensSVtolerances(ida_mem, reltolQS, abstolQS);`

Description The function IDAQuadSensSVtolerances specifies scalar relative and vector absolute tolerances.

Arguments

ida_mem	(void *) pointer to the IDAS memory block.
reltolQS	(realtype) is the scalar relative error tolerance.
abstolQS	(N_Vector*) is an array of Ns variables of type N_Vector. The N_Vector from abstolS[is] specifies the vector tolerances for is-th quadrature sensitivity.

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

IDA_SUCCESS	The optional value has been successfully set.
IDA_NO_QUAD	Quadrature integration was not initialized.
IDA_MEM_NULL	The ida_mem pointer is NULL.
IDA_NO_SENS	Sensitivities were not activated.
IDA_NO_QUADSENS	Quadratures depending on the sensitivities were not activated.
IDA_ILL_INPUT	One of the input tolerances was negative.

`IDAQuadSensEEtolerances`

Call `flag = IDAQuadSensEEtolerances(ida_mem);`

Description The function IDAQuadSensEEtolerances specifies that the tolerances for the sensitivity-dependent quadratures should be estimated from those provided for the pure quadrature variables.

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

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

`IDA_SUCCESS`	The optional value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_NO_SENS`	Sensitivities were not activated.
`IDA_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.

Notes When `IDAQuadSensEEtolerances` is used, before calling `IDASolve`, integration of pure quadratures must be initialized (see 4.7.1) and tolerances for pure quadratures must be also specified (see 4.7.4).

5.4.5 Optional outputs for sensitivity-dependent quadrature integration

IDAS provides the following functions that can be used to obtain solver performance information related to quadrature integration.

`IDAGetQuadSensNumRhsEvals`

Call `flag = IDAGetQuadSensNumRhsEvals(ida_mem, &nrhsQSevals);`

Description The function `IDAGetQuadSensNumRhsEvals` returns the number of calls made to the user's quadrature right-hand side function.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`nrhsQSevals`	(`long int`) number of calls made to the user's `rhsQS` function.

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

`IDA_SUCCESS`	The optional output value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_NO_QUADSENS`	Sensitivity-dependent quadrature integration has not been initialized.

`IDAGetQuadSensNumErrTestFails`

Call `flag = IDAGetQuadSensNumErrTestFails(ida_mem, &nQSetfails);`

Description The function `IDAGetQuadSensNumErrTestFails` returns the number of local error test failures due to quadrature variables.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`nQSetfails`	(`long int`) number of error test failures due to quadrature variables.

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

`IDA_SUCCESS`	The optional output value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_NO_QUADSENS`	Sensitivity-dependent quadrature integration has not been initialized.

`IDAGetQuadSensErrWeights`

Call `flag = IDAGetQuadSensErrWeights(ida_mem, eQSweight);`

Description The function `IDAGetQuadSensErrWeights` returns the quadrature error weights at the current time.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`eQSweight`	(`N_Vector *`) array of quadrature error weight vectors at the current time.

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

`IDA_SUCCESS`	The optional output value has been successfully set.
`IDA_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDA_NO_QUADSENS`	Sensitivity-dependent quadrature integration has not been initialized.

Notes The user must allocate memory for `eQSweight`.

If quadratures were not included in the error control mechanism (through a call to `IDASetQuadSensErrCon` with `errconQS=TRUE`), `IDAGetQuadSensErrWeights` does not set the `eQSweight` vector.

`IDAGetQuadSensStats`

Call `flag = IDAGetQuadSensStats(ida_mem, &nrhsQSevals, &nQSetfails);`

Description The function `IDAGetQuadSensStats` returns the IDAS integrator statistics as a group.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`nrhsQSevals`	(`long int`) number of calls to the user's `rhsQS` function.
`nQSetfails`	(`long int`) number of error test failures due to quadrature variables.

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

`IDA_SUCCESS`	the optional output values have been successfully set.
`IDA_MEM_NULL`	the `ida_mem` pointer is `NULL`.
`IDA_NO_QUADSENS`	Sensitivity-dependent quadrature integration has not been initialized.

5.4.6 User-supplied function for sensitivity-dependent quadrature integration

For the integration of sensitivity-dependent quadrature equations, the user must provide a function that defines the right-hand side of the sensitivity quadrature equations. For sensitivities of quadratures (2.10) with integrands q, the appropriate right-hand side functions are given by $\bar{q}_i = (\partial q/\partial y)s_i + (\partial q/\partial \dot{y})\dot{s}_i + \partial q/\partial p_i$. This user function must be of type `IDAQuadSensRhsFn`, defined as follows:

`IDAQuadSensRhsFn`

Definition
```
typedef int (*IDAQuadSensRhsFn)(int Ns, realtype t, N_Vector yy,
                                N_Vector yp, N_Vector *yyS, N_Vector *ypS,
                                N_Vector rrQ, N_Vector *rhsvalQS,
                                void *user_data, N_Vector tmp1,
                                N_Vector tmp2, N_Vector tmp3)
```

Purpose This function computes the sensitivity quadrature equation right-hand side for a given value of the independent variable t and state vector y.

Arguments

`Ns`	is the number of sensitivity vectors.
`t`	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 the dependent variable vector, $\dot{y}(t)$.
`yyS`	is an array of `Ns` variables of type `N_Vector` containing the dependent sensitivity vectors s_i.
`ypS`	is an array of `Ns` variables of type `N_Vector` containing the dependent sensitivity derivatives $\dot{s}_i$.
`rrQ`	is the current value of the quadrature right-hand side q.
`rhsvalQS`	contains the `Ns` output vectors.
`user_data`	is the `user_data` pointer passed to `IDASetUserData`.
`tmp1`	
`tmp2`	
`tmp3`	are `N_Vectors` which can be used as temporary storage.

Return value An `IDAQuadSensRhsFn` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `IDA_QRHS_FAIL` is returned).

Notes Allocation of memory for `rhsvalQS` is automatically handled within IDAS.

Both `yy` and `yp` are of type `N_Vector` and both `yyS` and `ypS` are pointers to an array containing `Ns` vectors of type `N_Vector`. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with IDAS do not perform any consistency checks with respect to their `N_Vector` arguments (see §7.1 and §7.2).

There is one situation in which recovery is not possible even if `IDAQuadSensRhsFn` function returns a recoverable error flag. That is when this occurs at the very first call to the `IDAQuadSensRhsFn`, in which case IDAS returns `IDA_FIRST_QSRHS_ERR`).

5.5 Note on using partial error control

For some problems, when sensitivities are excluded from the error control test, the behavior of IDAS may appear at first glance to be erroneous. One would expect that, in such cases, the sensitivity variables would not influence in any way the step size selection.

The short explanation of this behavior is that the step size selection implemented by the error control mechanism in IDAS is based on the magnitude of the correction calculated by the nonlinear solver. As mentioned in §5.2.1, even with partial error control selected in the call to `IDASensInit`, the sensitivity variables are included in the convergence tests of the nonlinear solver.

When using the simultaneous corrector method (§2.5), the nonlinear system that is solved at each step involves both the state and sensitivity equations. In this case, it is easy to see how the sensitivity variables may affect the convergence rate of the nonlinear solver and therefore the step size selection. The case of the staggered corrector approach is more subtle. The sensitivity variables at a given step are computed only once the solver for the nonlinear state equations has converged. However, if the nonlinear system corresponding to the sensitivity equations has convergence problems, IDAS will attempt to improve the initial guess by reducing the step size in order to provide a better prediction of the sensitivity variables. Moreover, even if there are no convergence failures in the solution of the sensitivity system, IDAS may trigger a call to the linear solver's setup routine which typically involves reevaluation of Jacobian information (Jacobian approximation in the case of IDADENSE and IDABAND, or preconditioner data in the case of the Krylov solvers). The new Jacobian information will be used by subsequent calls to the nonlinear solver for the state equations and, in this way, potentially affect the step size selection.

When using the simultaneous corrector method it is not possible to decide whether nonlinear solver convergence failures or calls to the linear solver setup routine have been triggered by convergence problems due to the state or the sensitivity equations. When using one of the staggered corrector methods, however, these situations can be identified by carefully monitoring the diagnostic information provided through optional outputs. If there are no convergence failures in the sensitivity nonlinear solver, and none of the calls to the linear solver setup routine were made by the sensitivity nonlinear solver, then the step size selection is not affected by the sensitivity variables.

Finally, the user must be warned that the effect of appending sensitivity equations to a given system of DAEs on the step size selection (through the mechanisms described above) is problem-dependent and can therefore lead to either an increase or decrease of the total number of steps that IDAS takes to complete the simulation. At first glance, one would expect that the impact of the sensitivity variables, if any, would be in the direction of increasing the step size and therefore reducing the total number of steps. The argument for this is that the presence of the sensitivity variables in the convergence test of the nonlinear solver can only lead to additional iterations (and therefore a smaller iteration error), or to additional calls to the linear solver setup routine (and therefore more up-to-date Jacobian information), both of which will lead to larger steps being taken by IDAS. However, this is true only locally. Overall, a larger integration step taken at a given time may lead to step size reductions at later times, due to either nonlinear solver convergence failures or error test failures.

Chapter 6

Using IDAS for Adjoint Sensitivity Analysis

This chapter describes the use of IDAS to compute sensitivities of derived functions using adjoint sensitivity analysis. As mentioned before, the adjoint sensitivity module of IDAS provides the infrastructure for integrating backward in time any system of DAEs that depends on the solution of the original IVP, by providing various interfaces to the main IDAS integrator, as well as several supporting user-callable functions. For this reason, in the following sections we refer to the *backward problem* and not to the *adjoint problem* when discussing details relevant to the DAEs that are integrated backward in time. The backward problem can be the adjoint problem (2.20) or (2.25), and can be augmented with some quadrature differential equations.

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

We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable functions and of the user-supplied functions that were not already described in Chapter 4.

6.1 A skeleton of the user's main program

The following is a skeleton of the user's main program as an application of IDAS. The user program is to have these steps in the order indicated, unless otherwise noted. For the sake of brevity, we defer many of the details to the later sections. As in §4.4, most steps are independent of the NVECTOR implementation used; where this is not the case, usage specifications are given for the two implementations provided with IDAS: steps marked [**P**] correspond to NVECTOR_PARALLEL, steps marked [**O**] correspond to NVECTOR_OPENMP, steps +marked [**T**] correspond to NVECTOR_PTHREADS, while steps marked [**S**] correspond to NVECTOR_SERIAL. Steps that are unchanged from the skeleton programs presented in §4.4, §5.1, and §5.4, are grayed out.

1. **Include necessary header files**

   The `idas.h` header file also defines additional types, constants, and function prototypes for the adjoint sensitivity module user-callable functions. In addition, the main program should include an NVECTOR implementation header file (`nvector_serial.h`, `nvector_openmp.h`, `nvector_pthreads.h`, or `nvector_parallel.h` for the implementations provided with IDAS) and, if Newton iteration was selected, the main header file of the desired linear solver module.

2. **[P] Initialize MPI**

Forward problem

3. **Set problem dimensions for the forward problem**

4. **Set initial conditions for the forward problem**

5. **Create IDAS object for the forward problem**

6. **Allocate internal memory for the forward problem**

7. **Specify integration tolerances for forward problem**

8. **Set optional inputs for the forward problem**

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

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

11. **Initialize quadrature problem or problems for forward problems, using `IDAQuadInit` and/or `IDAQuadSensInit`.**

12. **Initialize forward sensitivity problem**

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

    Call `IDAAdjInit()` to allocate memory for the combined forward-backward problem (see §6.2.1 for details). This call requires `Nd`, the number of steps between two consecutive checkpoints. `IDAAdjInit` also specifies the type of interpolation used (see §2.6.3).

14. **Integrate forward problem**

    Call `IDASolveF`, a wrapper for the IDAS main integration function `IDASolve`, either in `IDA_NORMAL` mode to the time `tout` or in `IDA_ONE_STEP` mode inside a loop (if intermediate solutions of the forward problem are desired (see §6.2.3)). The final value of `tret` is then the maximum allowable value for the endpoint T of the backward problem.

Backward problem(s)

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

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

16. **Set initial values for the backward problem**

    Set the endpoint time `tB0` $= T$ and the corresponding vectors `yB0` and `ypB0` at which the backward problem starts.

17. **Create the backward problem**

    Call `IDACreateB`, a wrapper for `IDACreate`, to create the IDAS memory block for the new backward problem. Unlike `IDACreate`, the function `IDACreateB` does not return a pointer to the newly created memory block (see §6.2.4). Instead, this pointer is attached to the internal adjoint memory block (created by `IDAAdjInit`) and returns an identifier called `which` that the user must later specify in any actions on the newly created backward problem.

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

    Call `IDAInitB` (or `IDAInitBS`, when the backward problem depends on the forward sensitivities). The two functions are actually wrappers for `IDAInit` and allocate internal memory, specify problem data, and initialize IDAS at `tB0` for the backward problem (see §6.2.4).

19. **Specify integration tolerances for backward problem**

Call `IDASStolerancesB(...)` or `IDASVtolerancesB(...)` to specify a scalar relative tolerance and scalar absolute tolerance, or a scalar relative tolerance and a vector of absolute tolerances, respectively. The functions are wrappers for `IDASStolerances(...)` and `IDASVtolerances(...)` but they require an extra argument `which`, the identifier of the backward problem returned by `IDACreateB`. See §6.2.5 for more information.

20. **Set optional inputs for the backward problem**

    Call `IDASet*B` functions to change from their default values any optional inputs that control the behavior of IDAS. Unlike their counterparts for the forward problem, these functions take an extra argument `which`, the identifier of the backward problem returned by `IDACreateB` (see §6.2.9).

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

    Initialize the linear solver module for the backward problem by calling the appropriate wrapper function: `IDADenseB`, `IDABandB`, `IDALapackDenseB`, `IDALapackBandB`, `IDAKLUB`, `IDASuperLUMTB`, `IDASpgmrB`, `IDASpbcgB`, or `IDASptfqmrB` (see §6.2.6). Note that it is not required to use the same linear solver module for both the forward and the backward problems; for example, the forward problem could be solved with the IDADENSE linear solver and the backward problem with IDASPGMR.

22. **Initialize quadrature calculation**

    If additional quadrature equations must be evaluated, call `IDAQuadInitB` or `IDAQuadInitBS` (if quadrature depends also on the forward sensitivities) as shown in §6.2.11.1. These functions are wrappers around `IDAQuadInit` and can be used to initialize and allocate memory for quadrature integration. Optionally, call `IDASetQuad*B` functions to change from their default values optional inputs that control the integration of quadratures during the backward phase.

23. **Integrate backward problem**

    Call `IDASolveB`, a second wrapper around the IDAS main integration function `IDASolve`, to integrate the backward problem from `tB0` (see §6.2.8). This function can be called either in `IDA_NORMAL` or `IDA_ONE_STEP` mode. Typically, `IDASolveB` will be called in `IDA_NORMAL` mode with an end time equal to the initial time t_0 of the forward problem.

24. **Extract quadrature variables**

    If applicable, call `IDAGetQuadB`, a wrapper around `IDAGetQuad`, to extract the values of the quadrature variables at the time returned by the last call to `IDASolveB`. See §6.2.11.2.

25. **Deallocate memory**

    Upon completion of the backward integration, call all necessary deallocation functions. These include appropriate destructors for the vectors `y` and `yB`, a call to `IDAFree` to free the IDAS memory block for the forward problem. If additional forward integration(s) are to be done for this problem, a call to `IDAAdjFree` (see §6.2.1) may be made to free and deallocate the memory allocated for the backward problems.

26. **Finalize MPI**

    [**P**] If MPI was initialized by the user main program, call `MPI_Finalize();`.

The above user interface to the adjoint sensitivity module in IDAS was motivated by the desire to keep it as close as possible in look and feel to the one for DAE IVP integration. Note that if steps (15)-(24) are not present, a program with the above structure will have the same functionality as one described in §4.4 for integration of DAEs, albeit with some overhead due to the checkpointing scheme.

If there are multiple backward problems associated with the same forward problem, repeat steps (15)-(24) above for each successive backward problem. In the process, each call to `IDACreateB` creates a new value of the identifier `which`.

6.2 User-callable functions for adjoint sensitivity analysis

6.2.1 Adjoint sensitivity allocation and deallocation functions

After the setup phase for the forward problem, but before the call to `IDASolveF`, memory for the combined forward-backward problem must be allocated by a call to the function `IDAAdjInit`. The form of the call to this function is

`IDAAdjInit`

Call	`flag = IDAAdjInit(ida_mem, Nd, interpType);`
Description	The function `IDAAdjInit` updates IDAS memory block by allocating the internal memory needed for backward integration. Space is allocated for the `Nd` $= N_d$ interpolation data points, and a linked list of checkpoints is initialized.
Arguments	`ida_mem` (`void *`) is the pointer to the IDAS memory block returned by a previous call to `IDACreate`. `Nd` (`long int`) is the number of integration steps between two consecutive checkpoints. `interpType` (`int`) specifies the type of interpolation used and can be `IDA_POLYNOMIAL` or `IDA_HERMITE`, indicating variable-degree polynomial and cubic Hermite interpolation, respectively (see §2.6.3).
Return value	The return value `flag` (of type `int`) is one of: `IDA_SUCCESS` `IDAAdjInit` was successful. `IDA_MEM_FAIL` A memory allocation request has failed. `IDA_MEM_NULL` `ida_mem` was NULL. `IDA_ILL_INPUT` One of the parameters was invalid: `Nd` was not positive or `interpType` is not one of the `IDA_POLYNOMIAL` or `IDA_HERMITE`.
Notes	The user must set `Nd` so that all data needed for interpolation of the forward problem solution between two checkpoints fits in memory. `IDAAdjInit` attempts to allocate space for (2Nd+3) variables of type `N_Vector`. If an error occurred, `IDAAdjInit` also sends a message to the error handler function.

`IDAAdjReInit`

Call	`flag = IDAAdjReInit(ida_mem);`
Description	The function `IDAAdjReInit` reinitializes the IDAS memory block for ASA, assuming that the number of steps between check points and the type of interpolation remain unchanged.
Arguments	`ida_mem` (`void *`) is the pointer to the IDAS memory block returned by a previous call to `IDACreate`.
Return value	The return value `flag` (of type `int`) is one of: `IDA_SUCCESS` `IDAAdjReInit` was successful. `IDA_MEM_NULL` `ida_mem` was NULL. `IDA_NO_ADJ` The function `IDAAdjInit` was not previously called.
Notes	The list of check points (and associated memory) is deleted. The list of backward problems is kept. However, new backward problems can be added to this list by calling `IDACreateB`. If a new list of backward problems is also needed, then free the adjoint memory (by calling `IDAAdjFree`) and reinitialize ASA with `IDAAdjInit`. The IDAS memory for the forward and backward problems can be reinitialized separately by calling `IDAReInit` and `IDAReInitB`, respectively.

`IDAAdjFree`

Call `IDAAdjFree(ida_mem);`

Description The function `IDAAdjFree` frees the memory related to backward integration allocated by a previous call to `IDAAdjInit`.

Arguments The only argument is the IDAS memory block pointer returned by a previous call to `IDACreate`.

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

Notes This function frees all memory allocated by `IDAAdjInit`. This includes workspace memory, the linked list of checkpoints, memory for the interpolation data, as well as the IDAS memory for the backward integration phase.

In general, `IDAAdjFree` need not be called by the user as it is invoked automatically by `IDAFree`.

6.2.2 Adjoint sensitivity optional input

At any time during the integration of the forward problem, the user can disable the checkpointing of the forward sensitivities by calling the following function:

`IDAAdjSetNoSensi`

Call `flag = IDAAdjSetNoSensi(ida_mem);`

Description The function `IDAAdjSetNoSensi` instructs `IDASolveF` not to save checkpointing data for forward sensitivities any more.

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

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

`IDA_SUCCESS`	The call to `IDACreateB` was successful.
`IDA_MEM_NULL`	The `ida_mem` was `NULL`.
`IDA_NO_ADJ`	The function `IDAAdjInit` has not been previously called.

6.2.3 Forward integration function

The function `IDASolveF` is very similar to the IDAS function `IDASolve` (see §4.5.6) in that it integrates the solution of the forward problem and returns the solution $(y, \dot{y})$. At the same time, however, `IDASolveF` stores checkpoint data every `Nd` integration steps. `IDASolveF` can be called repeatedly by the user. The call to this function has the form

`IDASolveF`

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

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

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`tout`	(`realtype`) the next time at which a computed solution is desired.
`tret`	(`realtype`) the time reached by the solver (output).
`yret`	(`N_Vector`) the computed solution vector y.
`ypret`	(`N_Vector`) the computed solution vector $\dot{y}$.
`itask`	(`int`) a flag indicating the job of the solver for the next 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 an approximate value of $y($`tout`$)$ and $\dot{y}($`tout`$)$. The `IDA_ONE_STEP` option

tells the solver to take just one internal step and return the solution at the point reached by that step.

`ncheck` (`int`) the number of (internal) checkpoints stored so far.

Return value On return, `IDASolveF` returns vectors `yret`, `ypret` and a corresponding independent variable value $t =$ `tret`, such that `yret` is the computed value of $y(t)$ and `ypret` the value of $\dot{y}(t)$. Additionally, it returns in `ncheck` the number of internal checkpoints saved; the total number of checkpoint intervals is `ncheck`+1. The return value `flag` (of type `int`) will be one of the following. For more details see §4.5.6.

`IDA_SUCCESS`	`IDASolveF` succeeded.
`IDA_TSTOP_RETURN`	`IDASolveF` succeeded by reaching the optional stopping point.
`IDA_NO_MALLOC`	The function `IDAInit` has not been previously called.
`IDA_ILL_INPUT`	One of the inputs to `IDASolveF` is illegal.
`IDA_TOO_MUCH_WORK`	The solver took `mxstep` internal steps but could not reach `tout`.
`IDA_TOO_MUCH_ACC`	The solver could not satisfy the accuracy demanded by the user for some internal step.
`IDA_ERR_FAILURE`	Error test failures occurred too many times during one internal time step or occurred with $\|h\| = h_{min}$.
`IDA_CONV_FAILURE`	Convergence test failures occurred too many times during one internal time step or occurred with $\|h\| = h_{min}$.
`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_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDA_MEM_FAIL`	A memory allocation request has failed (in an attempt to allocate space for a new checkpoint).

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

At this time, `IDASolveF` stores checkpoint information in memory only. Future versions will provide for a safeguard option of dumping checkpoint data into a temporary file as needed. The data stored at each checkpoint is basically a snapshot of the IDAS internal memory block and contains enough information to restart the integration from that time and to proceed with the same step size and method order sequence as during the forward integration.

In addition, `IDASolveF` also stores interpolation data between consecutive checkpoints so that, at the end of this first forward integration phase, interpolation information is already available from the last checkpoint forward. In particular, if no checkpoints were necessary, there is no need for the second forward integration phase.

It is illegal to change the integration tolerances between consecutive calls to `IDASolveF`, as this information is not captured in the checkpoint data.

6.2.4 Backward problem initialization functions

The functions `IDACreateB` and `IDAInitB` (or `IDAInitBS`) must be called in the order listed. They instantiate an IDAS solver object, provide problem and solution specifications, and allocate internal memory for the backward problem.

`IDACreateB`

Call `flag = IDACreateB(ida_mem, &which);`

Description The function `IDACreateB` instantiates an IDAS solver object for the backward problem.

Arguments	`ida_mem`	(`void *`) pointer to the IDAS memory block returned by `IDACreate`.
	`which`	(`int`) contains the identifier assigned by IDAS for the newly created backward problem. Any call to `IDA*B` functions requires such an identifier.

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

`IDA_SUCCESS`	The call to `IDACreateB` was successful.
`IDA_MEM_NULL`	The `ida_mem` was `NULL`.
`IDA_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDA_MEM_FAIL`	A memory allocation request has failed.

There are two initialization functions for the backward problem – one for the case when the backward problem does not depend on the forward sensitivities, and one for the case when it does. These two functions are described next.

The function `IDAInitB` initializes the backward problem when it does not depend on the forward sensitivities. It is essentially wrapper for `IDAInit` with some particularization for backward integration, as described below.

`IDAInitB`

Call `flag = IDAInitB(ida_mem, which, resB, tB0, yB0, ypB0);`

Description The function `IDAInitB` provides problem specification, allocates internal memory, and initializes the backward problem.

Arguments	`ida_mem`	(`void *`) pointer to the IDAS memory block returned by `IDACreate`.
	`which`	(`int`) represents the identifier of the backward problem.
	`resB`	(`IDAResFnB`) is the C function which computes fB, the residual of the backward DAE problem. This function has the form `resB(t, y, yp, yB, ypB, resvalB, user_dataB)` (for full details see §6.3.1).
	`tB0`	(`realtype`) specifies the endpoint T where final conditions are provided for the backward problem, normally equal to the endpoint of the forward integration.
	`yB0`	(`N_Vector`) is the initial value (at $t =$ `tB0`) of the backward solution.
	`ypB0`	(`N_Vector`) is the initial derivative value (at $t =$ `tB0`) of the backward solution.

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

`IDA_SUCCESS`	The call to `IDAInitB` was successful.
`IDA_NO_MALLOC`	The function `IDAInit` has not been previously called.
`IDA_MEM_NULL`	The `ida_mem` was `NULL`.
`IDA_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDA_BAD_TB0`	The final time `tB0` was outside the interval over which the forward problem was solved.
`IDA_ILL_INPUT`	The parameter `which` represented an invalid identifier, or one of `yB0`, `ypB0`, `resB` was `NULL`.

Notes The memory allocated by `IDAInitB` is deallocated by the function `IDAAdjFree`.

For the case when backward problem also depends on the forward sensitivities, user must call `IDAInitBS` instead of `IDAInitB`. Only the third argument of each function differs between these functions.

`IDAInitBS`

Call `flag = IDAInitBS(ida_mem, which, resBS, tB0, yB0, ypB0);`

Description The function `IDAInitBS` provides problem specification, allocates internal memory, and initializes the backward problem.

Arguments `ida_mem` (`void *`) pointer to the IDAS memory block returned by `IDACreate`.

`which`	(`int`) represents the identifier of the backward problem.
`resBS`	(`IDAResFnBS`) is the C function which computes fB, the residual or the backward DAE problem. This function has the form `resBS(t, y, yp, yS, ypS, yB, ypB, resvalB, user_dataB)` (for full details see §6.3.2).
`tB0`	(`realtype`) specifies the endpoint T where final conditions are provided for the backward problem.
`yB0`	(`N_Vector`) is the initial value (at $t =$ `tB0`) of the backward solution.
`ypB0`	(`N_Vector`) is the initial derivative value (at $t =$ `tB0`) of the backward solution.

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

`IDA_SUCCESS`	The call to `IDAInitB` was successful.
`IDA_NO_MALLOC`	The function `IDAInit` has not been previously called.
`IDA_MEM_NULL`	The `ida_mem` was `NULL`.
`IDA_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDA_BAD_TB0`	The final time `tB0` was outside the interval over which the forward problem was solved.
`IDA_ILL_INPUT`	The parameter `which` represented an invalid identifier, or one of `yB0`, `ypB0`, `resB` was `NULL`, or sensitivities were not active during the forward integration.

Notes The memory allocated by `IDAInitBS` is deallocated by the function `IDAAdjFree`.

The function `IDAReInitB` reinitializes IDAS for the solution of a series of backward problems, each identified by a value of the parameter `which`. `IDAReInitB` is essentially a wrapper for `IDAReInit`, and so all details given for `IDAReInit` in §4.5.10 apply. Also, `IDAReInitB` can be called to reinitialize a backward problem even if it has been initialized with the sensitivity-dependent version `IDAInitBS`. The call to the `IDAReInitB` function has the form

`IDAReInitB`

Call `flag = IDAReInitB(ida_mem, which, tB0, yB0, ypB0)`

Description The function `IDAReInitB` reinitializes IDAS the backward problem.

Arguments

`ida_mem`	(`void *`) pointer to IDAS memory block returned by `IDACreate`.
`which`	(`int`) represents the identifier of the backward problem.
`tB0`	(`realtype`) specifies the endpoint T where final conditions are provided for the backward problem.
`yB0`	(`N_Vector`) is the initial value (at $t =$ `tB0`) of the backward solution.
`ypB0`	(`N_Vector`) is the initial derivative value (at $t =$ `tB0`) of the backward solution.

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

`IDA_SUCCESS`	The call to `IDAReInitB` was successful.
`IDA_NO_MALLOC`	The function `IDAInit` has not been previously called.
`IDA_MEM_NULL`	The `ida_mem` memory block pointer was `NULL`.
`IDA_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDA_BAD_TB0`	The final time `tB0` is outside the interval over which the forward problem was solved.
`IDA_ILL_INPUT`	The parameter `which` represented an invalid identifier, or one of `yB0`, `ypB0` was `NULL`.

6.2.5 Tolerance specification functions for backward problem

One of the following two functions must be called to specify the integration tolerances for the backward problem. Note that this call must be made after the call to `IDAInitB` or `IDAInitBS`.

`IDASStolerancesB`

Call `flag = IDASStolerances(ida_mem, which, reltolB, abstolB);`

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

Arguments
- `ida_mem` (`void *`) pointer to the IDAS memory block returned by `IDACreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `reltolB` (`realtype`) is the scalar relative error tolerance.
- `abstolB` (`realtype`) is the scalar absolute error tolerance.

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

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

`IDASVtolerancesB`

Call `flag = IDASVtolerancesB(ida_mem, which, reltolB, abstolB);`

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

Arguments
- `ida_mem` (`void *`) pointer to the IDAS memory block returned by `IDACreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `reltol` (`realtype`) is the scalar relative error tolerance.
- `abstol` (`N_Vector`) is the vector of absolute error tolerances.

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

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

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

6.2.6 Linear solver initialization functions for backward problem

All IDAS linear solver modules available for forward problems provide additional specification functions for backward problems. The initialization functions described in §4.5.3 cannot be directly used since the optional user-defined Jacobian-related functions have different prototypes for the backward problem than for the forward problem (see §6.3).

The following wrapper functions can be used to initialize one of the linear solver modules for the backward problem. Their arguments are identical to those of the functions in §4.5.3 with the exception of the additional second argument, `which`, the identifier of the backward problem.

```
flag = IDADenseB(ida_mem, which, nB);
flag = IDABandB(ida_mem, which, nB, mupperB, mlowerB);
flag = IDALapackDenseB(ida_mem, which, nB);
flag = IDALapackBandB(ida_mem, which, nB, mupperB, mlowerB);
flag = IDAKLUB(ida_mem, which, nB, nnzB);
flag = IDASuperLUMTB(ida_mem, which, num_threads, nB, nnzB);
flag = IDASpgmrB(ida_mem, which, maxlB);
flag = IDASpbcgB(ida_mem, which, maxlB);
flag = IDASptfqmrB(ida_mem, which, maxlB);
```

Their return value `flag` (of type `int`) can have any of the return values of their counterparts. If the `ida_mem` argument was `NULL`, `flag` will be `IDADLS_MEM_NULL`, `IDASLS_MEM_NULL` or `IDASPILS_MEM_NULL`. Also, if `which` is not a valid identifier, the functions will return `IDADLS_ILL_INPUT`, `IDASLS_ILL_INPUT` or `IDASPILS_ILL_INPUT`.

6.2.7 Initial condition calculation functions for backward problem

IDAS provides support for calculation of consistent initial conditions for certain backward index-one problems of semi-implicit form through the functions `IDACalcICB` and `IDACalcICBS`. Calling them is optional. It is only necessary when the initial conditions do not satisfy the adjoint system.

The above functions provide the same functionality for backward problems as `IDACalcIC` with parameter `icopt` = `IDA_YA_YDP_INIT` provides for forward problems (see §4.5.4): compute the algebraic components of yB and differential components of $\dot{y}B$, given the differential components of yB. They require that the `IDASetIdB` was previously called to specify the differential and algebraic components.

Both functions require forward solutions at the final time `tB0`. `IDACalcICBS` also needs forward sensitivities at the final time `tB0`.

`IDACalcICB`

Call `flag = IDACalcICB(ida_mem, which, tBout1, N_Vector yfin, N_Vector ypfin);`

Description The function `IDACalcICB` corrects the initial values `yB0` and `ypB0` at time `tB0` for the backward problem.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`which`	(`int`) is the identifier of the backward problem.
`tBout1`	(`realtype`) is the first value of t at which a solution will be requested (from `IDASolveB`). This value is needed here only to determine the direction of integration and rough scale in the independent variable t.
`yfin`	(`N_Vector`) the forward solution at the final time `tB0`.
`ypfin`	(`N_Vector`) the forward solution derivative at the final time `tB0`.

Return value The return value `flag` (of type `int`) can be any that is returned by `IDACalcIC` (see §4.5.4). However `IDACalcICB` can also return one of the following:

`IDA_NO_ADJ`	`IDAAdjInit` has not been previously called.
`IDA_ILL_INPUT`	Parameter `which` represented an invalid identifier.

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

Note that `IDACalcICB` will correct the values of $yB(tB_0)$ and $\dot{y}B(tB_0)$ which were specified in the previous call to `IDAInitB` or `IDAReInitB`. To obtain the corrected values, call `IDAGetconsistentICB` (see §6.2.10.2).

In the case where the backward problem also depends on the forward sensitivities, user must call the following function to correct the initial conditions:

`IDACalcICBS`

Call `flag = IDACalcICBS(ida_mem, which, tBout1, N_Vector yfin, N_Vector ypfin, N_Vector ySfin, N_Vector ypSfin);`

Description The function `IDACalcICBS` corrects the initial values `yB0` and `ypB0` at time `tB0` for the backward problem.

Arguments
- `ida_mem` (`void *`) pointer to the IDAS memory block.
- `which` (`int`) is the identifier of the backward problem.
- `tBout1` (`realtype`) is the first value of t at which a solution will be requested (from `IDASolveB`).This value is needed here only to determine the direction of integration and rough scale in the independent variable t.
- `yfin` (`N_Vector`) the forward solution at the final time `tB0`.
- `ypfin` (`N_Vector`) the forward solution derivative at the final time `tB0`.
- `ySfin` (`N_Vector *`) a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution at the final time `tB0`.
- `ypSfin` (`N_Vector *`) a pointer to an array of `Ns` vectors containing the derivatives of the forward solution sensitivities at the final time `tB0`.

Return value The return value `flag` (of type `int`) can be any that is returned by `IDACalcIC` (see §4.5.4). However `IDACalcICBS` can also return one of the following:

- `IDA_NO_ADJ` `IDAAdjInit` has not been previously called.
- `IDA_ILL_INPUT` Parameter `which` represented an invalid identifier, sensitivities were not active during forward integration, or `IDAInitBS` (or `IDAReInitBS`) has not been previously called.

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

Note that `IDACalcICBS` will correct the values of $yB(tB_0)$ and $\dot{y}B(tB_0)$ which were specified in the previous call to `IDAInitBS` or `IDAReInitBS`. To obtain the corrected values, call `IDAGetConsistentICB` (see §6.2.10.2).

6.2.8 Backward integration function

The function `IDASolveB` performs the integration of the backward problem. It is essentially a wrapper for the IDAS main integration function `IDASolve` and, in the case in which checkpoints were needed, it evolves the solution of the backward problem through a sequence of forward-backward integration pairs between consecutive checkpoints. In each pair, the first run integrates the original IVP forward in time and stores interpolation data; the second run integrates the backward problem backward in time and performs the required interpolation to provide the solution of the IVP to the backward problem.

The function `IDASolveB` does not return the solution `yB` itself. To obtain that, call the function `IDAGetB`, which is also described below.

The call to `IDASolveB` has the form

`IDASolveB`

Call `flag = IDASolveB(ida_mem, tBout, itaskB);`

Description The function `IDASolveB` integrates the backward DAE problem.

Arguments
- `ida_mem` (`void *`) pointer to the IDAS memory returned by `IDACreate`.
- `tBout` (`realtype`) the next time at which a computed solution is desired.
- `itaskB` (`int`) a flag indicating the job of the solver for the next step. The `IDA_NORMAL` task is to have the solver take internal steps until it has reached or just passed the user-specified value `tBout`. The solver then interpolates in order to return an approximate value of yB(`tBout`). The `IDA_ONE_STEP` option tells the solver to take just one internal step in the direction of `tBout` and return.

Return value The return value `flag` (of type `int`) will be one of the following. For more details see §4.5.6.

IDA_SUCCESS	IDASolveB succeeded.
IDA_MEM_NULL	The ida_mem was NULL.
IDA_NO_ADJ	The function IDAAdjInit has not been previously called.
IDA_NO_BCK	No backward problem has been added to the list of backward problems by a call to IDACreateB
IDA_NO_FWD	The function IDASolveF has not been previously called.
IDA_ILL_INPUT	One of the inputs to IDASolveB is illegal.
IDA_BAD_ITASK	The itaskB argument has an illegal value.
IDA_TOO_MUCH_WORK	The solver took mxstep internal steps but could not reach tBout.
IDA_TOO_MUCH_ACC	The solver could not satisfy the accuracy demanded by the user for some internal step.
IDA_ERR_FAILURE	Error test failures occurred too many times during one internal time step.
IDA_CONV_FAILURE	Convergence test failures occurred too many times during one internal time step.
IDA_LSETUP_FAIL	The linear solver's setup function failed in an unrecoverable manner.
IDA_SOLVE_FAIL	The linear solver's solve function failed in an unrecoverable manner.
IDA_BCKMEM_NULL	The IDAS memory for the backward problem was not created with a call to IDACreateB.
IDA_BAD_TBOUT	The desired output time tBout is outside the interval over which the forward problem was solved.
IDA_REIFWD_FAIL	Reinitialization of the forward problem failed at the first checkpoint (corresponding to the initial time of the forward problem).
IDA_FWD_FAIL	An error occurred during the integration of the forward problem.

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

In the case of multiple checkpoints and multiple backward problems, a given call to IDASolveB in IDA_ONE_STEP mode may not advance every problem one step, depending on the relative locations of the current times reached. But repeated calls will eventually advance all problems to tBout.

To obtain the solution yB to the backward problem, call the function IDAGetB as follows:

IDAGetB

Call `flag = IDAGetB(ida_mem, which, &tret, yB, ypB);`

Description The function IDAGetB provides the solution yB of the backward DAE problem.

Arguments

ida_mem	(void *) pointer to the IDAS memory returned by IDACreate.
which	(int) the identifier of the backward problem.
tret	(realtype) the time reached by the solver (output).
yB	(N_Vector) the backward solution at time tret.
ypB	(N_Vector) the backward solution derivative at time tret.

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

IDA_SUCCESS	IDAGetB was successful.
IDA_MEM_NULL	ida_mem is NULL.
IDA_NO_ADJ	The function IDAAdjInit has not been previously called.
IDA_ILL_INPUT	The parameter which is an invalid identifier.

! Notes The user must allocate space for yB and ypB.

6.2.9 Optional input functions for the backward problem

6.2.9.1 Main solver optional input functions

The adjoint module in IDAS provides wrappers for most of the optional input functions defined in §4.5.7.1. The only difference is that the user must specify the identifier `which` of the backward problem within the list managed by IDAS.

The optional input functions defined for the backward problem are:

```
flag = IDASetUserDataB(ida_mem, which, user_dataB);
flag = IDASetMaxOrdB(ida_mem, which, maxordB);
flag = IDASetMaxNumStepsB(ida_mem, which, mxstepsB);
flag = IDASetInitStepB(ida_mem, which, hinB)
flag = IDASetMaxStepB(ida_mem, which, hmaxB);
flag = IDASetSuppressAlgB(ida_mem, which, suppressalgB);
flag = IDASetIdB(ida_mem, which, idB);
flag = IDASetConstraintsB(ida_mem, which, constraintsB);
```

Their return value `flag` (of type `int`) can have any of the return values of their counterparts, but it can also be `IDA_NO_ADJ` if `IDAAdjInit` has not been called, or `IDA_ILL_INPUT` if `which` was an invalid identifier.

6.2.9.2 Dense linear solver

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

`IDADlsSetDenseJacFnB`

Call `flag = IDADlsSetDenseJacFnB(ida_mem, which, jacB);`

Description The function `IDADlsSetDenseJacFnB` specifies the dense Jacobian approximation function to be used for the backward problem.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory returned by `IDACreate`.
`which`	(`int`) represents the identifier of the backward problem.
`jacB`	(`IDADlsDenseJacFnB`) user-defined dense Jacobian approximation function.

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

`IDADLS_SUCCESS`	`IDADlsSetDenseJacFnB` succeeded.
`IDADLS_MEM_NULL`	The `ida_mem` was `NULL`.
`IDADLS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDADLS_LMEM_NULL`	The linear solver has not been initialized with a call to `IDADenseB` or `IDALapackDenseB`.
`IDADLS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function type `IDADlsDenseJacFnB` is described in §6.3.5.

`IDADlsSetDenseJacFnBS`

Call `flag = IDADlsSetDenseJacFnBS(ida_mem, which, jacBS);`

Description The function `IDADlsSetDenseJacFnBS` specifies the dense Jacobian approximation function to be used for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory returned by `IDACreate`.
`which`	(`int`) represents the identifier of the backward problem.
`jacBS`	(`IDADlsDenseJacFnBS`) user-defined dense Jacobian approximation function.

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

`IDADLS_SUCCESS`	`IDADlsSetDenseJacFnBS` succeeded.
`IDADLS_MEM_NULL`	The `ida_mem` was `NULL`.
`IDADLS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDADLS_LMEM_NULL`	The linear solver has not been initialized with a call to `IDADenseB` or `IDALapackDenseB`.
`IDADLS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function type `IDADlsDenseJacFnBS` is described in §6.3.5.

6.2.9.3 Band linear solver

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

`IDADlsSetBandJacFnB`

Call `flag = IDADlsSetBandJacFnB(ida_mem, which, jacB);`

Description The function `IDADlsSetBandJacFnB` specifies the banded Jacobian approximation function to be used for the backward problem.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory returned by `IDACreate`.
`which`	(`int`) represents the identifier of the backward problem.
`jacB`	(`IDADlsBandJacFnB`) user-defined banded Jacobian approximation function.

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

`IDADLS_SUCCESS`	`IDADlsSetBandJacFnB` succeeded.
`IDADLS_MEM_NULL`	The `ida_mem` was `NULL`.
`IDADLS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDADLS_LMEM_NULL`	The linear solver has not been initialized with a call to `IDABandB` or `IDALapackBandB`.
`IDADLS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function type `IDADlsBandJacFnB` is described in §6.3.6.

`IDADlsSetBandJacFnBS`

Call `flag = IDADlsSetBandJacFnBS(ida_mem, which, jacBS);`

Description The function `IDADlsSetBandJacFnBS` specifies the banded Jacobian approximation function to be used for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory returned by `IDACreate`.
`which`	(`int`) represents the identifier of the backward problem.
`jacBS`	(`IDADlsBandJacFnBS`) user-defined banded Jacobian approximation function.

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

`IDADLS_SUCCESS`	`IDADlsSetBandJacFnBS` succeeded.
`IDADLS_MEM_NULL`	The `ida_mem` was `NULL`.
`IDADLS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDADLS_LMEM_NULL`	The linear solver has not been initialized with a call to `IDABandB` or `IDALapackBandB`.
`IDADLS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function type `IDADlsBandJacFnBS` is described in §6.3.6.

6.2.9.4 Sparse linear solvers

Optional inputs for the IDAKLU and IDASUPERLUMT linear solver modules can be set for the backward problem through the following functions.

The following wrapper functions can be used to to set the fill-reducing ordering and, in the case of KLU, reinitialize the sparse solver in the sparse linear solver modules for the backward problem. Their arguments are identical to those of the functions in §4.5.3 with the exception of the additional second argument, `which`, the identifier of the backward problem.

```
flag = IDAKLUReInitB(ida_mem, which, nB, nnzB, reinit_typeB);
flag = IDAKLUSetOrderingB(ida_mem, which, ordering_choiceB);
flag = IDASuperLUMTSetOrderingB(ida_mem, which, num_threads, ordering_choiceB);
```

Their return value `flag` (of type `int`) can have any of the return values of their counterparts. If the `ida_mem` argument was `NULL`, `flag` will be `IDASLS_MEM_NULL`. Also, if `which` is not a valid identifier, the functions will return `IDASLS_ILL_INPUT`.

`IDASlsSetSparseJacFnB`

Call `flag = IDASlsSetSparseJacFnB(ida_mem, which, jacB);`

Description The function `IDASlsSetSparseJacFnB` specifies the sparse Jacobian approximation function to be used for the backward problem.

Arguments
- `ida_mem` (`void *`) pointer to the IDAS memory returned by `IDACreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `jacB` (`IDASlsSparseJacFnB`) user-defined sparse Jacobian approximation function.

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

`IDASLS_SUCCESS`	`IDASlsSetSparseJacFnB` succeeded.
`IDASLS_MEM_NULL`	The `ida_mem` was `NULL`.
`IDASLS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDASLS_LMEM_NULL`	The linear solver has not been initialized with a call to `IDAKLUB` or `IDASuperLUMTB`.
`IDASLS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function type `IDASlsSparseJacFnB` is described in §6.3.7.

`IDASlsSetSparseJacFnBS`

Call `flag = IDASlsSetSparseJacFnBS(ida_mem, which, jacBS);`

Description The function `IDASlsSetSparseJacFnBS` specifies the sparse Jacobian approximation function to be used for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments
- `ida_mem` (`void *`) pointer to the IDAS memory returned by `IDACreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `jacBS` (`IDASlsSparseJacFnBS`) user-defined sparse Jacobian approximation function.

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

`IDASLS_SUCCESS`	`IDASlsSetSparseJacFnBS` succeeded.
`IDASLS_MEM_NULL`	The `ida_mem` was `NULL`.
`IDASLS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDASLS_LMEM_NULL`	The linear solver has not been initialized with a call to `IDAKLUB` or `IDASuperLUMTB`.
`IDASLS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function type `IDASlsSparseJacFnBS` is described in §6.3.7.

6.2.9.5 SPILS linear solvers

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

`IDASpilsSetPreconditionerB`

Call `flag = IDASpilsSetPreconditionerB(ida_mem, which, psetupB, psolveB);`

Description The function `IDASpilsSetPrecSolveFnB` specifies the preconditioner setup and solve functions for the backward integration.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`which`	(`int`) the identifier of the backward problem.
`psetupB`	(`IDASpilsPrecSetupFnB`) user-defined preconditioner setup function.
`psolveB`	(`IDASpilsPrecSolveFnB`) user-defined preconditioner solve function.

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

`IDASPILS_SUCCESS`	The optional value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` memory block pointer was `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.
`IDASPILS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDASPILS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function types `IDASpilsPrecSolveFnB` and `IDASpilsPrecSetupFnB` are described in §6.3.9 and §6.3.10, respectively. The `psetupB` argument may be `NULL` if no setup operation is involved in the preconditioner.

`IDASpilsSetPreconditionerBS`

Call `flag = IDASpilsSetPreconditionerBS(ida_mem, which, psetupBS, psolveBS);`

Description The function `IDASpilsSetPrecSolveFnBS` specifies the preconditioner setup and solve functions for the backward integration, in the case where the backward problem depends on the forward sensitivities.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`which`	(`int`) the identifier of the backward problem.
`psetupBS`	(`IDASpilsPrecSetupFnBS`) user-defined preconditioner setup function.
`psolveBS`	(`IDASpilsPrecSolveFnBS`) user-defined preconditioner solve function.

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

`IDASPILS_SUCCESS`	The optional value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` memory block pointer was `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.
`IDASPILS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDASPILS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function types `IDASpilsPrecSolveFnBS` and `IDASpilsPrecSetupFnBS` are described in §6.3.9 and §6.3.10, respectively. The `psetupBS` argument may be `NULL` if no setup operation is involved in the preconditioner.

`IDASpilsSetJacTimesVecFnB`

Call `flag = IDASpilsSetJacTimesVecFnB(ida_mem, which, jtvB);`

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

Arguments	`ida_mem`	(`void *`) pointer to the IDAS memory block.
	`which`	(`int`) the identifier of the backward problem.
	`jtvB`	(`IDASpilsJacTimesVecFnB`) user-defined Jacobian-vector product function.

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

`IDASPILS_SUCCESS`	The optional value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` memory block pointer was `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.
`IDASPILS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDASPILS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function type `IDASpilsJacTimesVecFnB` is described in §6.3.8.

`IDASpilsSetJacTimesVecFnBS`

Call `flag = IDASpilsSetJacTimesVecFnBS(ida_mem, which, jtvBS);`

Description The function `IDASpilsSetJacTimesVecFnBS` specifies the Jacobian-vector product function to be used, in the case where the backward problem depends on the forward sensitivities.

Arguments	`ida_mem`	(`void *`) pointer to the IDAS memory block.
	`which`	(`int`) the identifier of the backward problem.
	`jtvBS`	(`IDASpilsJacTimesVecFnBS`) user-defined Jacobian-vector product function.

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

`IDASPILS_SUCCESS`	The optional value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` memory block pointer was `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.
`IDASPILS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDASPILS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function type `IDASpilsJacTimesVecFnBS` is described in §6.3.8.

`IDASpilsSetGSTypeB`

Call `flag = IDASpilsSetGSType(ida_mem, which, gstypeB);`

Description The function `IDASpilsSetGSTypeB` specifies the type of Gram-Schmidt orthogonalization to be used with IDASPGMR. 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 IDAS memory block.
	`which`	(`int`) the identifier of the backward problem.
	`gstypeB`	(`int`) type of Gram-Schmidt orthogonalization.

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

`IDASPILS_SUCCESS`	The optional value has been successfully set.
`IDASPILS_MEM_NULL`	`ida_mem` was `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.
`IDASPILS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDASPILS_ILL_INPUT`	The parameter `which` represented an invalid identifier or the value of `gstypeB` was not valid.

Notes The default value is `MODIFIED_GS`.

This option is available only with IDASPGMR.

`IDASpilsSetMaxlB`

Call `flag = IDASpilsSetMaxlB(ida_mem, which, maxlB);`

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

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`which`	(`int`) the identifier of the backward problem.
`maxlB`	(`realtype`) maximum dimension of the Krylov subspace.

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

`IDASPILS_SUCCESS`	The optional value has been successfully set.
`IDASPILS_MEM_NULL`	`ida_mem` was `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.
`IDASPILS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDASPILS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The maximum subspace dimension is initially specified in the call to `IDASpbcgB` or `IDASptfqmrB`. The call to `IDASpilsSetMaxlB` is needed only if `maxlB` is being changed from its previous value.

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

`IDASpilsSetEpsLinB`

Call `flag = IDASpilsSetEpsLinB(ida_mem, eplifacB);`

Description The function `IDASpilsSetEpsLinB` specifies the factor by which the Krylov linear solver's convergence test constant is reduced from the Newton iteration test constant. (See §2.1).

Arguments

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

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

`IDASPILS_SUCCESS`	The optional value has been successfully set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.
`IDASPILS_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDASPILS_ILL_INPUT`	The value of `eplifacB` is negative.

Notes The default value is 0.05.

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

6.2.10 Optional output functions for the backward problem

6.2.10.1 Main solver optional output functions

The user of the adjoint module in IDAS has access to any of the optional output functions described in §4.5.9, both for the main solver and for the linear solver modules. The first argument of these `IDAGet*` and `IDA*Get*` functions is the pointer to the IDAS memory block for the backward problem. In order to call any of these functions, the user must first call the following function to obtain this pointer:

`IDAGetAdjIDABmem`

Call	`ida_memB = IDAGetAdjIDABmem(ida_mem, which);`
Description	The function `IDAGetAdjIDABmem` returns a pointer to the IDAS memory block for the backward problem.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block created by `IDACreate`.
	`which` (`int`) the identifier of the backward problem.
Return value	The return value, `ida_memB` (of type `void *`), is a pointer to the IDAS memory for the backward problem.
Notes	The user should not modify `ida_memB` in any way.
	Optional output calls should pass `ida_memB` as the first argument; thus, for example, to get the number of integration steps: `flag = IDAGetNumSteps(idas_memB,&nsteps)`.

To get values of the *forward* solution during a backward integration, use the following function. The input value of `t` would typically be equal to that at which the backward solution has just been obtained with `IDAGetB`. In any case, it must be within the last checkpoint interval used by `IDASolveB`.

`IDAGetAdjY`

Call	`flag = IDAGetAdjY(ida_mem, t, y, yp);`
Description	The function `IDAGetAdjY` returns the interpolated value of the forward solution y and its derivative during a backward integration.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block created by `IDACreate`.
	`t` (`realtype`) value of the independent variable at which y is desired (input).
	`y` (`N_Vector`) forward solution $y(t)$.
	`yp` (`N_Vector`) forward solution derivative $\dot{y}(t)$.
Return value	The return value `flag` (of type `int`) is one of:
	`IDA_SUCCESS` `IDAGetAdjY` was successful.
	`IDA_MEM_NULL` `ida_mem` was `NULL`.
	`IDA_GETY_BADT` The value of `t` was outside the current checkpoint interval.
Notes	The user must allocate space for `y` and `yp`.

6.2.10.2 Initial condition calculation optional output function

`IDAGetConsistentICB`

Call	`flag = IDAGetConsistentICB(ida_mem, which, yB0_mod, ypB0_mod);`
Description	The function `IDAGetConsistentICB` returns the corrected initial conditions for backward problem calculated by `IDACalcICB`.
Arguments	`ida_mem` (`void *`) pointer to the IDAS memory block.
	`which` is the identifier of the backward problem.
	`yB0_mod` (`N_Vector`) consistent initial vector.
	`ypB0_mod` (`N_Vector`) consistent initial derivative vector.
Return value	The return value `flag` (of type `int`) is one of
	`IDA_SUCCESS` The optional output value has been successfully set.
	`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
	`IDA_NO_ADJ` `IDAAdjInit` has not been previously called.
	`IDA_ILL_INPUT` Parameter `which` did not refer a valid backward problem identifier.

Notes	If the consistent solution vector or consistent derivative vector is not desired, pass `NULL` for the corresponding argument. ⚠ The user must allocate space for `yB0_mod` and `ypB0_mod` (if not `NULL`).

6.2.11 Backward integration of quadrature equations

Not only the backward problem but also the backward quadrature equations may or may not depend on the forward sensitivities. Accordingly, one of the `IDAQuadInitB` or `IDAQuadInitBS` should be used to allocate internal memory and to initialize backward quadratures. For any other operation (extraction, optional input/output, reinitialization, deallocation), the same function is called regardless of whether or not the quadratures are sensitivity-dependent.

6.2.11.1 Backward quadrature initialization functions

The function `IDAQuadInitB` initializes and allocates memory for the backward integration of quadrature equations that do not depende on forward sensititvities. It has the following form:

`IDAQuadInitB`

Call `flag = IDAQuadInitB(ida_mem, which, rhsQB, yQB0);`

Description The function `IDAQuadInitB` provides required problem specifications, allocates internal memory, and initializes backward quadrature integration.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`which`	(`int`) the identifier of the backward problem.
`rhsQB`	(`IDAQuadRhsFnB`) is the C function which computes fQB, the residual of the backward quadrature equations. This function has the form `rhsQB(t, y, yp, yB, ypB, rhsvalBQ, user_dataB)` (see §6.3.3).
`yQB0`	(`N_Vector`) is the value of the quadrature variables at `tB0`.

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

`IDA_SUCCESS`	The call to `IDAQuadInitB` was successful.
`IDA_MEM_NULL`	`ida_mem` was `NULL`.
`IDA_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDA_MEM_FAIL`	A memory allocation request has failed.
`IDA_ILL_INPUT`	The parameter `which` is an invalid identifier.

The function `IDAQuadInitBS` initializes and allocates memory for the backward integration of quadrature equations that depend on the forward sensitivities.

`IDAQuadInitBS`

Call `flag = IDAQuadInitBS(ida_mem, which, rhsQBS, yQBS0);`

Description The function `IDAQuadInitBS` provides required problem specifications, allocates internal memory, and initializes backward quadrature integration.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`which`	(`int`) the identifier of the backward problem.
`rhsQBS`	(`IDAQuadRhsFnBS`) is the C function which computes $fQBS$, the residual of the backward quadrature equations. This function has the form `rhsQBS(t, y, yp, yS, ypS, yB, ypB, rhsvalBQS, user_dataB)` (see §6.3.4).
`yQBS0`	(`N_Vector`) is the value of the sensitivity-dependent quadrature variables at `tB0`.

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

`IDA_SUCCESS`	The call to `IDAQuadInitBS` was successful.

IDA_MEM_NULL	ida_mem was NULL.
IDA_NO_ADJ	The function IDAAdjInit has not been previously called.
IDA_MEM_FAIL	A memory allocation request has failed.
IDA_ILL_INPUT	The parameter which is an invalid identifier.

The integration of quadrature equations during the backward phase can be re-initialized by calling

`IDAQuadReInitB`

Call `flag = IDAQuadReInitB(ida_mem, which, yQB0);`

Description The function IDAQuadReInitB re-initializes the backward quadrature integration.

Arguments

ida_mem	(void *) pointer to the IDAS memory block.
which	(int) the identifier of the backward problem.
yQB0	(N_Vector) is the value of the quadrature variables at tB0.

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

IDA_SUCCESS	The call to IDAQuadReInitB was successful.
IDA_MEM_NULL	ida_mem was NULL.
IDA_NO_ADJ	The function IDAAdjInit has not been previously called.
IDA_MEM_FAIL	A memory allocation request has failed.
IDA_NO_QUAD	Quadrature integration was not activated through a previous call to IDAQuadInitB.
IDA_ILL_INPUT	The parameter which is an invalid identifier.

Notes IDAQuadReInitB can be used after a call to either IDAQuadInitB or IDAQuadInitBS.

6.2.11.2 Backward quadrature extraction function

To extract the values of the quadrature variables at the last return time of IDASolveB, IDAS provides a wrapper for the function IDAGetQuad (see §4.7.3). The call to this function has the form

`IDAGetQuadB`

Call `flag = IDAGetQuadB(ida_mem, which, &tret, yQB);`

Description The function IDAGetQuadB returns the quadrature solution vector after a successful return from IDASolveB.

Arguments

ida_mem	(void *) pointer to the IDAS memory.
tret	(realtype) the time reached by the solver (output).
yQB	(N_Vector) the computed quadrature vector.

Return value

Notes T

he user must allocate space for yQB. The return value flag of IDAGetQuadB is one of:

IDA_SUCCESS	IDAGetQuadB was successful.
IDA_MEM_NULL	ida_mem is NULL.
IDA_NO_ADJ	The function IDAAdjInit has not been previously called.
IDA_NO_QUAD	Quadrature integration was not initialized.
IDA_BAD_DKY	yQB was NULL.
IDA_ILL_INPUT	The parameter which is an invalid identifier.

6.2.11.3 Optional input/output functions for backward quadrature integration

Optional values controlling the backward integration of quadrature equations can be changed from their default values through calls to one of the following functions which are wrappers for the corresponding optional input functions defined in §4.7.4. The user must specify the identifier `which` of the backward problem for which the optional values are specified.

```
flag = IDASetQuadErrConB(ida_mem, which, errconQ);
flag = IDAQuadSStolerancesB(ida_mem, which, reltolQ, abstolQ);
flag = IDAQuadSVtolerancesB(ida_mem, which, reltolQ, abstolQ);
```

Their return value `flag` (of type `int`) can have any of the return values of its counterparts, but it can also be `IDA_NO_ADJ` if the function `IDAAdjInit` has not been previously called or `IDA_ILL_INPUT` if the parameter `which` was an invalid identifier.

Access to optional outputs related to backward quadrature integration can be obtained by calling the corresponding `IDAGetQuad*` functions (see §4.7.5). A pointer `ida_memB` to the IDAS memory block for the backward problem, required as the first argument of these functions, can be obtained through a call to the functions `IDAGetAdjIDABmem` (see §6.2.10).

6.3 User-supplied functions for adjoint sensitivity analysis

In addition to the required DAE residual function and any optional functions for the forward problem, when using the adjoint sensitivity module in IDAS, the user must supply one function defining the backward problem DAE and, optionally, functions to supply Jacobian-related information and one or two functions that define the preconditioner (if one of the IDASPILS solvers is selected) for the backward problem. Type definitions for all these user-supplied functions are given below.

6.3.1 DAE residual for the backward problem

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

`IDAResFnB`

Definition
```
typedef int (*IDAResFnB)(realtype t, N_Vector y, N_Vector yp,
                         N_Vector yB, N_Vector ypB,
                         N_Vector resvalB, void *user_dataB);
```

Purpose: This function evaluates the residual of the backward problem DAE system. This could be (2.20) or (2.25).

Arguments

`t`	is the current value of the independent variable.
`y`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`resvalB`	is the output vector containing the residual for the backward DAE problem.
`user_dataB`	is a pointer to user data, same as passed to `IDASetUserDataB`.

Return value: An `IDAResFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if an unrecoverabl failure occurred (in which case the integration stops and `IDASolveB` returns `IDA_RESFUNC_FAIL`).

Notes: Allocation of memory for `resvalB` is handled within IDAS.

The `y`, `yp`, `yB`, `ypB`, and `resvalB` arguments are all of type `N_Vector`, but `yB`, `ypB`, and `resvalB` typically have different internal representations from `y` and `yp`. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with IDAS do not perform any consistency checks with respect to their `N_Vector` arguments (see §7.1 and §7.2).

The `user_dataB` pointer is passed to the user's `resB` function every time it is called and can be the same as the `user_data` pointer used for the forward problem.

Before calling the user's `resB` function, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the residual function which will halt the integration and `IDASolveB` will return `IDA_RESFUNC_FAIL`.

6.3.2 DAE residual for the backward problem depending on the forward sensitivities

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

`IDAResFnBS`

Definition
```
typedef int (*IDAResFnBS)(realtype t, N_Vector y, N_Vector yp,
                          N_Vector *yS, N_Vector *ypS,
                          N_Vector yB, N_Vector ypB,
                          N_Vector resvalB, void *user_dataB);
```

Purpose: This function evaluates the residual of the backward problem DAE system. This could be (2.20) or (2.25).

Arguments

`t`	is the current value of the independent variable.
`y`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yS`	a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
`ypS`	a pointer to an array of `Ns` vectors containing the derivatives of the forward sensitivities.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`resvalB`	is the output vector containing the residual for the backward DAE problem.
`user_dataB`	is a pointer to user data, same as passed to `IDASetUserDataB`.

Return value: An `IDAResFnBS` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if an unrecoverable error occurred (in which case the integration stops and `IDASolveB` returns `IDA_RESFUNC_FAIL`).

Notes: Allocation of memory for `resvalB` is handled within IDAS.

The `y`, `yp`, `yB`, `ypB`, and `resvalB` arguments are all of type `N_Vector`, but `yB`, `ypB`, and `resvalB` typically have different internal representations from `y` and `yp`. Likewise for each `yS[i]` and `ypS[i]`. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with IDAS do not perform any consistency checks with respect to their `N_Vector` arguments (see §7.1 and §7.2).

The `user_dataB` pointer is passed to the user's `resBS` function every time it is called and can be the same as the `user_data` pointer used for the forward problem.

Before calling the user's `resBS` function, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the residual function which will halt the integration and `IDASolveB` will return `IDA_RESFUNC_FAIL`.

6.3.3 Quadrature right-hand side for the backward problem

The user must provide an `fQB` function of type `IDAQuadRhsFnB` defined by

`IDAQuadRhsFnB`

Definition
```
typedef int (*IDAQuadRhsFnB)(realtype t, N_Vector y, N_Vector yp,
                             N_Vector yB, N_Vector ypB,
                             N_Vector rhsvalBQ, void *user_dataB);
```

Purpose This function computes the quadrature equation right-hand side for the backward problem.

Arguments

`t`	is the current value of the independent variable.
`y`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`rhsvalBQ`	is the output vector containing the residual for the backward quadrature equations.
`user_dataB`	is a pointer to user data, same as passed to `IDASetUserDataB`.

Return value An `IDAQuadRhsFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `IDASolveB` returns `IDA_QRHSFUNC_FAIL`).

Notes Allocation of memory for `rhsvalBQ` is handled within IDAS.

The `y`, `yp`, `yB`, `ypB`, and `rhsvalBQ` arguments are all of type `N_Vector`, but they typically all have different internal representations. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with IDAS do not perform any consistency checks with repsect to their `N_Vector` arguments (see §7.1 and §7.2).

The `user_dataB` pointer is passed to the user's `fQB` function every time it is called and can be the same as the `user_data` pointer used for the forward problem.

! Before calling the user's `fQB` function, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the quadrature right-hand side function which will halt the integration and `IDASolveB` will return `IDA_QRHSFUNC_FAIL`.

6.3.4 Sensitivity-dependent quadrature right-hand side for the backward problem

The user must provide an `fQBS` function of type `IDAQuadRhsFnBS` defined by

`IDAQuadRhsFnBS`

Definition
```
typedef int (*IDAQuadRhsFnBS)(realtype t, N_Vector y, N_Vector yp,
                              N_Vector *yS, N_Vector *ypS,
                              N_Vector yB, N_Vector ypB,
                              N_Vector rhsvalBQS, void *user_dataB);
```

Purpose This function computes the quadrature equation residual for the backward problem.

Arguments

`t`	is the current value of the independent variable.
`y`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yS`	a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
`ypS`	a pointer to an array of `Ns` vectors containing the derivatives of the forward sensitivities.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`rhsvalBQS`	is the output vector containing the residual for the backward quadrature equations.
`user_dataB`	is a pointer to user data, same as passed to `IDASetUserDataB`.

Return value An `IDAQuadRhsFnBS` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `IDASolveB` returns `IDA_QRHSFUNC_FAIL`).

Notes Allocation of memory for `rhsvalBQS` is handled within IDAS.

The `y`, `yp`, `yB`, `ypB`, and `rhsvalBQS` arguments are all of type `N_Vector`, but they typically do not all have the same internal representations. Likewise for each `yS[i]` and `ypS[i]`. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with IDAS do not perform any consistency checks with repsect to their `N_Vector` arguments (see §7.1 and §7.2).

The `user_dataB` pointer is passed to the user's `fQBS` function every time it is called and can be the same as the `user_data` pointer used for the forward problem.

Before calling the user's `fQBS` function, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the quadrature right-hand side function which will halt the integration and `IDASolveB` will return `IDA_QRHSFUNC_FAIL`.

6.3.5 Jacobian information for the backward problem (direct method with dense Jacobian)

If the direct linear solver with dense treatment of the Jacobian is selected for the backward problem (i.e. `IDADenseB` or `IDALapackDenseB` is called in step 21 of §6.1), the user may provide, through a call to `IDADlsSetDenseJacFnB` or `IDADlsSetDenseJacFnBS` (see §6.2.9), a function of one of the following two types:

`IDADlsDenseJacFnB`

Definition
```
typedef int (*IDADlsDenseJacFnB)(long int NeqB, realtype tt,
                                 realtype cjB, N_Vector yy, N_Vector yp,
                                 N_Vector yB, N_Vector ypB,
                                 N_Vector resvalB,
                                 DlsMat JacB, void *user_dataB,
                                 N_Vector tmp1B, N_Vector tmp2B,
                                 N_Vector tmp3B);
```

Purpose This function computes the dense Jacobian of the backward problem (or an approximation to it).

Arguments

`NeqB`	is the backward problem size (number of equations).
`tt`	is the current value of the independent variable.
`cjB`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`yy`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`resvalB`	is the current value of the residual for the backward problem.
`JacB`	is the output approximate dense Jacobian matrix.
`user_dataB`	is a pointer to user data — the parameter passed to `IDASetUserDataB`.
`tmp1B`	
`tmp2B`	
`tmp3B`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDADlsDenseJacFnB` as temporary storage or work space.

Return value An `IDADlsDenseJacFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct, while IDADENSE sets `last_flag` to `IDADLS_JACFUNC_RECVR`), or a negative value if it failed unrecoverably (in which case the integration is halted, `IDASolveB` returns `IDA_LSETUP_FAIL` and IDADENSE sets `last_flag` to `IDADLS_JACFUNC_UNRECVR`).

Notes A user-supplied dense Jacobian function must load the `NeqB` by `NeqB` dense matrix `JacB` with an approximation to the Jacobian matrix at the point (`tt`,`yy`,`yB`), where `yy` is the solution of the original IVP at time `tt` and `yB` is the solution of the backward problem at the same time. Only nonzero elements need to be loaded into `JacB` as this matrix is set to zero before the call to the Jacobian function. The type of `JacB` is `DlsMat`. The user is referred to §4.6.5 for details regarding accessing a `DlsMat` object.

Before calling the user's `IDADlsDenseJacFnB`, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the Jacobian function which will halt the integration (`IDASolveB` returns `IDA_LSETUP_FAIL` and IDADENSE sets `last_flag` to `IDADLS_JACFUNC_UNRECVR`).

`IDADlsDenseJacFnBS`

Definition
```
typedef int (*IDADlsDenseJacFnBS)(long int NeqB, realtype tt,
                                  realtype cjB, N_Vector yy, N_Vector yp,
                                  N_Vector *yS, N_Vector *ypS,
                                  N_Vector yB, N_Vector ypB,
                                  N_Vector resvalB,
                                  DlsMat JacB, void *user_dataB,
                                  N_Vector tmp1B, N_Vector tmp2B,
                                  N_Vector tmp3B);
```

Purpose This function computes the dense Jacobian of the backward problem (or an approximation to it), in the case where the backward problem depends on the forward sensitivities.

Arguments

`NeqB`	is the backward problem size (number of equations).
`tt`	is the current value of the independent variable.
`cjB`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`yy`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.

`yS`	a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
`ypS`	a pointer to an array of `Ns` vectors containing the derivatives of the forward solution sensitivities.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`resvalB`	is the current value of the residual for the backward problem.
`JacB`	is the output approximate dense Jacobian matrix.
`user_dataB`	is a pointer to user data — the parameter passed to `IDASetUserDataB`.
`tmp1B`	
`tmp2B`	
`tmp3B`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDADlsDenseJacFnBS` as temporary storage or work space.

Return value An `IDADlsDenseJacFnBS` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct, while IDADENSE sets `last_flag` to `IDADLS_JACFUNC_RECVR`), or a negative value if it failed unrecoverably (in which case the integration is halted, `IDASolveB` returns `IDA_LSETUP_FAIL` and IDADENSE sets `last_flag` to `IDADLS_JACFUNC_UNRECVR`).

Notes A user-supplied dense Jacobian function must load the `NeqB` by `NeqB` dense matrix `JacB` with an approximation to the Jacobian matrix at the point (`tt`,`yy`,`yS`,`yB`), where `yy` is the solution of the original IVP at time `tt`, `yS` is the array of forward sensitivities at time `tt`, and `yB` is the solution of the backward problem at the same time. Only nonzero elements need to be loaded into `JacB` as this matrix is set to zero before the call to the Jacobian function. The type of `JacB` is `DlsMat`. The user is referred to §4.6.5 for details regarding accessing a `DlsMat` object.

Before calling the user's `IDADlsDenseJacFnBS`, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the Jacobian function which will halt the integration (`IDASolveB` returns `IDA_LSETUP_FAIL` and IDADENSE sets `last_flag` to `IDADLS_JACFUNC_UNRECVR`).

6.3.6 Jacobian information for the backward problem (direct method with banded Jacobian)

If the direct linear solver with banded treatment of the Jacobian is selected for the backward problem (i.e. `IDABandB` or `IDALapackBandB` is called in step 21 of §6.1), the user may provide, through a call to `IDADlsSetBandJacFnB` or `IDADlsSetBandJacFnBS` (see §6.2.9), a function of one of the following two types:

`IDADlsBandJacFnB`

Definition
```
typedef int (*IDADlsBandJacFnB)(long int NeqB,
                                long int mupperB, long int mlowerB,
                                realtype tt, realtype cjB,
                                N_Vector yy, N_Vector yp,
                                N_Vector yB, N_Vector ypB,
                                N_Vector resvalB, DlsMat JacB,
                                void *user_dataB,
                                N_Vector tmp1B, N_Vector tmp2B,
                                N_Vector tmp3B);
```

Purpose This function computes the banded Jacobian of the backward problem (or a banded approximation to it).

Arguments	`NeqB`	is the backward problem size.
	`mlowerB`	
	`mupperB`	are the lower and upper half-bandwidth of the Jacobian.
	`tt`	is the current value of the independent variable.
	`cjB`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
	`yy`	is the current value of the forward solution vector.
	`yp`	is the current value of the forward solution derivative vector.
	`yB`	is the current value of the backward dependent variable vector.
	`ypB`	is the current value of the backward dependent derivative vector.
	`resvalB`	is the current value of the residual for the backward problem.
	`JacB`	is the output approximate band Jacobian matrix.
	`user_dataB`	is a pointer to user data — the parameter passed to `IDASetUserDataB`.
	`tmp1B`	
	`tmp2B`	
	`tmp3B`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDADlsBandJacFnB` as temporary storage or work space.

Return value An `IDADlsBandJacFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct, while IDABAND sets `last_flag` to `IDADLS_JACFUNC_RECVR`), or a negative value if it failed unrecoverably (in which case the integration is halted, `IDASolveB` returns `IDA_LSETUP_FAIL` and IDADENSE sets `last_flag` to `IDADLS_JACFUNC_UNRECVR`).

Notes A user-supplied band Jacobian function must load the band matrix `JacB` (of type `DlsMat`) with the elements of the Jacobian at the point (`tt`,`yy`,`yB`), where `yy` is the solution of the original IVP at time `tt` and `yB` is the solution of the backward problem at the same time. Only nonzero elements need to be loaded into `JacB` because `JacB` is preset to zero before the call to the Jacobian function. More details on the accessor macros provided for a `DlsMat` object and on the rest of the arguments passed to a function of type `IDADlsBandJacFnB` are given in §4.6.6.

! Before calling the user's `IDADlsBandJacFnB`, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the Jacobian function which will halt the integration (`IDASolveB` returns `IDA_LSETUP_FAIL` and IDABAND sets `last_flag` to `IDADLS_JACFUNC_UNRECVR`).

`IDADlsBandJacFnBS`

Definition
```
typedef int (*IDADlsBandJacFnBS)(long int NeqB,
                                  long int mupperB, long int mlowerB,
                                  realtype tt, realtype cjB,
                                  N_Vector yy, N_Vector yp,
                                  N_Vector *yS, N_Vector *ypS,
                                  N_Vector yB, N_Vector ypB,
                                  N_Vector resvalB, DlsMat JacB,
                                  void *user_dataB,
                                  N_Vector tmp1B, N_Vector tmp2B,
                                  N_Vector tmp3B);
```

Purpose This function computes the banded Jacobian of the backward problem (or a banded approximation to it), in the case where the backward problem depends on the forward sensitivities.

Arguments `NeqB` is the backward problem size.

`mlowerB`	
`mupperB`	are the lower and upper half-bandwidth of the Jacobian.
`tt`	is the current value of the independent variable.
`cjB`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`yy`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yS`	a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
`ypS`	a pointer to an array of `Ns` vectors containing the derivatives of the forward sensitivities.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`resvalB`	is the current value of the residual for the backward problem.
`JacB`	is the output approximate band Jacobian matrix.
`user_dataB`	is a pointer to user data — the parameter passed to `IDASetUserDataB`.
`tmp1B`	
`tmp2B`	
`tmp3B`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDADlsBandJacFnBS` as temporary storage or work space.

Return value An `IDADlsBandJacFnBS` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct, while IDABAND sets `last_flag` to `IDADLS_JACFUNC_RECVR`), or a negative value if it failed unrecoverably (in which case the integration is halted, `IDASolveB` returns `IDA_LSETUP_FAIL` and IDADENSE sets `last_flag` to `IDADLS_JACFUNC_UNRECVR`).

Notes A user-supplied band Jacobian function must load the band matrix `JacB` (of type `DlsMat`) with the elements of the Jacobian at the point (`tt`,`yy`,`yS`,`yB`), where `yy` is the solution of the original IVP at time `tt`, `yS` is the array of forward sensitivities at time `tt`, and `yB` is the solution of the backward problem at the same time. Only nonzero elements need to be loaded into `JacB` because `JacB` is preset to zero before the call to the Jacobian function. More details on the accessor macros provided for a `DlsMat` object and on the rest of the arguments passed to a function of type `IDADlsBandJacFnBS` are given in §4.6.6.

Before calling the user's `IDADlsBandJacFnBS`, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the Jacobian function which will halt the integration (`IDASolveB` returns `IDA_LSETUP_FAIL` and IDABAND sets `last_flag` to `IDADLS_JACFUNC_UNRECVR`).

6.3.7 Jacobian information for the backward problem (direct method with sparse Jacobian)

If the direct linear solver with sparse treatment of the Jacobian is selected for the backward problem (i.e. `IDAKLUB` or `IDASuperLUMTB` is called in step 21 of §6.1), the user must provide, through a call to `IDASlsSetSparseJacFnB` or `IDASlsSetSparseJacFnBS` (see §6.2.9), a function of one of the following two types:

`IDASlsSparseJacFnB`

Definition
```
typedef int (*IDASlsSparseJacFnB)(realtype tt, realtype cjB,
                                  N_Vector yy, N_Vector yp,
                                  N_Vector yB, N_Vector ypB,
                                  N_Vector rrB, SlsMat JacB,
                                  void *user_dataB, N_Vector tmp1B,
                                  N_Vector tmp2B, N_Vector tmp3B);
```

Purpose This function computes the sparse Jacobian of the backward problem (or an approximation to it).

Arguments

`tt`	is the current value of the independent variable.
`cjB`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`yy`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`rrB`	is the current value of the residual for the backward problem.
`JacB`	is the output approximate sparse Jacobian matrix.
`user_dataB`	is a pointer to user data — the parameter passed to `IDASetUserDataB`.
`tmp1B`	
`tmp2B`	
`tmp3B`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDASlsSparseJacFnB` as temporary storage or work space.

Return value An `IDASlsSparseJacFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct, while IDAKLU or IDASUPERLUMT sets `last_flag` to `IDASLS_JACFUNC_RECVR`), or a negative value if it failed unrecoverably (in which case the integration is halted, `IDASolveB` returns `IDA_LSETUP_FAIL` and IDAKLU or IDASUPERLUMT sets `last_flag` to `IDASLS_JACFUNC_UNRECVR`).

Notes A user-supplied sparse Jacobian function must load the compressed-sparse-column matrix `JacB` with an approximation to the Jacobian matrix at the point (`tt`,`yy`,`yB`), where `yy` is the solution of the original IVP at time `tt` and `yB` is the solution of the backward problem at the same time. Storage for `JacB` already exists on entry to this function, although the user should ensure that sufficient space is allocated in `JacB` to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and row index arrays as needed. The type of `JacB` is `SlsMat`, and the amount of allocated space is available within the `SlsMat` structure as `NNZ`. The `SlsMat` type is further documented in the Section §9.2. The user is referred to §4.6.7 for details regarding accessing a `SlsMat` object.

Before calling the user's `IDASlsSparseJacFnB`, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the Jacobian function which will halt the integration (`IDASolveB` returns `IDA_LSETUP_FAIL` and IDAKLU or IDASUPERLUMT sets `last_flag` to `IDASLS_JACFUNC_UNRECVR`).

`IDASlsSparseJacFnBS`

Definition
```
typedef int (*IDASlsSparseJacFnBS)(realtype tt, realtype cjB,
                                   N_Vector yy, N_Vector yp,
                                   N_Vector *yS, N_Vector *ypS,
                                   N_Vector yB, N_Vector ypB,
                                   N_Vector rrB, SlsMat JacB,
                                   void *user_dataB, N_Vector tmp1B,
                                   N_Vector tmp2B, N_Vector tmp3B);
```

Purpose This function computes the sparse Jacobian of the backward problem (or an approximation to it), in the case where the backward problem depends on the forward sensitivities.

Arguments

`tt`	is the current value of the independent variable.
`cjB`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`yy`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yS`	a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
`ypS`	a pointer to an array of `Ns` vectors containing the derivatives of the forward solution sensitivities.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`rrB`	is the current value of the residual for the backward problem.
`JacB`	is the output approximate sparse Jacobian matrix.
`user_dataB`	is a pointer to user data — the parameter passed to `IDASetUserDataB`.
`tmp1B`	
`tmp2B`	
`tmp3B`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDASlsSparseJacFnBS` as temporary storage or work space.

Return value An `IDASlsSparseJacFnBS` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct, while IDAKLU or IDASUPERLUMT sets `last_flag` to `IDASLS_JACFUNC_RECVR`), or a negative value if it failed unrecoverably (in which case the integration is halted, `IDASolveB` returns `IDA_LSETUP_FAIL` and IDAKLU or IDASUPERLUMT sets `last_flag` to `IDASLS_JACFUNC_UNRECVR`).

Notes A user-supplied sparse Jacobian function must load the compressed-sparse-column matrix `JacB` with an approximation to the Jacobian matrix at the point (`tt`,`yy`,`yS`,`yB`), where `yy` is the solution of the original IVP at time `tt`, `yS` is the array of forward sensitivities at time `tt`, and `yB` is the solution of the backward problem at the same time. Storage for `JacB` already exists on entry to this function, although the user should ensure that sufficient space is allocated in `JacB` to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and row index arrays as needed. The type of `JacB` is `SlsMat`, and the amount of allocated space is available within the `SlsMat` structure as `NNZ`. The `SlsMat` type is further documented in the Section §9.2. The user is referred to §4.6.7 for details regarding accessing a `SlsMat` object.

Before calling the user's `IDASlsSparseJacFnBS`, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the Jacobian function which will halt the integration (`IDASolveB` returns `IDA_LSETUP_FAIL` and IDAKLU or IDASUPERLUMT sets `last_flag` to `IDASLS_JACFUNC_UNRECVR`).

6.3.8 Jacobian information for the backward problem (matrix-vector product)

If one of the Krylov iterative linear solvers SPGMR, SPBCG, or SPTFQMR is selected (`IDASp*B` is called in step 21 of §6.1), the user may provide a function of one of the following two forms:

`IDASpilsJacTimesVecFnB`

Definition
```
typedef int (*IDASpilsJacTimesVecFnB)(realtype t,
                                      N_Vector yy, N_Vector yp,
                                      N_Vector yB, N_Vector ypB,
                                      N_Vector resvalB,
                                      N_Vector vB, N_Vector JvB,
                                      realtype cjB, void *user_dataB,
                                      N_Vector tmp1B, N_Vector tmp2B);
```

Purpose This function computes the action of the backward problem Jacobian `JB` on a given vector `vB`.

Arguments

`t`	is the current value of the independent variable.
`yy`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`resvalB`	is the current value of the residual for the backward problem.
`vB`	is the vector by which the Jacobian must be multiplied.
`JvB`	is the computed output vector, `JB*vB`.
`cjB`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`user_dataB`	is a pointer to user data — the same as the `user_dataB` parameter passed to `IDASetUserDataB`.
`tmp1B`	
`tmp2B`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDASpilsJacTimesVecFnB` as temporary storage or work space.

Return value The return value of a function of type `IDASpilsJtimesFnB` should be 0 if successful or nonzero if an error was encountered, in which case the integration is halted.

Notes A user-supplied Jacobian-vector product function must load the vector `JvB` with the product of the Jacobian of the backward problem at the point (`t`, `y`, `yB`) and the vector `vB`. Here, `y` is the solution of the original IVP at time `t` and `yB` is the solution of the backward problem at the same time. The rest of the arguments are equivalent to those passed to a function of type `IDASpilsJacTimesVecFn` (see §4.6.8). If the backward problem is the adjoint of $\dot{y} = f(t, y)$, then this function is to compute $-(\partial f/\partial y)^T v_B$.

`IDASpilsJacTimesVecFnBS`

Definition
```
typedef int (*IDASpilsJacTimesVecFnBS)(realtype t,
                                       N_Vector yy, N_Vector yp,
                                       N_Vector *yyS, N_Vector *ypS,
                                       N_Vector yB, N_Vector ypB,
                                       N_Vector resvalB,
                                       N_Vector vB, N_Vector JvB,
                                       realtype cjB, void *user_dataB,
                                       N_Vector tmp1B, N_Vector tmp2B);
```

Purpose This function computes the action of the backward problem Jacobian `JB` on a given vector `vB`, in the case where the backward problem depends on the forward sensitivities.

Arguments

`t`	is the current value of the independent variable.
`yy`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yyS`	a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.

ypS	a pointer to an array of Ns vectors containing the derivatives of the forward sensitivities.
yB	is the current value of the backward dependent variable vector.
ypB	is the current value of the backward dependent derivative vector.
resvalB	is the current value of the residual for the backward problem.
vB	is the vector by which the Jacobian must be multiplied.
JvB	is the computed output vector, JB*vB.
cjB	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
user_dataB	is a pointer to user data — the same as the user_dataB parameter passed to IDASetUserDataB.
tmp1B	
tmp2B	are pointers to memory allocated for variables of type N_Vector which can be used by IDASpilsJacTimesVecFnBS as temporary storage or work space.

Return value The return value of a function of type IDASpilsJtimesFnBS should be 0 if successful or nonzero if an error was encountered, in which case the integration is halted.

Notes A user-supplied Jacobian-vector product function must load the vector JvB with the product of the Jacobian of the backward problem at the point (t, y, yB) and the vector vB. Here, y is the solution of the original IVP at time t and yB is the solution of the backward problem at the same time. The rest of the arguments are equivalent to those passed to a function of type IDASpilsJacTimesVecFn (see §4.6.8).

6.3.9 Preconditioning for the backward problem (linear system solution)

If preconditioning is used during integration of the backward problem, then the user must provide a C function to solve the linear system $Pz = r$, where P is a left preconditioner matrix. This function must have one of the following two forms:

IDASpilsPrecSolveFnB

Definition
```
typedef int (*IDASpilsPrecSolveFnB)(realtype t,
                                    N_Vector yy, N_Vector yp,
                                    N_Vector yB, N_Vector ypB,
                                    N_Vector resvalB,
                                    N_Vector rvecB, N_Vector zvecB,
                                    realtype cjB, realtype deltaB,
                                    void *user_dataB, N_Vector tmpB);
```

Purpose This function solves the preconditioning system $Pz = r$ for the backward problem.

Arguments

t	is the current value of the independent variable.
yy	is the current value of the forward solution vector.
yp	is the current value of the forward solution derivative vector.
yB	is the current value of the backward dependent variable vector.
ypB	is the current value of the backward dependent derivative vector.
resvalB	is the current value of the residual for the backward problem.
rvecB	is the right-hand side vector r of the linear system to be solved.
zvecB	is the computed output vector.
cjB	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
deltaB	is an input tolerance to be used if an iterative method is employed in the solution.

`user_dataB`	is a pointer to user data — the same as the `user_dataB` parameter passed to the function `IDASetUserDataB`.
`tmpB`	is a pointer to memory allocated for a variable of type `N_Vector` which can be used for work space.

Return value The return value of a preconditioner solve function for the backward problem should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

`IDASpilsPrecSolveFnBS`

Definition
```
typedef int (*IDASpilsPrecSolveFnBS)(realtype t,
                                     N_Vector yy, N_Vector yp,
                                     N_Vector *yyS, N_Vector *ypS,
                                     N_Vector yB, N_Vector ypB,
                                     N_Vector resvalB,
                                     N_Vector rvecB, N_Vector zvecB,
                                     realtype cjB, realtype deltaB,
                                     void *user_dataB, N_Vector tmpB);
```

Purpose This function solves the preconditioning system $Pz = r$ for the backward problem, for the case in which the backward problem depends on the forward sensitivities.

Arguments

`t`	is the current value of the independent variable.
`yy`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yyS`	a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
`ypS`	a pointer to an array of `Ns` vectors containing the derivatives of the forward sensitivities.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`resvalB`	is the current value of the residual for the backward problem.
`rvecB`	is the right-hand side vector r of the linear system to be solved.
`zvecB`	is the computed output vector.
`cjB`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`deltaB`	is an input tolerance to be used if an iterative method is employed in the solution.
`user_dataB`	is a pointer to user data — the same as the `user_dataB` parameter passed to the function `IDASetUserDataB`.
`tmpB`	is a pointer to memory allocated for a variable of type `N_Vector` which can be used for work space.

Return value The return value of a preconditioner solve function for the backward problem should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

6.3.10 Preconditioning for the backward problem (Jacobian data)

If the user's preconditioner requires that any Jacobian-related data be preprocessed or evaluated, then this needs to be done in a user-supplied C function of one of the following two types:

`IDASpilsPrecSetupFnB`

Definition
```
typedef int (*IDASpilsPrecSetupFnB)(realtype t,
                                    N_Vector yy, N_Vector yp,
                                    N_Vector yB, N_Vector ypB,
                                    N_Vector resvalB,
                                    realtype cjB, void *user_dataB,
                                    N_Vector tmp1B, N_Vector tmp2B,
                                    N_Vector tmp3B);
```

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

Arguments The arguments of an `IDASpilsPrecSetupFnB` are as follows:

`t`	is the current value of the independent variable.
`yy`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution vector.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`resvalB`	is the current value of the residual for the backward problem.
`cjB`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`user_dataB`	is a pointer to user data — the same as the `user_dataB` parameter passed to the function `IDASetUserDataB`.
`tmp1B`	
`tmp2B`	
`tmp3B`	are pointers to memory allocated for vectors which can be used as temporary storage or work space.

Return value The return value of a preconditioner setup function for the backward problem should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

`IDASpilsPrecSetupFnBS`

Definition
```
typedef int (*IDASpilsPrecSetupFnBS)(realtype t,
                                     N_Vector yy, N_Vector yp,
                                     N_Vector *yyS, N_Vector *ypS,
                                     N_Vector yB, N_Vector ypB,
                                     N_Vector resvalB,
                                     realtype cjB, void *user_dataB,
                                     N_Vector tmp1B, N_Vector tmp2B,
                                     N_Vector tmp3B);
```

Purpose This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments The arguments of an `IDASpilsPrecSetupFnBS` are as follows:

`t`	is the current value of the independent variable.
`yy`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution vector.
`yyS`	a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.

`ypS`	a pointer to an array of `Ns` vectors containing the derivatives of the forward sensitivities.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`resvalB`	is the current value of the residual for the backward problem.
`cjB`	is the scalar in the system Jacobian, proportional to the inverse of the step size (α in Eq. (2.6)).
`user_dataB`	is a pointer to user data — the same as the `user_dataB` parameter passed to the function `IDASetUserDataB`.
`tmp1B`	
`tmp2B`	
`tmp3B`	are pointers to memory allocated for vectors which can be used as temporary storage or work space.

Return value The return value of a preconditioner setup function for the backward problem should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

6.4 Using the band-block-diagonal preconditioner for backward problems

As on the forward integration phase, the efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. The band-block-diagonal preconditioner module IDABBDPRE, provides interface functions through which it can be used on the backward integration phase.

The adjoint module in IDAS offers an interface to the band-block-diagonal preconditioner module IDABBDPRE described in section §4.8. This generates a preconditioner that is a block-diagonal matrix with each block being a band matrix and can be used with one of the Krylov linear solvers and with the MPI-parallel vector module NVECTOR_PARALLEL.

In order to use the IDABBDPRE module in the solution of the backward problem, the user must define one or two additional functions, described at the end of this section.

6.4.1 Usage of IDABBDPRE for the backward problem

The IDABBDPRE module is initialized by calling the following function, *after* one of the IDASPILS linear solvers has been specified, by calling the appropriate function (see §6.2.6).

`IDABBDPrecInitB`

Call
```
flag = IDABBDPrecInitB(ida_mem, which, NlocalB, mudqB, mldqB,
                       mukeepB, mlkeepB, dqrelyB, GresB, GcommB);
```

Description The function `IDABBDPrecInitB` initializes and allocates memory for the IDABBDPRE preconditioner for the backward problem.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`which`	(`int`) the identifier of the backward problem.
`NlocalB`	(`long int`) local vector dimension for the backward problem.
`mudqB`	(`long int`) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
`mldqB`	(`long int`) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
`mukeepB`	(`long int`) upper half-bandwidth of the retained banded approximate Jacobian block.

`mlkeepB`	(`long int`) lower half-bandwidth of the retained banded approximate Jacobian block.
`dqrelyB`	(`realtype`) the relative increment in components of `yB` used in the difference quotient approximations. The default is `dqrelyB`= $\sqrt{\text{unit roundoff}}$, which can be specified by passing `dqrely`= 0.0.
`GresB`	(`IDABBDLocalFnB`) the C function which computes $G_B(t, y, \dot{y}, y_B, \dot{y}_B)$, the function approximating the residual of the backward problem.
`GcommB`	(`IDABBDCommFnB`) the optional C function which performs all interprocess communication required for the computation of G_B.

Return value If successful, `IDABBDPrecInitB` creates, allocates, and stores (internally in the IDAS solver block) a pointer to the newly created IDABBDPRE memory block. The return value `flag` (of type `int`) is one of:

`IDASPILS_SUCCESS`	The call to `IDABBDPrecInitB` was successful.
`IDASPILS_MEM_FAIL`	A memory allocation request has failed.
`IDASPILS_MEM_NULL`	The `ida_mem` argument was `NULL`.
`IDASPILS_LMEM_NULL`	No linear solver has been attached.
`IDASPILS_ILL_INPUT`	An invalid parameter has been passed.

To reinitialize the IDABBDPRE preconditioner module for the backward problem, possibly with a change in `mudqB`, `mldqB`, or `dqrelyB`, call the following function:

`IDABBDPrecReInitB`

Call `flag = IDABBDPrecReInitB(ida_mem, which, mudqB, mldqB, dqrelyB);`

Description The function `IDABBDPrecReInitB` reinitializes the IDABBDPRE preconditioner for the backward problem.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block returned by `IDACreate`.
`which`	(`int`) the identifier of the backward problem.
`mudqB`	(`long int`) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
`mldqB`	(`long int`) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
`dqrelyB`	(`realtype`) the relative increment in components of `yB` used in the difference quotient approximations.

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

`IDASPILS_SUCCESS`	The call to `IDABBDPrecReInitB` was successful.
`IDASPILS_MEM_FAIL`	A memory allocation request has failed.
`IDASPILS_MEM_NULL`	The `ida_mem` argument was `NULL`.
`IDASPILS_PMEM_NULL`	The `IDABBDPrecInitB` has not been previously called.
`IDASPILS_LMEM_NULL`	No linear solver has been attached.
`IDASPILS_ILL_INPUT`	An invalid parameter has been passed.

For more details on IDABBDPRE see §4.8.

6.4.2 User-supplied functions for IDABBDPRE

To use the IDABBDPRE module, the user must supply one or two functions which the module calls to construct the preconditioner: a required function `GresB` (of type `IDABBDLocalFnB`) which approximates the residual of the backward problem and which is computed locally, and an optional function `GcommB` (of type `IDABBDCommFnB`) which performs all interprocess communication necessary to evaluate this approximate residual (see §4.8). The prototypes for these two functions are described below.

`IDABBDLocalFnB`

Definition
```
typedef int (*IDABBDLocalFnB)(long int NlocalB, realtype t,
                              N_Vector y, N_Vector yp,
                              N_Vector yB, N_Vector ypB,
                              N_Vector gB, void *user_dataB);
```

Purpose This `GresB` function loads the vector `gB`, an approximation to the residual of the backward problem, as a function of `t`, `y`, `yp`, and `yB` and `ypB`.

Arguments

`NlocalB`	is the local vector length for the backward problem.
`t`	is the value of the independent variable.
`y`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`gB`	is the output vector, $G_B(t, y, \dot{y}, y_B, \dot{y}_B)$.
`user_dataB`	is a pointer to user data — the same as the `user_dataB` parameter passed to `IDASetUserDataB`.

Return value An `IDABBDLocalFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `IDASolveB` returns `IDA_LSETUP_FAIL`).

Notes This routine must assume that all interprocess communication of data needed to calculate `gB` has already been done, and this data is accessible within `user_dataB`.

! Before calling the user's `IDABBDLocalFnB`, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the preconditioner setup function which will halt the integration (`IDASolveB` returns `IDA_LSETUP_FAIL`).

`IDABBDCommFnB`

Definition
```
typedef int (*IDABBDCommFnB)(long int NlocalB, realtype t,
                             N_Vector y, N_Vector yp,
                             N_Vector yB, N_Vector ypB,
                             void *user_dataB);
```

Purpose This `GcommB` function performs all interprocess communications necessary for the execution of the `GresB` function above, using the input vectors `y`, `yp`, `yB` and `ypB`.

Arguments

`NlocalB`	is the local vector length.
`t`	is the value of the independent variable.
`y`	is the current value of the forward solution vector.
`yp`	is the current value of the forward solution derivative vector.
`yB`	is the current value of the backward dependent variable vector.
`ypB`	is the current value of the backward dependent derivative vector.
`user_dataB`	is a pointer to user data — the same as the `user_dataB` parameter passed to `IDASetUserDataB`.

Return value An `IDABBDCommFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `IDASolveB` returns `IDA_LSETUP_FAIL`).

Notes The `GcommB` function is expected to save communicated data in space defined within the structure `user_dataB`.

Each call to the `GcommB` function is preceded by a call to the function that evaluates the residual of the backward problem with the same `t`, `y`, `yp`, `yB` and `ypB` arguments. If there is no additional communication needed, then pass `GcommB` = `NULL` to `IDABBDPrecInitB`.

Chapter 7

Description of the NVECTOR module

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type `N_Vector`) through a set of operations defined by the particular NVECTOR implementation. Users can provide their own specific implementation of the NVECTOR module, or use one of four provided within SUNDIALS – a serial implementation and three parallel implementations. The generic operations are described below. In the sections following, the implementations provided with SUNDIALS are described.

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

```
typedef struct _generic_N_Vector *N_Vector;

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

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

```
struct _generic_N_Vector_Ops {
  N_Vector    (*nvclone)(N_Vector);
  N_Vector    (*nvcloneempty)(N_Vector);
  void        (*nvdestroy)(N_Vector);
  void        (*nvspace)(N_Vector, long int *, long int *);
  realtype*   (*nvgetarraypointer)(N_Vector);
  void        (*nvsetarraypointer)(realtype *, N_Vector);
  void        (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
  void        (*nvconst)(realtype, N_Vector);
  void        (*nvprod)(N_Vector, N_Vector, N_Vector);
  void        (*nvdiv)(N_Vector, N_Vector, N_Vector);
  void        (*nvscale)(realtype, N_Vector, N_Vector);
  void        (*nvabs)(N_Vector, N_Vector);
  void        (*nvinv)(N_Vector, N_Vector);
  void        (*nvaddconst)(N_Vector, realtype, N_Vector);
  realtype    (*nvdotprod)(N_Vector, N_Vector);
  realtype    (*nvmaxnorm)(N_Vector);
  realtype    (*nvwrmsnorm)(N_Vector, N_Vector);
  realtype    (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvmin)(N_Vector);
  realtype    (*nvwl2norm)(N_Vector, N_Vector);
  realtype    (*nvl1norm)(N_Vector);
  void        (*nvcompare)(realtype, N_Vector, N_Vector);
  booleantype (*nvinvtest)(N_Vector, N_Vector);
  booleantype (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvminquotient)(N_Vector, N_Vector);
};
```

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

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

Table 7.1 contains a complete list of all vector operations defined by the generic NVECTOR module.

Finally, note that the generic NVECTOR module defines the functions `N_VCloneVectorArray` and `N_VCloneEmptyVectorArray`. Both functions create (by cloning) an array of `count` variables of type `N_Vector`, each of the same type as an existing `N_Vector`. Their prototypes are

```
N_Vector *N_VCloneVectorArray(int count, N_Vector w);
N_Vector *N_VCloneEmptyVectorArray(int count, N_Vector w);
```

and their definitions are based on the implementation-specific `N_VClone` and `N_VCloneEmpty` operations, respectively.

An array of variables of type `N_Vector` can be destroyed by calling `N_VDestroyVectorArray`, whose prototype is

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

and whose definition is based on the implementation-specific `N_VDestroy` operation.

A particular implementation of the NVECTOR module must:

- Specify the *content* field of `N_Vector`.
- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different `N_Vector` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free an `N_Vector` with the new *content* field and with *ops* pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `N_Vector` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the *content* field of the newly defined `N_Vector`.

Table 7.1: Description of the NVECTOR operations

Name	**Usage and Description**
`N_VClone`	`v = N_VClone(w);` Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not copy the vector, but rather allocates storage for the new vector.
`N_VCloneEmpty`	`v = N_VCloneEmpty(w);` Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not allocate storage for data.
`N_VDestroy`	`N_VDestroy(v);` Destroys the `N_Vector v` and frees memory allocated for its internal data.
`N_VSpace`	`N_VSpace(nvSpec, &lrw, &liw);` Returns storage requirements for one `N_Vector`. `lrw` contains the number of realtype words and `liw` contains the number of integer words. This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied NVECTOR module if that information is not of interest.
`N_VGetArrayPointer`	`vdata = N_VGetArrayPointer(v);` Returns a pointer to a `realtype` array from the `N_Vector v`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the solver-specific interfaces to the dense and banded (serial) linear solvers, the sparse linear solvers (serial and threaded), and in the interfaces to the banded (serial) and band-block-diagonal (parallel) preconditioner modules provided with SUNDIALS.
`N_VSetArrayPointer`	`N_VSetArrayPointer(vdata, v);` Overwrites the data in an `N_Vector` with a given array of `realtype`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVECTOR module for a parallel environment.
`N_VLinearSum`	`N_VLinearSum(a, x, b, y, z);` Performs the operation $z = ax + by$, where a and b are `realtype` scalars and x and y are of type `N_Vector`: $z_i = ax_i + by_i$, $i = 0, \ldots, n-1$.
`N_VConst`	`N_VConst(c, z);` Sets all components of the `N_Vector z` to `realtype c`: $z_i = c$, $i = 0, \ldots, n-1$.

continued on next page

continued from last page

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

continued on next page

continued from last page

Name	**Usage and Description**
`N_VWL2Norm`	`m = N_VWL2Norm(x, w);` Returns the weighted Euclidean ℓ_2 norm of the `N_Vector x` with `realtype` weight vector `w`: $m = \sqrt{\sum_{i=0}^{n-1}(x_i w_i)^2}$.
`N_VL1Norm`	`m = N_VL1Norm(x);` Returns the ℓ_1 norm of the `N_Vector x`: $m = \sum_{i=0}^{n-1} \|x_i\|$.
`N_VCompare`	`N_VCompare(c, x, z);` Compares the components of the `N_Vector x` to the `realtype` scalar `c` and returns an `N_Vector z` such that: $z_i = 1.0$ if $\|x_i\| \geq c$ and $z_i = 0.0$ otherwise.
`N_VInvTest`	`t = N_VInvTest(x, z);` Sets the components of the `N_Vector z` to be the inverses of the components of the `N_Vector x`, with prior testing for zero values: $z_i = 1.0/x_i$, $i = 0, \ldots, n-1$. This routine returns a boolean assigned to `TRUE` if all components of `x` are nonzero (successful inversion) and returns `FALSE` otherwise.
`N_VConstrMask`	`t = N_VConstrMask(c, x, m);` Performs the following constraint tests: $x_i > 0$ if $c_i = 2$, $x_i \geq 0$ if $c_i = 1$, $x_i \leq 0$ if $c_i = -1$, $x_i < 0$ if $c_i = -2$. There is no constraint on x_i if $c_i = 0$. This routine returns a boolean assigned to `FALSE` if any element failed the constraint test and assigned to `TRUE` if all passed. It also sets a mask vector `m`, with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.
`N_VMinQuotient`	`minq = N_VMinQuotient(num, denom);` This routine returns the minimum of the quotients obtained by termwise dividing $\texttt{num}_i$ by $\texttt{denom}_i$. A zero element in `denom` will be skipped. If no such quotients are found, then the large value `BIG_REAL` (defined in the header file `sundials_types.h`) is returned.

7.1 The NVECTOR_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of *data*.

```
struct _N_VectorContent_Serial {
  long int length;
  booleantype own_data;
  realtype *data;
};
```

The following five macros are provided to access the content of an NVECTOR_SERIAL vector. The suffix `_S` in the names denotes the serial version.

- `NV_CONTENT_S`

  This routine gives access to the contents of the serial vector `N_Vector`.

The assignment `v_cont` = `NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_S(v) ( (N_VectorContent_Serial)(v->content) )
```

- `NV_OWN_DATA_S`, `NV_DATA_S`, `NV_LENGTH_S`

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

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

  The assignment `v_len = NV_LENGTH_S(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_S(v) = len_v` sets the length of `v` to be `len_v`.

  Implementation:

  ```
  #define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
  #define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
  #define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
  ```

- `NV_Ith_S`

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

  The assignment `r = NV_Ith_S(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_S(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
  ```

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in Table 7.1. Their names are obtained from those in Table 7.1 by appending the suffix `_Serial`. The module NVECTOR_SERIAL provides the following additional user-callable routines:

- `N_VNew_Serial`

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

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

- `N_VNewEmpty_Serial`

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

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

- `N_VMake_Serial`

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

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

- `N_VCloneVectorArray_Serial`

  This function creates (by cloning) an array of `count` serial vectors.

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

- `N_VCloneEmptyVectorArray_Serial`

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

  ```
  N_Vector *N_VCloneEmptyVectorArray_Serial(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Serial`

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

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

- `N_VPrint_Serial`

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

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

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_S(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_S(v,i)` within the loop.
- `N_VNewEmpty_Serial`, `N_VMake_Serial`, and `N_VCloneEmptyVectorArray_Serial` set the field *own_data* = `FALSE`. `N_VDestroy_Serial` and `N_VDestroyVectorArray_Serial` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.
- To maximize efficiency, vector operations in the NVECTOR_SERIAL implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

7.2 The NVECTOR_PARALLEL implementation

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

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

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

- `NV_CONTENT_P`

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

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

  Implementation:

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

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

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

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

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

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

Implementation:

```
#define NV_OWN_DATA_P(v)   ( NV_CONTENT_P(v)->own_data )

#define NV_DATA_P(v)       ( NV_CONTENT_P(v)->data )

#define NV_LOCLENGTH_P(v)  ( NV_CONTENT_P(v)->local_length )

#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )
```

- `NV_COMM_P`

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

  Implementation:

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

- `NV_Ith_P`

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

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

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

  Implementation:

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

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Table 7.1 Their names are obtained from those in Table 7.1 by appending the suffix `_Parallel`. The module NVECTOR_PARALLEL provides the following additional user-callable routines:

- `N_VNew_Parallel`

  This function creates and allocates memory for a parallel vector.

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

- `N_VNewEmpty_Parallel`

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

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

- `N_VMake_Parallel`

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

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

- `N_VCloneVectorArray_Parallel`

  This function creates (by cloning) an array of `count` parallel vectors.

  ```
  N_Vector *N_VCloneVectorArray_Parallel(int count, N_Vector w);
  ```

- `N_VCloneEmptyVectorArray_Parallel`

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

  ```
  N_Vector *N_VCloneEmptyVectorArray_Parallel(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Parallel`

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

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

- `N_VPrint_Parallel`

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

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

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the local component array via `v_data = NV_DATA_P(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_P(v,i)` within the loop.
- `N_VNewEmpty_Parallel`, `N_VMake_Parallel`, and `N_VCloneEmptyVectorArray_Parallel` set the field *own_data* = `FALSE`. `N_VDestroy_Parallel` and `N_VDestroyVectorArray_Parallel` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.
- To maximize efficiency, vector operations in the NVECTOR_PARALLEL implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

7.3 The NVECTOR_OPENMP implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least $100,000$ before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The OpenMP NVECTOR implementation provided with SUNDIALS, NVECTOR_OPENMP, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using OpenMP.

```
struct _N_VectorContent_OpenMP {
  long int length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The following six macros are provided to access the content of an NVECTOR_OPENMP vector. The suffix _OMP in the names denotes the OpenMP version.

- NV_CONTENT_OMP

  This routine gives access to the contents of the OpenMP vector N_Vector.

  The assignment v_cont = NV_CONTENT_OMP(v) sets v_cont to be a pointer to the OpenMP N_Vector content structure.

  Implementation:

  ```
  #define NV_CONTENT_OMP(v) ( (N_VectorContent_OpenMP)(v->content) )
  ```

- NV_OWN_DATA_OMP, NV_DATA_OMP, NV_LENGTH_OMP, NV_NUM_THREADS_OMP

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

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

  The assignment v_len = NV_LENGTH_OMP(v) sets v_len to be the length of v. On the other hand, the call NV_LENGTH_OMP(v) = len_v sets the length of v to be len_v.

  The assignment v_num_threads = NV_NUM_THREADS_OMP(v) sets v_num_threads to be the number of threads from v. On the other hand, the call NV_NUM_THREADS_OMP(v) = num_threads_v sets the number of threads for v to be num_threads_v.

  Implementation:

  ```
  #define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )
  #define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )
  #define NV_LENGTH_OMP(v) ( NV_CONTENT_OMP(v)->length )
  #define NV_NUM_THREADS_OMP(v) ( NV_CONTENT_OMP(v)->num_threads )
  ```

- NV_Ith_OMP

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

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

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_OMP(v,i) ( NV_DATA_OMP(v)[i] )
  ```

The NVECTOR_OPENMP module defines OpenMP implementations of all vector operations listed in Table 7.1. Their names are obtained from those in Table 7.1 by appending the suffix _OpenMP. The module NVECTOR_OPENMP provides the following additional user-callable routines:

- N_VNew_OpenMP

  This function creates and allocates memory for a OpenMP N_Vector. Arguments are the vector length and number of threads.

  ```
  N_Vector N_VNew_OpenMP(long int vec_length, int num_threads);
  ```

- `N_VNewEmpty_OpenMP`

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

  ```
  N_Vector N_VNewEmpty_OpenMP(long int vec_length, int num_threads);
  ```

- `N_VMake_OpenMP`

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

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

- `N_VCloneVectorArray_OpenMP`

  This function creates (by cloning) an array of `count` OpenMP vectors.

  ```
  N_Vector *N_VCloneVectorArray_OpenMP(int count, N_Vector w);
  ```

- `N_VCloneEmptyVectorArray_OpenMP`

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

  ```
  N_Vector *N_VCloneEmptyVectorArray_OpenMP(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_OpenMP`

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

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

- `N_VPrint_OpenMP`

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

  ```
  void N_VPrint_OpenMP(N_Vector v);
  ```

Notes

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

- `N_VNewEmpty_OpenMP`, `N_VMake_OpenMP`, and `N_VCloneEmptyVectorArray_OpenMP` set the field *own_data* = `FALSE`. `N_VDestroy_OpenMP` and `N_VDestroyVectorArray_OpenMP` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.

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

7.4 The NVECTOR_PTHREADS implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least $100,000$ before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The Pthreads NVECTOR implementation provided with SUNDIALS, NVECTOR_PTHREADS, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using POSIX threads (Pthreads).

```
struct _N_VectorContent_Pthreads {
  long int length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The following six macros are provided to access the content of an NVECTOR_PTHREADS vector. The suffix `_PT` in the names denotes the Pthreads version.

- `NV_CONTENT_PT`

  This routine gives access to the contents of the Pthreads vector `N_Vector`.

  The assignment `v_cont = NV_CONTENT_PT(v)` sets `v_cont` to be a pointer to the Pthreads `N_Vector` content structure.

  Implementation:

  ```
  #define NV_CONTENT_PT(v) ( (N_VectorContent_Pthreads)(v->content) )
  ```

- `NV_OWN_DATA_PT`, `NV_DATA_PT`, `NV_LENGTH_PT`, `NV_NUM_THREADS_PT`

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

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

  The assignment `v_len = NV_LENGTH_PT(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_PT(v) = len_v` sets the length of `v` to be `len_v`.

  The assignment `v_num_threads = NV_NUM_THREADS_PT(v)` sets `v_num_threads` to be the number of threads from `v`. On the other hand, the call `NV_NUM_THREADS_PT(v) = num_threads_v` sets the number of threads for `v` to be `num_threads_v`.

  Implementation:

  ```
  #define NV_OWN_DATA_PT(v) ( NV_CONTENT_PT(v)->own_data )

  #define NV_DATA_PT(v) ( NV_CONTENT_PT(v)->data )

  #define NV_LENGTH_PT(v) ( NV_CONTENT_PT(v)->length )

  #define NV_NUM_THREADS_PT(v) ( NV_CONTENT_PT(v)->num_threads )
  ```

- `NV_Ith_PT`

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

  The assignment `r = NV_Ith_PT(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_PT(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_PT(v,i) ( NV_DATA_PT(v)[i] )
  ```

The NVECTOR_PTHREADS module defines Pthreads implementations of all vector operations listed in Table 7.1. Their names are obtained from those in Table 7.1 by appending the suffix `_Pthreads`. The module NVECTOR_PTHREADS provides the following additional user-callable routines:

- `N_VNew_Pthreads`

  This function creates and allocates memory for a Pthreads `N_Vector`. Arguments are the vector length and number of threads.

  ```
  N_Vector N_VNew_Pthreads(long int vec_length, int num_threads);
  ```

- `N_VNewEmpty_Pthreads`

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

  ```
  N_Vector N_VNewEmpty_Pthreads(long int vec_length, int num_threads);
  ```

- `N_VMake_Pthreads`

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

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

- `N_VCloneVectorArray_Pthreads`

  This function creates (by cloning) an array of `count` Pthreads vectors.

  ```
  N_Vector *N_VCloneVectorArray_Pthreads(int count, N_Vector w);
  ```

- `N_VCloneEmptyVectorArray_Pthreads`

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

  ```
  N_Vector *N_VCloneEmptyVectorArray_Pthreads(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Pthreads`

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

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

- `N_VPrint_Pthreads`

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

  ```
  void N_VPrint_Pthreads(N_Vector v);
  ```

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_PT(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_PT(v,i)` within the loop.
- `N_VNewEmpty_Pthreads`, `N_VMake_Pthreads`, and `N_VCloneEmptyVectorArray_Pthreads` set the field *own_data* = `FALSE`. `N_VDestroy_Pthreads` and `N_VDestroyVectorArray_Pthreads` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.
- To maximize efficiency, vector operations in the NVECTOR_PTHREADS implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

7.5 NVECTOR Examples

There are `NVector` examples that may be installed for each implementation: serial, parallel, OpenMP, and Pthreads. Each implementation makes use of the functions in `test_nvector.c`. These example functions show simple usage of the `NVector` family of functions. The input to the examples are the vector length, number of threads (if threaded implementation), and a print timing flag.
The following is a list of the example functions in `test_nvector.c`:

- `Test_N_VClone`: Creates clone of vector and checks validity of clone.
- `Test_N_VCloneEmpty`: Creates clone of empty vector and checks validity of clone.

- `Test_N_VCloneVectorArray`: Creates clone of vector array and checks validity of cloned array.
- `Test_N_VCloneVectorArray`: Creates clone of empty vector array and checks validity of cloned array.
- `Test_N_VGetArrayPointer`: Get array pointer.
- `Test_N_VSetArrayPointer`: Allocate new vector, set pointer to new vector array, and check values.
- `Test_N_VLinearSum` Case 1a: Test y = x + y
- `Test_N_VLinearSum` Case 1b: Test y = -x + y
- `Test_N_VLinearSum` Case 1c: Test y = ax + y
- `Test_N_VLinearSum` Case 2a: Test x = x + y
- `Test_N_VLinearSum` Case 2b: Test x = x - y
- `Test_N_VLinearSum` Case 2c: Test x = x + by
- `Test_N_VLinearSum` Case 3: Test z = x + y
- `Test_N_VLinearSum` Case 4a: Test z = x - y
- `Test_N_VLinearSum` Case 4b: Test z = -x + y
- `Test_N_VLinearSum` Case 5a: Test z = x + by
- `Test_N_VLinearSum` Case 5b: Test z = ax + y
- `Test_N_VLinearSum` Case 6a: Test z = -x + by
- `Test_N_VLinearSum` Case 6b: Test z = ax - y
- `Test_N_VLinearSum` Case 7: Test z = a(x + y)
- `Test_N_VLinearSum` Case 8: Test z = a(x - y)
- `Test_N_VLinearSum` Case 9: Test z = ax + by
- `Test_N_VConst`: Fill vector with constant and check result.
- `Test_N_VProd`: Test vector multiply: z = x * y
- `Test_N_VDiv`: Test vector division: z = x / y
- `Test_N_VScale`: Case 1: scale: x = cx
- `Test_N_VScale`: Case 2: copy: z = x
- `Test_N_VScale`: Case 3: negate: z = -x
- `Test_N_VScale`: Case 4: combination: z = cx
- `Test_N_VAbs`: Create absolute value of vector.
- `Test_N_VAddConst`: add constant vector: z = c + x
- `Test_N_VDotProd`: Calculate dot product of two vectors.
- `Test_N_VMaxNorm`: Create vector with known values, find and validate max norm.
- `Test_N_VWrmsNorm`: Create vector of known values, find and validate weighted root mean square.

- `Test_N_VWrmsNormMask`: Case 1: Create vector of known values, find and validate weighted root mean square using all elements.
- `Test_N_VWrmsNormMask`: Case 2: Create vector of known values, find and validate weighted root mean square using no elements.
- `Test_N_VMin`: Create vector, find and validate the min.
- `Test_N_VWL2Norm`: Create vector, find and validate the weighted Euclidean L2 norm.
- `Test_N_VL1Norm`: Create vector, find and validate the L1 norm.
- `Test_N_VCompare`: Compare vector with constant returning and validating comparison vector.
- `Test_N_VInvTest`: Test z[i] = 1 / x[i]
- `Test_N_VConstrMask`: Test mask of vector x with vector c.
- `Test_N_VMinQuotient`: Fill two vectors with known values. Calculate and validate minimum quotient.

7.6 NVECTOR functions used by IDAS

In Table 7.2 below, we list the vector functions used in the NVECTOR module used by the IDAS package. The table also shows, for each function, which of the code modules uses the function. The IDAS column shows function usage within the main integrator module, while the remaining five columns show function usage within each of the five IDAS linear solvers, the IDABBDPRE preconditioner module, and the IDAS adjoint sensitivity module (denoted here by IDAA). Here IDADLS stands for IDADENSE and IDABAND; IDASPILS stands for IDASPGMR, IDASPBCG, and IDASPTFQMR; and IDASLS stands for IDAKLU and IDASUPERLUMT.

There is one subtlety in the IDASPILS column hidden by the table, explained here for the case of the IDASPGMR module. The `N_VDotProd` function is called both within the interface file `ida_spgmr.c` and within the implementation files `sundials_spgmr.c` and `sundials_iterative.c` for the generic SPGMR solver upon which the IDASPGMR solver is built. Also, although `N_VDiv` and `N_VProd` are not called within the interface file `ida_spgmr.c`, they are called within the implementation file `sundials_spgmr.c`, and so are required by the IDASPGMR solver module. Analogous statements apply to the IDASPBCG and IDASPTFQMR modules, except that they do not use `sundials_iterative.c`. This issue does not arise for the direct IDAS linear solvers because the generic DENSE and BAND solvers (used in the implementation of IDADENSE and IDABAND) do not make calls to any vector functions.

Of the functions listed in Table 7.1, `N_VWL2Norm`, `N_VL1Norm`, `N_VCloneEmpty`, and `N_VInvTest` are *not* used by IDAS. Therefore a user-supplied NVECTOR module for IDAS could omit these four functions.

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

	IDAS	IDADLS	IDASPILS	IDASLS	IDABBDPRE	IDAA
N_VClone	✓		✓		✓	✓
N_VDestroy	✓		✓		✓	✓
N_VCloneVectorArray	✓					✓
N_VDestroyVectorArray	✓					✓
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 8

Providing Alternate Linear Solver Modules

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

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

A linear solver module must also provide a user-callable **specification function** (like those described in §4.5.3) which will attach the above five functions to the main IDAS memory block. The IDAS memory block is a structure defined in the header file `idas_impl.h`. A pointer to such a structure is defined as the type `IDAMem`. The five fields in an `IDAMem` structure that must point to the linear solver's functions are `ida_linit`, `ida_lsetup`, `ida_lsolve`, `ida_lperf`, and `ida_lfree`, respectively. Note that of the five interface functions, only `lsolve` is required. The `lfree` function must be provided only if the solver specification function makes any memory allocation. For any of the functions that are *not* provided, the corresponding field should be set to `NULL`. The linear solver specification function must also set the value of the field `ida_setupNonNull` in the IDAS memory block — to `TRUE` if `lsetup` is used, or `FALSE` otherwise.

Typically, the linear solver will require a block of memory specific to the solver, and a principal function of the specification function is to allocate that memory block, and initialize it. Then the field `ida_lmem` in the IDAS memory block is available to attach a pointer to that linear solver memory. This block can then be used to facilitate the exchange of data between the five interface functions.

If the linear solver involves adjustable parameters, the specification function should set the default values of those. User-callable functions may be defined that could, optionally, override the default parameter values.

We encourage the use of performance counters in connection with the various operations involved with the linear solver. Such counters would be members of the linear solver memory block, would be initialized in the `linit` function, and would be incremented by the `lsetup` and `lsolve` functions. Then, user-callable functions would be needed to obtain the values of these counters.

For consistency with the existing IDAS linear solver modules, we recommend that the return value of the specification function be 0 for a successful return, and a negative value if an error occurs. Possible error conditions include: the pointer to the main IDAS memory block is `NULL`, an input is illegal, the NVECTOR implementation is not compatible, or a memory allocation fails.

To be used during the backward integration with the IDAS module, a linear solver module must also provide an additional user-callable specification function (like those described in §6.2.6) which will attach the five functions to the IDAS memory block for each backward integration. Note that this block, of type `IDAMem`, is not directly accessible to the specification function, but rather is itself a field in the IDAS memory block. For a given backward problem identifier `which`, the corresponding memory block must be located in the linked list starting at `ida_mem->ida_adj_mem->IDAB_mem`; see for example the function `IDADenseB` for specific details. This specification function must also allocate the linear solver memory for the backward problem, and attach that, as well as a corresponding memory free function, to the above block `IDAB_mem`, of type `struct IDABMemRec`. The specification function for backward integration should return a negative value if it encounters an illegal input, if backward integration has not been initialized, or if its memory allocation failed.

These five functions, which interface between IDAS and the linear solver module, necessarily have fixed call sequences. Thus a user wishing to implement another linear solver within the IDAS package must adhere to this set of interfaces. The following is a complete description of the call list for each of these functions. Note that the call list of each function includes a pointer to the main IDAS memory block, by which the function can access various data related to the IDAS solution. The contents of this memory block are given in the file `idas_impl.h` (but not reproduced here, for the sake of space).

8.1 Initialization function

The type definition of `linit` is

`linit`

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

Purpose The purpose of `linit` is to complete initializations for the specific linear solver, such as counters and statistics. It should also set pointers to data blocks that will later be passed to functions associated with the linear solver. The `linit` function is called once only, at the start of the problem, during the first call to `IDASolve`.

Arguments `IDA_mem` is the IDAS memory pointer of type `IDAMem`.

Return value An `linit` function should return 0 if it has successfully initialized the IDAS linear solver and a negative value otherwise.

8.2 Setup function

The type definition of `lsetup` is

`lsetup`

Definition
```
int (*lsetup)(IDAMem IDA_mem, N_Vector yyp, N_Vector ypp, N_Vector resp,
              N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);
```

Purpose The job of `lsetup` is to prepare the linear solver for subsequent calls to `lsolve`, in the solution of systems $Mx = b$, where M is some approximation to the system Jacobian, $J = \partial F/\partial y + cj\ \partial F/\partial \dot{y}$. (See Eqn. (2.6), in which $\alpha = cj$). Here cj is available as `IDA_mem->ida_cj`.

The `lsetup` function may call a user-supplied function, or a function within the linear solver module, to compute Jacobian-related data that is required by the linear solver. It may also preprocess that data as needed for `lsolve`, which may involve calling a generic function (such as for LU factorization). This data may be intended either for direct use (in a direct linear solver) or for use in a preconditioner (in a preconditioned iterative linear solver).

The `lsetup` function is not called at every time step, but only as frequently as the solver determines that it is appropriate to perform the setup task. In this way, Jacobian-related data generated by `lsetup` is expected to be used over a number of time steps.

Arguments	`IDA_mem`	is the IDAS memory pointer of type `IDAMem`.
	`yyp`	is the predicted y vector for the current IDAS internal step.
	`ypp`	is the predicted $\dot{y}$ vector for the current IDAS internal step.
	`resp`	is the value of the residual function at `yyp` and `ypp`, i.e. $F(t_n, y_{pred}, \dot{y}_{pred})$.
	`vtemp1`	
	`vtemp2`	
	`vtemp3`	are temporary variables of type `N_Vector` provided for use by `lsetup`.

Return value The `lsetup` function should return 0 if successful, a positive value for a recoverable error, and a negative value for an unrecoverable error. On a recoverable error return, the solver will attempt to recover by reducing the step size.

8.3 Solve function

The type definition of `lsolve` is

`lsolve`

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

Purpose The `lsolve` function must solve the linear system, $Mx = b$, where M is some approximation to the system Jacobian, $J = \partial F/\partial y + cj\,\partial F/\partial \dot{y}$ (see Eqn. (2.6), in which $\alpha = cj$), and the right-hand side vector, b, is input. Here cj is available as `IDA_mem->ida_cj`.

`lsolve` is called once per Newton iteration, hence possibly several times per time step.

If there is an `lsetup` function, this `lsolve` function should make use of any Jacobian data that was computed and preprocessed by `lsetup`, either for direct use, or for use in a preconditioner.

Arguments	`IDA_mem`	is the IDAS memory pointer of type `IDAMem`.
	`b`	is the right-hand side vector b. The solution is to be returned in the vector `b`.
	`weight`	is a vector that contains the error weights. These are the W_i of (2.7). This weight vector is included here to enable the computation of weighted norms needed to test for the convergence of iterative methods (if any) within the linear solver.
	`ycur`	is a vector that contains the solver's current approximation to $y(t_n)$.
	`ypcur`	is a vector that contains the solver's current approximation to $\dot{y}(t_n)$.
	`rescur`	is a vector that contains the current residual, $F(t_n$, `ycur`, `ypcur`$)$.

Return value The `lsolve` function should return a positive value for a recoverable error and a negative value for an unrecoverable error. Success is indicated by a 0 return value. On a recoverable error return, the solver will attempt to recover, such as by calling the `lsetup` function with current arguments.

8.4 Performance monitoring function

The type definition of `lperf` is

`lperf`

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

Purpose The `lperf` function is to monitor the performance of the linear solver. It can be used to compute performance metrics related to the linear solver and issue warnings if these indicate poor performance of the linear solver. The `lperf` function is called with `perftask` = 0 at the start of each call to `IDASolve`, and then is called with `perftask` = 1 just before each internal time step.

Arguments `IDA_mem` is the IDAS memory pointer of type `IDAMem`.

`perftask` is a task flag. `perftask` = 0 means initialize needed counters. `perftask` = 1 means evaluate performance and issue warnings if needed. Counters that are used to compute performance metrics (e.g. counts of iterations within the `lsolve` function) should be initialized here when `perftask` = 0, and used for the calculation of metrics when `perftask` = 1.

Return value The `lperf` return value is ignored.

8.5 Memory deallocation function

The type definition of `lfree` is

`lfree`

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

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

Arguments The argument `IDA_mem` is the IDAS memory pointer of type `IDAMem`.

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

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

Chapter 9

General Use Linear Solver Components in SUNDIALS

In this chapter, we describe linear solver code components that are included in SUNDIALS, but which are of potential use as generic packages in themselves, either in conjunction with the use of SUNDIALS or separately.

These generic modules in SUNDIALS are organized in three families, the *dls* family, which includes direct linear solvers appropriate for sequential computations; the *sls* family, which includes sparse matrix solvers; and the *spils* family, which includes scaled preconditioned iterative (Krylov) linear solvers. The solvers in each family share common data structures and functions.

The *dls* family contains the following two generic linear solvers:

- The DENSE package, a linear solver for dense matrices either specified through a matrix type (defined below) or as simple arrays.
- The BAND package, a linear solver for banded matrices either specified through a matrix type (defined below) or as simple arrays.

Note that this family also includes the Blas/Lapack linear solvers (dense and band) available to the SUNDIALS solvers, but these are not discussed here.

The *sls* family contains a sparse matrix package and interfaces between it and two sparse direct solver packages:

- The KLU package, a linear solver for compressed-sparse-column matrices, [1, 14].
- The SUPERLUMT package, a threaded linear solver for compressed-sparse-column matrices, [2, 25, 15].

The *spils* family contains the following generic linear solvers:

- The SPGMR package, a solver for the scaled preconditioned GMRES method.
- The SPFGMR package, a solver for the scaled preconditioned Flexible GMRES method.
- The SPBCG package, a solver for the scaled preconditioned Bi-CGStab method.
- The SPTFQMR package, a solver for the scaled preconditioned TFQMR method.

For reasons related to installation, the names of the files involved in these packages begin with the prefix `sundials_`. But despite this, each of the *dls* and *spils* solvers is in fact generic, in that it is usable completely independently of SUNDIALS.

For the sake of space, the functions for the `dense` and `band` modules that work with a matrix type, and the functions in the SPGMR, SPFGMR, SPBCG, and SPTFQMR modules are only summarized briefly, since they are less likely to be of direct use in connection with a SUNDIALS solver. However, the functions for dense matrices treated as simple arrays and sparse matrices are fully described, because we expect that they will be useful in the implementation of preconditioners used with the combination of one of the SUNDIALS solvers and one of the *spils* linear solvers.

9.1 The DLS modules: DENSE and BAND

The files comprising the DENSE generic linear solver, and their locations in the SUNDIALS *srcdir*, are as follows:

- header files (located in *srcdir*/`include/sundials`)
  `sundials_direct.h`, `sundials_dense.h`,
  `sundials_types.h`, `sundials_math.h`, `sundials_config.h`
- source files (located in *srcdir*/`src/sundials`)
  `sundials_direct.c`, `sundials_dense.c`, `sundials_math.c`

The files comprising the BAND generic linear solver are as follows:

- header files (located in *srcdir*/`include/sundials`)
  `sundials_direct.h`, `sundials_band.h`,
  `sundials_types.h`, `sundials_math.h`, `sundials_config.h`
- source files (located in *srcdir*/`src/sundials`)
  `sundials_direct.c`, `sundials_band.c`, `sundials_math.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are relevant to the DENSE and BAND packages by themselves.

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  `#define SUNDIALS_DOUBLE_PRECISION 1`
  `#define SUNDIALS_SINGLE_PRECISION 1`
  `#define SUNDIALS_EXTENDED_PRECISION 1`
- (optional) use of generic math functions: `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the macros `SUNMIN` and `SUNMAX`, and the function `SUNRabs`.

The files listed above for either module can be extracted from the SUNDIALS *srcdir* and compiled by themselves into a separate library or into a larger user code.

9.1.1 Type DlsMat

The type `DlsMat`, defined in `sundials_direct.h` is a pointer to a structure defining a generic matrix, and is used with all linear solvers in the *dls* family:

```
typedef struct _DlsMat {
  int type;
  long int M;
  long int N;
  long int ldim;
  long int mu;
  long int ml;
  long int s_mu;
  realtype *data;
  long int ldata;
  realtype **cols;
} *DlsMat;
```

For the DENSE module, the relevant fields of this structure are as follows. Note that a dense matrix of type `DlsMat` need not be square.

type - `SUNDIALS_DENSE` (=1)

M - number of rows

N - number of columns

ldim - leading dimension (`ldim` $\geq$ `M`)

data - pointer to a contiguous block of `realtype` variables

ldata - length of the data array (= `ldim`$\cdot$`N`). The (`i`,`j`)-th element of a dense matrix `A` of type `DlsMat` (with $0 \leq$ `i` $<$ `M` and $0 \leq$ `j` $<$ `N`) is given by the expression `(A->data)[0][j*M+i]`

cols - array of pointers. `cols[j]` points to the first element of the j-th column of the matrix in the array data. The (`i`,`j`)-th element of a dense matrix `A` of type `DlsMat` (with $0 \leq$ `i` $<$ `M` and $0 \leq$ `j` $<$ `N`) is given by the expression `(A->cols)[j][i]`

For the BAND module, the relevant fields of this structure are as follows (see Figure 9.1 for a diagram of the underlying data representation in a banded matrix of type `DlsMat`). Note that only square band matrices are allowed.

type - `SUNDIALS_BAND` (=2)

M - number of rows

N - number of columns (`N` = `M`)

mu - upper half-bandwidth, $0 \leq$ `mu` $<$ min(`M`,`N`)

ml - lower half-bandwidth, $0 \leq$ `ml` $<$ min(`M`,`N`)

s_mu - storage upper bandwidth, `mu` $\leq$ `s_mu` $<$ `N`. The LU decomposition routine writes the LU factors into the storage for A. The upper triangular factor U, however, may have an upper bandwidth as big as min(`N`-1,`mu`+`ml`) because of partial pivoting. The `s_mu` field holds the upper half-bandwidth allocated for A.

ldim - leading dimension (`ldim` $\geq$ `s_mu`)

data - pointer to a contiguous block of `realtype` variables. The elements of a banded matrix of type `DlsMat` are stored columnwise (i.e. columns are stored one on top of the other in memory). Only elements within the specified half-bandwidths are stored. `data` is a pointer to `ldata` contiguous locations which hold the elements within the band of A.

ldata - length of the data array (= `ldim`$\cdot$(`s_mu`+`ml`+1)

cols - array of pointers. `cols[j]` is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from `s_mu`$-$`mu` (to access the uppermost element within the band in the j-th column) to `s_mu`+`ml` (to access the lowest element within the band in the j-th column). Indices from 0 to `s_mu`$-$`mu`$-$1 give access to extra storage elements required by the LU decomposition function. Finally, `cols[j][i-j+s_mu]` is the (i,j)-th element, $j-$`mu` $\leq i \leq j+$`ml`.

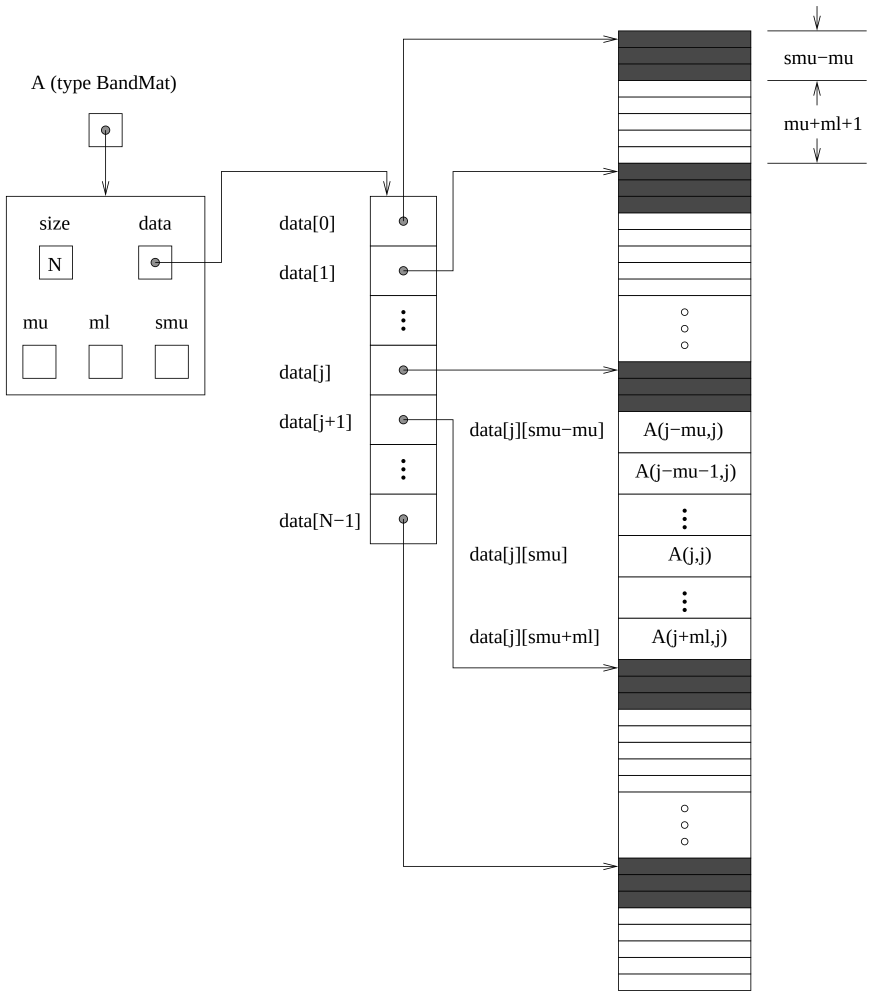


Figure 9.1: Diagram of the storage for a banded matrix of type `DlsMat`. Here `A` is an $N \times N$ band matrix of type `DlsMat` with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to $N-1$ and the (i,j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the `BandGBTRF` and `BandGBTRS` routines.

9.1.2 Accessor macros for the DLS modules

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

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

- DENSE_ELEM

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

  DENSE_ELEM references the (i,j)-th element of the $M \times N$ DlsMat A, $0 \leq$ i $< M$, $0 \leq$ j $< N$.

- DENSE_COL

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

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

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

- BAND_ELEM

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

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

- BAND_COL

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

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

- BAND_COL_ELEM

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

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

9.1.3 Functions in the DENSE module

The DENSE module defines two sets of functions with corresponding names. The first set contains functions (with names starting with a capital letter) that act on dense matrices of type DlsMat. The second set contains functions (with names starting with a lower case letter) that act on matrices represented as simple arrays.

The following functions for DlsMat dense matrices are available in the DENSE package. For full details, see the header files sundials_direct.h and sundials_dense.h.

- NewDenseMat: allocation of a DlsMat dense matrix;
- DestroyMat: free memory for a DlsMat matrix;

- `PrintMat`: print a `DlsMat` matrix to standard output.
- `NewLintArray`: allocation of an array of `long int` integers for use as pivots with `DenseGETRF` and `DenseGETRS`;
- `NewIntArray`: allocation of an array of `int` integers for use as pivots with the Lapack dense solvers;
- `NewRealArray`: allocation of an array of `realtype` for use as right-hand side with `DenseGETRS`;
- `DestroyArray`: free memory for an array;
- `SetToZero`: load a matrix with zeros;
- `AddIdentity`: increment a square matrix by the identity matrix;
- `DenseCopy`: copy one matrix to another;
- `DenseScale`: scale a matrix by a scalar;
- `DenseGETRF`: LU factorization with partial pivoting;
- `DenseGETRS`: solution of $Ax = b$ using LU factorization (for square matrices A);
- `DensePOTRF`: Cholesky factorization of a real symmetric positive matrix;
- `DensePOTRS`: solution of $Ax = b$ using the Cholesky factorization of A;
- `DenseGEQRF`: QR factorization of an $m \times n$ matrix, with $m \geq n$;
- `DenseORMQR`: compute the product $w = Qv$, with Q calculated using `DenseGEQRF`;
- `DenseMatvec`: compute the product $y = Ax$, for an M by N matrix A;

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

- `newDenseMat`

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

- `destroyMat`

  `destroyMat(a)` frees the dense matrix `a` allocated by `newDenseMat`;

- `newLintArray`

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

- `newIntArray`

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

- `newRealArray`

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

- `destroyArray`

  `destroyArray(p)` frees the array `p` allocated by `newLintArray`, `newIntArray`, or `newRealArray`;

- `denseCopy`

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

- `denseScale`

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

- `denseAddIdentity`

  `denseAddIdentity(a,n)` increments the *square* `n` by `n` dense matrix `a` by the identity matrix I_n;

- `denseGETRF`

  `denseGETRF(a,m,n,p)` factors the `m` by `n` dense matrix `a`, using Gaussian elimination with row pivoting. It overwrites the elements of `a` with its LU factors and keeps track of the pivot rows chosen in the pivot array `p`.

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

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

     `denseGETRF` returns 0 if successful. Otherwise it encountered a zero diagonal element during the factorization, indicating that the matrix `a` does not have full column rank. In this case it returns the column index (numbered from one) at which it encountered the zero.

- `denseGETRS`

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

- `densePOTRF`

  `densePOTRF(a,m)` calculates the Cholesky decomposition of the `m` by `m` dense matrix `a`, assumed to be symmetric positive definite. Only the lower triangle of `a` is accessed and overwritten with the Cholesky factor.

- `densePOTRS`

  `densePOTRS(a,m,b)` solves the `m` by `m` linear system $ax = b$. It assumes that the Cholesky factorization of `a` has been calculated in the lower triangular part of `a` by a successful call to `densePOTRF(a,m)`.

- `denseGEQRF`

  `denseGEQRF(a,m,n,beta,wrk)` calculates the QR decomposition of the `m` by `n` matrix `a` (`m` $\geq$ `n`) using Householder reflections. On exit, the elements on and above the diagonal of `a` contain the `n` by `n` upper triangular matrix `R`; the elements below the diagonal, with the array `beta`, represent the orthogonal matrix `Q` as a product of elementary reflectors. The real array `wrk`, of length `m`, must be provided as temporary workspace.

- `denseORMQR`

  `denseORMQR(a,m,n,beta,v,w,wrk)` calculates the product $w = Qv$ for a given vector `v` of length `n`, where the orthogonal matrix Q is encoded in the `m` by `n` matrix `a` and the vector `beta` of length `n`, after a successful call to `denseGEQRF(a,m,n,beta,wrk)`. The real array `wrk`, of length `m`, must be provided as temporary workspace.

- `denseMatvec`

  `denseMatvec(a,x,y,m,n)` calculates the product $y = ax$ for a given vector `x` of length `n`, and `m` by `n` matrix `a`.

9.1.4 Functions in the BAND module

The BAND module defines two sets of functions with corresponding names. The first set contains functions (with names starting with a capital letter) that act on band matrices of type `DlsMat`. The second set contains functions (with names starting with a lower case letter) that act on matrices represented as simple arrays.

The following functions for `DlsMat` banded matrices are available in the BAND package. For full details, see the header files `sundials_direct.h` and `sundials_band.h`.

- `NewBandMat`: allocation of a `DlsMat` band matrix;
- `DestroyMat`: free memory for a `DlsMat` matrix;
- `PrintMat`: print a `DlsMat` matrix to standard output.
- `NewLintArray`: allocation of an array of `int` integers for use as pivots with `BandGBRF` and `BandGBRS`;
- `NewIntArray`: allocation of an array of `int` integers for use as pivots with the Lapack band solvers;
- `NewRealArray`: allocation of an array of `realtype` for use as right-hand side with `BandGBRS`;
- `DestroyArray`: free memory for an array;
- `SetToZero`: load a matrix with zeros;
- `AddIdentity`: increment a square matrix by the identity matrix;
- `BandCopy`: copy one matrix to another;
- `BandScale`: scale a matrix by a scalar;
- `BandGBTRF`: LU factorization with partial pivoting;
- `BandGBTRS`: solution of $Ax = b$ using LU factorization;
- `BandMatvec`: compute the product $y = Ax$, for a square band matrix A;

The following functions for small band matrices are available in the BAND package:

- `newBandMat`

  `newBandMat(n, smu, ml)` allocates storage for an `n` by `n` band matrix with lower half-bandwidth `ml`.

- `destroyMat`

  `destroyMat(a)` frees the band matrix `a` allocated by `newBandMat`;

- `newLintArray`

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

- `newIntArray`

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

- `newRealArray`

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

- `destroyArray`

  `destroyArray(p)` frees the array `p` allocated by `newLintArray`, `newIntArray`, or `newRealArray`;

- `bandCopy`

  `bandCopy(a,b,n,a_smu, b_smu,copymu, copyml)` copies the `n` by `n` band matrix `a` into the `n` by `n` band matrix `b`;

- `bandScale`

  `bandScale(c,a,n,mu,ml,smu)` scales every element in the `n` by `n` band matrix `a` by `c`;

- `bandAddIdentity`

  `bandAddIdentity(a,n,smu)` increments the `n` by `n` band matrix `a` by the identity matrix;

- `bandGETRF`

  `bandGETRF(a,n,mu,ml,smu,p)` factors the `n` by `n` band matrix `a`, using Gaussian elimination with row pivoting. It overwrites the elements of `a` with its LU factors and keeps track of the pivot rows chosen in the pivot array `p`.

- `bandGETRS`

  `bandGETRS(a,n,smu,ml,p,b)` solves the `n` by `n` linear system $ax = b$. It assumes that `a` (of size `n` $\times$ `n`) has been LU-factored and the pivot array `p` has been set by a successful call to `bandGETRF(a,n,mu,ml,smu,p)`. The solution x is written into the `b` array.

- `bandMatvec`

  `bandMatvec(a,x,y,n,mu,ml,smu)` calculates the product $y = ax$ for a given vector `x` of length `n`, and `n` by `n` band matrix `a`.

9.2 The SLS module

SUNDIALS provides a compressed-sparse-column matrix type and sparse matrix support functions. In addition, SUNDIALS provides interfaces to the publically available KLU and SuperLU_MT sparse direct solver packages. The files comprising the SLS matrix module, used in the KLU and SUPERLUMT linear solver packages, and their locations in the SUNDIALS *srcdir*, are as follows:

- header files (located in *srcdir*/`include/sundials`)
  `sundials_sparse.h`, `sundials_klu_impl.h`,
  `sundials_superlumt_impl.h`, `sundials_types.h`,
  `sundials_math.h`, `sundials_config.h`

- source files (located in *srcdir*/`src/sundials`)
  `sundials_sparse.c`, `sundials_math.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are relevant to the SLS package by itself:

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  `#define SUNDIALS_DOUBLE_PRECISION 1`
  `#define SUNDIALS_SINGLE_PRECISION 1`
  `#define SUNDIALS_EXTENDED_PRECISION 1`
- (optional) use of generic math functions: `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the macros `SUNMIN` and `SUNMAX`, and the function `SUNRabs`.

9.2.1 Type SlsMat

The type `SlsMat`, defined in `sundials_sparse.h` is a pointer to a structure defining a generic compressed-sparse-column matrix, and is used with all linear solvers in the *sls* family:

```
typedef struct _SlsMat {
  int M;
  int N;
  int NNZ;
  realtype *data;
  int *rowvals;
  int *colptrs;
} *SlsMat;
```

The fields of this structure are as follows (see Figure 9.2 for a diagram of the underlying compressed-sparse-column representation in a sparse matrix of type `SlsMat`). Note that a sparse matrix of type `SlsMat` need not be square.

M - number of rows

N - number of columns

NNZ - maximum number of nonzero entries in the matrix (allocated length of `data` and `rowvals` arrays)

data - pointer to a contiguous block of `realtype` variables (of length `NNZ`), containing the values of the nonzero entries in the matrix

rowvals - pointer to a contiguous block of `int` variables (of length `NNZ`), containing the row indices of each nonzero entry held in `data`

colptrs - pointer to a contiguous block of `int` variables (of length `N+1`). Each entry provides the index of the first column entry into the `data` and `rowvals` arrays, e.g. if `colptr[3]=7`, then the first nonzero entry in the fourth column of the matrix is located in `data[7]`, and is located in row `rowvals[7]` of the matrix. The last entry contains the total number of nonzero values in the matrix and hence points one past the end of the active data in the `data` and `rowvals` arrays.

For example, the 5×4 matrix

$$\begin{bmatrix} 0 & 3 & 1 & 0 \\ 3 & 0 & 0 & 2 \\ 0 & 7 & 0 & 0 \\ 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

could be stored in a `SlsMat` structure as either

```
M = 5;
N = 4;
NNZ = 8;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
rowvals = {1, 3, 0, 2, 0, 1, 3, 4};
colptrs = {0, 2, 4, 5, 8};
```

or

```
M = 5;
N = 4;
NNZ = 10;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *};
rowvals = {1, 3, 0, 2, 0, 1, 3, 4, *, *};
colptrs = {0, 2, 4, 5, 8};
```

where the first has no unused space, and the second has additional storage (the entries marked with `*` may contain any values). Note in both cases that the final value in `colptrs` is 8. The work associated with operations on the sparse matrix is proportional to this value and so one should use the best understanding of the number of nonzeroes here.

9.2.2 Functions in the SLS module

The SLS module defines functions that act on sparse matrices of type `SlsMat`. For full details, see the header file `sundials_sparse.h`.

- `NewSparseMat`

  `NewSparseMat(M, N, NNZ)` allocates storage for an `M` by `N` sparse matrix, with storage for up to `NNZ` nonzero entries.

- `SlsConvertDls`

  `SlsConvertDls(A)` converts a dense or band matrix `A` of type `DlsMat` into a new sparse matrix of type `SlsMat` by retaining only the nonzero values of the matrix `A`.

- `DestroySparseMat`

  `DestroySparseMat(A)` frees the memory for a sparse matrix `A` allocated by either `NewSparseMat` or `SlsConvertDls`.

- `SlsSetToZero(A)` zeros out the `SlsMat` matrix `A`. The storage for `A` is left unchanged.

- `CopySparseMat`

  `CopySparseMat(A, B)` copies the `SlsMat A` into the `SlsMat B`. It is assumed that the matrices have the same row/column dimensions. If `B` has insufficient storage to hold all the nonzero entries of `A`, the data and row index arrays in `B` are reallocated to match those in `A`.

- `ScaleSparseMat`

  `ScaleSparseMat(c, A)` scales every element in the `SlsMat A` by the `realtype` scalar `c`.

- `AddIdentitySparseMat`

  `AddIdentitySparseMat(A)` increments the `SlsMat A` by the identity matrix. If `A` is not square, only the existing diagonal values are incremented. Resizes the `data` and `rowvals` arrays of `A` to allow for new nonzero entries on the diagonal.

- `SlsAddMat`

  `SlsAddMat(A, B)` adds two `SlsMat` matrices `A` and `B`, placing the result back in `A`. Resizes the `data` and `rowvals` arrays of `A` upon completion to exactly match the nonzero storage for the result. Upon successful completion, the return value is zero; otherwise -1 is returned.

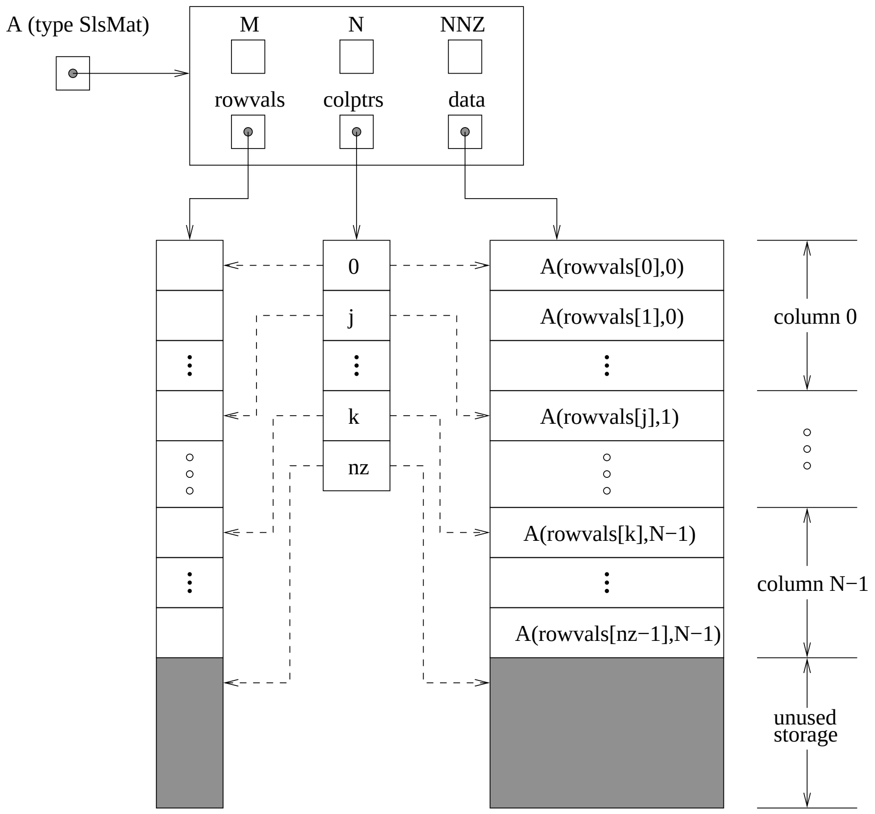


Figure 9.2: Diagram of the storage for a compressed-sparse-column matrix of type `SlsMat`. Here `A` is an $M \times N$ sparse matrix of type `SlsMat` with storage for up to `NNZ` nonzero entries (the allocated length of both `data` and `rowvals`). The entries in `rowvals` may assume values from 0 to $M - 1$, corresponding to the row index (zero-based) of each nonzero value. The entries in `data` contain the values of the nonzero entries, with the row i, column j entry of `A` (again, zero-based) denoted as `A(i,j)`. The `colptrs` array contains $N + 1$ entries; the first N denote the starting index of each column within the `rowvals` and `data` arrays, while the final entry points one past the final nonzero entry. Here, although `NNZ` values are allocated, only `nz` are actually filled in; the greyed-out portions of `data` and `rowvals` indicate extra allocated space.

- `ReallocSparseMat`

  `ReallocSparseMat(A)` eliminates unused storage in the `SlsMat A` by resizing the internal `data` and `rowvals` arrays to contain exactly `colptrs[N]` values.

- `SlsMatvec`

  `SlsMatvec(A, x, y)` computes the sparse matrix-vector product, $y = Ax$. If the `SlsMat A` is a sparse matrix of dimension $M \times N$, then it is assumed that `x` is a `realtype` array of length N, and `y` is a `realtype` array of length M. Upon successful completion, the return value is zero; otherwise -1 is returned.

- `PrintSparseMat`

  `PrintSparseMat(A)` Prints the `SlsMat` matrix `A` to standard output.

9.2.3 The KLU solver

klu is a sparse matrix factorization and solver library written by Tim Davis [1, 14]. klu has a symbolic factorization routine that computes the permutation of the linear system matrix to block triangular form and the permutations that will pre-order the diagonal blocks (the only ones that need to be factored) to reduce fill-in (using AMD, COLAMD, CHOLAMD, natural, or an ordering given by the user). Note that SUNDIALS uses the COLAMD ordering by default with klu.

klu breaks the factorization into two separate parts. The first is a symbolic factorization and the second is a numeric factorization that returns the factored matrix along with final pivot information. klu also has a refactor routine that can be called instead of the numeric factorization. This routine will reuse the pivot information. This routine also returns diagnostic information that a user can examine to determine if numerical stability is being lost and a full numerical factorization should be done instead of the refactor.

The klu interface in sundials will perform the symbolic factorization once. It then calls the numerical factorization once and will call the refactor routine until estimates of the numerical conditioning suggest a new factorization should be completed. The klu interface also has a `ReInit` routine that can be used to force a full refactorization at the next solver setup call.

In order to use the sundials interface to klu, it is assumed that klu has been installed on the system prior to installation of sundials, and that sundials has been configured appropriately to link with klu (see Appendix A for details).

Designed for serial calculations only, klu is supported for calculations employing sundials' serial or shared-memory parallel nvector modules (see Sections 7.1, 7.3 and 7.4 for details).

9.2.4 The SUPERLUMT solver

superlumt is a threaded sparse matrix factorization and solver library written by X. Sherry Li [2, 25, 15]. The package performs matrix factorization using threads to enhance efficiency in shared memory parallel environments. It should be noted that threads are only used in the factorization step.

In order to use the sundials interface to superlumt, it is assumed that superlumt has been installed on the system prior to installation of sundials, and that sundials has been configured appropriately to link with superlumt (see Appendix A for details).

Designed for serial and threaded calculations only, superlumt is supported for calculations employing sundials' serial or shared-memory parallel nvector modules (see Sections 7.1, 7.3 and 7.4 for details).

9.3 The SPILS modules: SPGMR, SPFGMR, SPBCG, and SPTFQMR

The *spils* modules contain implementations of some of the most commonly use scaled preconditioned Krylov solvers. A linear solver module from the *spils* family can be used in conjunction with any

NVECTOR implementation library.

9.3.1 The SPGMR module

The SPGMR package, in the files `sundials_spgmr.h` and `sundials_spgmr.c`, includes an implementation of the scaled preconditioned GMRES method. A separate code module, implemented in `sundials_iterative.(h,c)`, contains auxiliary functions that support SPGMR, as well as the other Krylov solvers in SUNDIALS (SPFGMR, SPBCG, and SPTFQMR). For full details, including usage instructions, see the header files `sundials_spgmr.h` and `sundials_iterative.h`.

The files comprising the SPGMR generic linear solver, and their locations in the SUNDIALS *srcdir*, are as follows:

- header files (located in *srcdir*`/include/sundials`)
  `sundials_spgmr.h`, `sundials_iterative.h`, `sundials_nvector.h`,
  `sundials_types.h`, `sundials_math.h`, `sundials_config.h`
- source files (located in *srcdir*`/src/sundials`)
  `sundials_spgmr.c`, `sundials_iterative.c`, `sundials_nvector.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are required to use the SPGMR package by itself:

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  ```
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
  ```
- (optional) use of generic math functions:
  ```
  #define SUNDIALS_USE_GENERIC_MATH 1
  ```

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the macros `SUNMIN`, `SUNMAX`, and `SUNSQR`, and the functions `SUNRabs` and `SUNRsqrt`.

The generic NVECTOR files, `sundials_nvector.(h,c)` are needed for the definition of the generic `N_Vector` type and functions. The NVECTOR functions used by the SPGMR module are: `N_VDotProd`, `N_VLinearSum`, `N_VScale`, `N_VProd`, `N_VDiv`, `N_VConst`, `N_VClone`, `N_VCloneVectorArray`, `N_VDestroy`, and `N_VDestroyVectorArray`.

The nine files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into an SPGMR library or into a larger user code.

The following functions are available in the SPGMR package:

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

The following functions are available in the support package `sundials_iterative.(h,c)`:

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

9.3.2 The SPFGMR module

The SPFGMR package, in the files `sundials_spfgmr.h` and `sundials_spfgmr.c`, includes an implementation of the scaled preconditioned Flexible GMRES method. For full details, including usage instructions, see the file `sundials_spfgmr.h`.

The files needed to use the SPFGMR module by itself are the same as for the SPGMR module, but with `sundials_spfgmr.(h,c)` in place of `sundials_spgmr.(h,c)`.

The following functions are available in the SPFGMR package:

- `SpfgmrMalloc`: allocation of memory for `SpfgmrSolve`;
- `SpfgmrSolve`: solution of $Ax = b$ by the SPFGMR method;
- `SpfgmrFree`: free memory allocated by `SpfgmrMalloc`.

9.3.3 The SPBCG module

The SPBCG package, in the files `sundials_spbcgs.h` and `sundials_spbcgs.c`, includes an implementation of the scaled preconditioned Bi-CGStab method. For full details, including usage instructions, see the file `sundials_spbcgs.h`.

The files needed to use the SPBCG module by itself are the same as for the SPGMR module, but with `sundials_spbcgs.(h,c)` in place of `sundials_spgmr.(h,c)`.

The following functions are available in the SPBCG package:

- `SpbcgMalloc`: allocation of memory for `SpbcgSolve`;
- `SpbcgSolve`: solution of $Ax = b$ by the SPBCG method;
- `SpbcgFree`: free memory allocated by `SpbcgMalloc`.

9.3.4 The SPTFQMR module

The SPTFQMR package, in the files `sundials_sptfqmr.h` and `sundials_sptfqmr.c`, includes an implementation of the scaled preconditioned TFQMR method. For full details, including usage instructions, see the file `sundials_sptfqmr.h`.

The files needed to use the SPTFQMR module by itself are the same as for the SPGMR module, but with `sundials_sptfqmr.(h,c)` in place of `sundials_spgmr.(h,c)`.

The following functions are available in the SPTFQMR package:

- `SptfqmrMalloc`: allocation of memory for `SptfqmrSolve`;
- `SptfqmrSolve`: solution of $Ax = b$ by the SPTFQMR method;
- `SptfqmrFree`: free memory allocated by `SptfqmrMalloc`.

Appendix A

SUNDIALS Package Installation Procedure

The installation of any SUNDIALS package is accomplished by installing the SUNDIALS suite as a whole, according to the instructions that follow. The same procedure applies whether or not the downloaded file contains one or all solvers in SUNDIALS.

The SUNDIALS suite (or individual solvers) are distributed as compressed archives (`.tar.gz`). The name of the distribution archive is of the form *solver*-`x.y.z.tar.gz`, where *solver* is one of: `sundials`, `cvode`, `cvodes`, `arkode`, `ida`, `idas`, or `kinsol`, and `x.y.z` represents the version number (of the SUNDIALS suite or of the individual solver) . To begin the installation, first uncompress and expand the sources, by issuing

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

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

Starting with version `2.6.0` of SUNDIALS, CMake is the only supported method of installation. The explanations on the installation procedure begins with a few common observations:

- The remainder of this chapter will follow these conventions:

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

  builddir is the (temporary) directory under which SUNDIALS is built.

  instdir is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory *instdir*`/include` while libraries are installed under *instdir*`/lib`, with *instdir* specified at configuration time.

- For SUNDIALS CMake-based installation, in-source builds are prohibited; in other words, the build directory *builddir* can **not** be the same as *srcdir* and such an attempt will lead to an error. This prevents "polluting" the source tree and allows efficient builds for different configurations and/or options.

- The installation directory *instdir* can **not** be the same as the source directory *srcdir*.

- By default, only the libraries and header files are exported to the installation directory *instdir*. If enabled by the user (with the appropriate toggle for CMake), the examples distributed with SUNDIALS will be built together with the solver libraries but the installation step will result in exporting (by default in a subdirectory of the installation directory) the example sources and sample outputs together with automatically generated configuration files that reference the *installed* SUNDIALS headers and libraries. As such, these configuration files for the SUNDIALS examples can be used as "templates" for your own problems. CMake installs `CMakeLists.txt` files and also (as an option available only under Unix/Linux) `Makefile` files. Note this installation approach also allows the option of building the SUNDIALS examples without having to install them. (This can be used as a sanity check for the freshly built libraries.)

- Even if generation of shared libraries is enabled, only static libraries are created for the FCMIX modules. (Because of the use of fixed names for the Fortran user-provided subroutines, FCMIX shared libraries would result in "undefined symbol" errors at link time.)

A.1 CMake-based installation

CMake-based installation provides a platform-independent build system. CMake can generate Unix and Linux Makefiles, as well as KDevelop, Visual Studio, and (Apple) XCode project files from the same configuration file. In addition, CMake also provides a GUI front end and which allows an interactive build and installation process.

The SUNDIALS build process requires CMake version `2.8.1` or higher and a working compiler. On Unix-like operating systems, it also requires Make (and `curses`, including its development libraries, for the GUI front end to CMake, `ccmake`), while on Windows it requires Visual Studio. While many Linux distributions offer CMake, the version included is probably out of date. Many new CMake features have been added recently, and you should download the latest version from `http://www.cmake.org`. Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake website. Once CMake is installed, Linux/Unix users will be able to use `ccmake`, while Windows users will be able to use `CMakeSetup`.

As previously noted, when using CMake to configure, build and install SUNDIALS, it is always required to use a separate build directory. While in-source builds are possible, they are explicitly prohibited by the SUNDIALS CMake scripts (one of the reasons being that, unlike autotools, CMake does not provide a `make distclean` procedure and it is therefore difficult to clean-up the source tree after an in-source build). By ensuring a separate build directory, it is an easy task for the user to clean-up all traces of the build by simply removing the build directory. CMake does generate a `make clean` which will remove files generated by the compiler and linker.

A.1.1 Configuring, building, and installing on Unix-like systems

The default CMake configuration will build all included solvers and associated examples and will build static and shared libraries. The *installdir* defaults to `/usr/local` and can be changed by setting the `CMAKE_INSTALL_PREFIX` variable. Support for FORTRAN and all other options are disabled.

CMake can be used from the command line with the `cmake` command, or from a `curses`-based GUI by using the `ccmake` command. Examples for using both methods will be presented. For the examples shown it is assumed that there is a top level SUNDIALS directory with appropriate source, build and install directories:

```
% mkdir (...)sundials/instdir
% mkdir (...)sundials/builddir
% cd (...)sundials/builddir
```

Building with the GUI

Using CMake with the GUI follows this general process:

- Select and modify values, run configure (`c` key)
- New values are denoted with an asterisk
- To set a variable, move the cursor to the variable and press enter
  - If it is a boolean (ON/OFF) it will toggle the value
  - If it is string or file, it will allow editing of the string

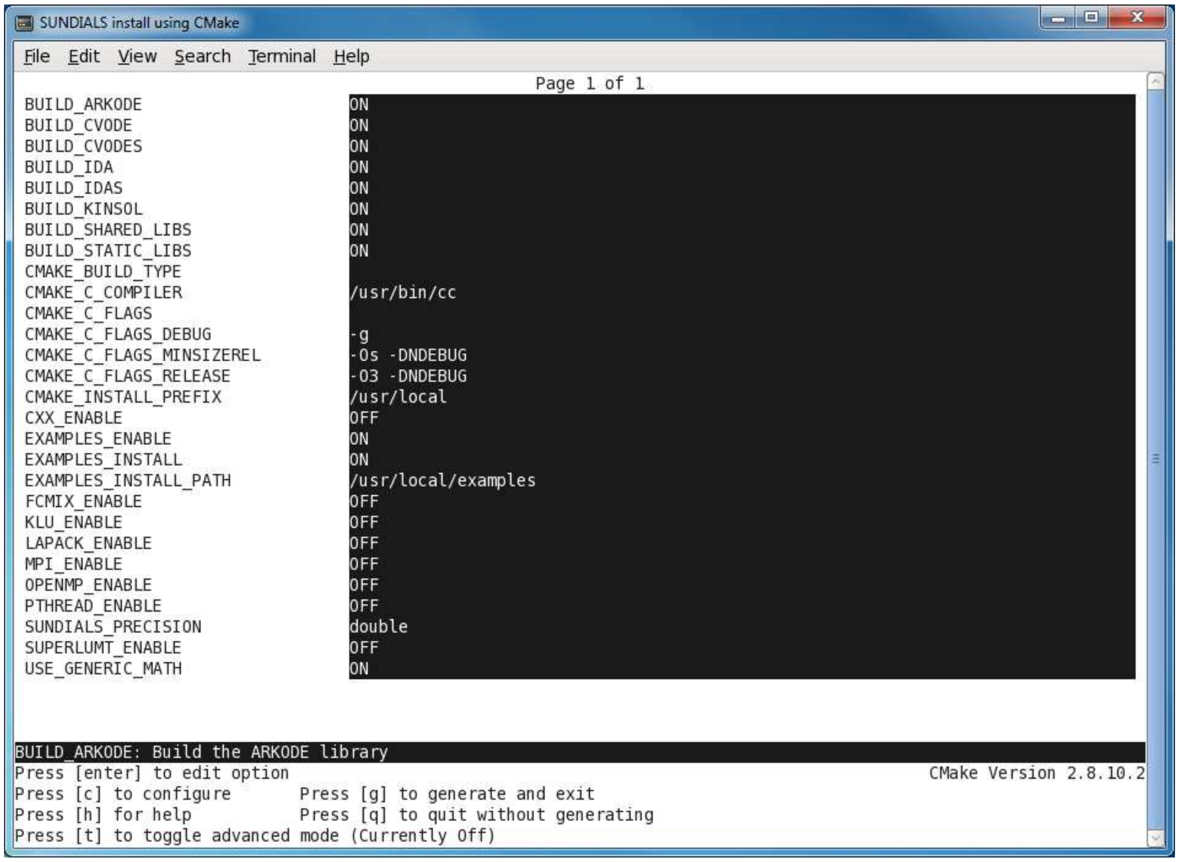
  – For file and directories, the `<tab>` key can be used to complete

- Repeat until all values are set as desired and the generate option is available (`g` key)
- Some variables (advanced variables) are not visible right away
- To see advanced variables, toggle to advanced mode (`t` key)
- To search for a variable press `/` key, and to repeat the search, press the `n` key

To build the default configuration using the GUI, from the *builddir* enter the ccmake command and point to the *srcdir*:

```
% ccmake ../srcdir
```

The default configuration screen is shown in Figure A.1.



```
                                          Page 1 of 1
BUILD_ARKODE                     ON
BUILD_CVODE                      ON
BUILD_CVODES                     ON
BUILD_IDA                        ON
BUILD_IDAS                       ON
BUILD_KINSOL                     ON
BUILD_SHARED_LIBS                ON
BUILD_STATIC_LIBS                ON
CMAKE_BUILD_TYPE
CMAKE_C_COMPILER                 /usr/bin/cc
CMAKE_C_FLAGS
CMAKE_C_FLAGS_DEBUG              -g
CMAKE_C_FLAGS_MINSIZEREL         -Os -DNDEBUG
CMAKE_C_FLAGS_RELEASE            -O3 -DNDEBUG
CMAKE_INSTALL_PREFIX             /usr/local
CXX_ENABLE                       OFF
EXAMPLES_ENABLE                  ON
EXAMPLES_INSTALL                 ON
EXAMPLES_INSTALL_PATH            /usr/local/examples
FCMIX_ENABLE                     OFF
KLU_ENABLE                       OFF
LAPACK_ENABLE                    OFF
MPI_ENABLE                       OFF
OPENMP_ENABLE                    OFF
PTHREAD_ENABLE                   OFF
SUNDIALS_PRECISION               double
SUPERLUMT_ENABLE                 OFF
USE_GENERIC_MATH                 ON



BUILD_ARKODE: Build the ARKODE library
Press [enter] to edit option                                       CMake Version 2.8.10.2
Press [c] to configure       Press [g] to generate and exit
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

SUNDIALS install using CMake
File Edit View Search Terminal Help

Figure A.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

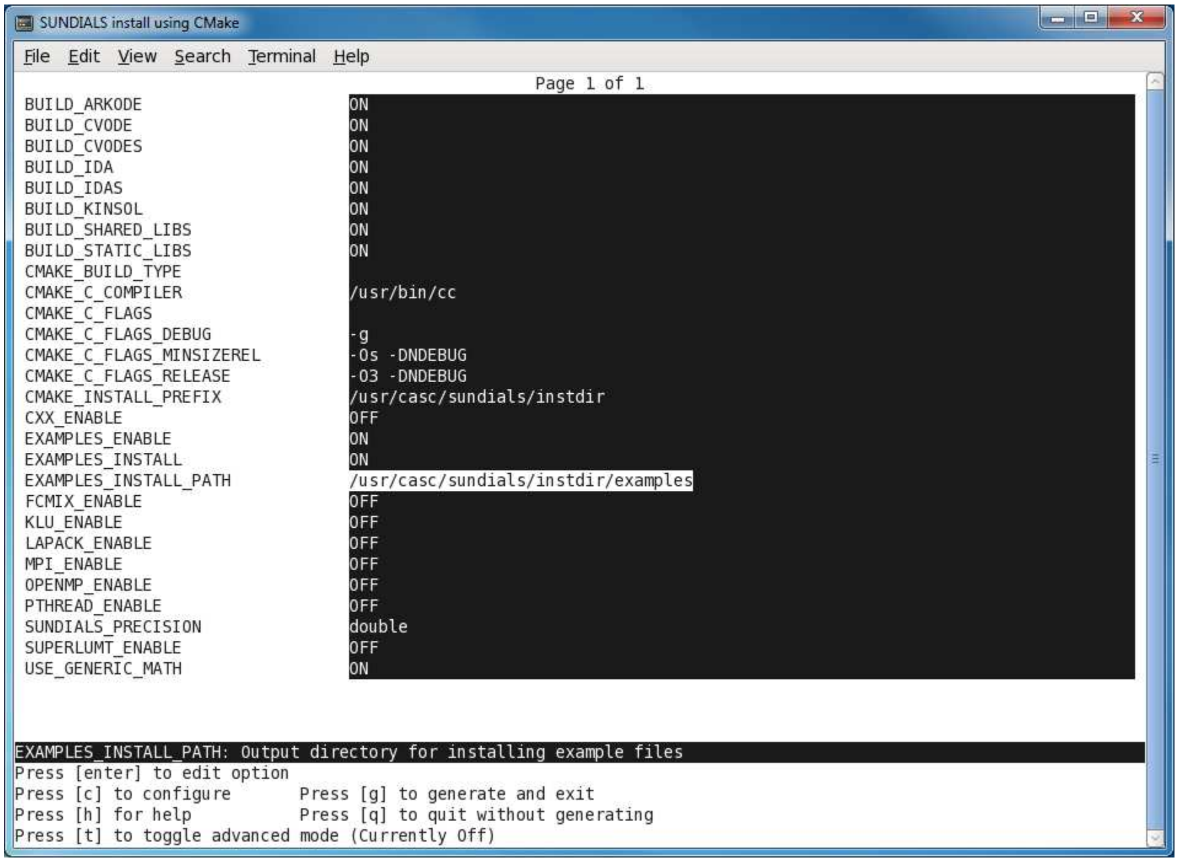
The default *instdir* for both SUNDIALS and corresponding examples can be changed by setting the `CMAKE_INSTALL_PREFIX` and the `EXAMPLES_INSTALL_PATH` as shown in figure A.2.

Pressing the (`g` key) will generate makefiles including all dependencies and all rules to build SUNDIALS on this system. Back at the command prompt, you can now run:

```
% make
```

To install SUNDIALS in the installation directory specified in the configuration, simply run:

```
% make install
```


SUNDIALS install using CMake
File Edit View Search Terminal Help

```
                                Page 1 of 1
BUILD_ARKODE                     ON
BUILD_CVODE                      ON
BUILD_CVODES                     ON
BUILD_IDA                        ON
BUILD_IDAS                       ON
BUILD_KINSOL                     ON
BUILD_SHARED_LIBS                ON
BUILD_STATIC_LIBS                ON
CMAKE_BUILD_TYPE
CMAKE_C_COMPILER                 /usr/bin/cc
CMAKE_C_FLAGS
CMAKE_C_FLAGS_DEBUG              -g
CMAKE_C_FLAGS_MINSIZEREL         -Os -DNDEBUG
CMAKE_C_FLAGS_RELEASE            -O3 -DNDEBUG
CMAKE_INSTALL_PREFIX             /usr/casc/sundials/instdir
CXX_ENABLE                       OFF
EXAMPLES_ENABLE                  ON
EXAMPLES_INSTALL                 ON
EXAMPLES_INSTALL_PATH            /usr/casc/sundials/instdir/examples
FCMIX_ENABLE                     OFF
KLU_ENABLE                       OFF
LAPACK_ENABLE                    OFF
MPI_ENABLE                       OFF
OPENMP_ENABLE                    OFF
PTHREAD_ENABLE                   OFF
SUNDIALS_PRECISION               double
SUPERLUMT_ENABLE                 OFF
USE_GENERIC_MATH                 ON


EXAMPLES_INSTALL_PATH: Output directory for installing example files
Press [enter] to edit option
Press [c] to configure       Press [g] to generate and exit
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Figure A.2: Changing the *instdir* for SUNDIALS and corresponding examples

Building from the command line

Using CMake from the command line is simply a matter of specifying CMake variable settings with the cmake command. The following will build the default configuration:

```
% cmake -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> ../srcdir
% make
% make install
```

A.1.2 Configuration options (Unix/Linux)

A complete list of all available options for a CMake-based SUNDIALS configuration is provide below. Note that the default values shown are for a typical configuration on a Linux system and are provided as illustration only.

BUILD_ARKODE - Build the ARKODE library
Default: ON

BUILD_CVODE - Build the CVODE library
Default: ON

BUILD_CVODES - Build the CVODES library
Default: ON

`BUILD_IDA` - Build the IDA library
Default: ON

`BUILD_IDAS` - Build the IDAS library
Default: ON

`BUILD_KINSOL` - Build the KINSOL library
Default: ON

`BUILD_SHARED_LIBS` - Build shared libraries
Default: OFF

`BUILD_STATIC_LIBS` - Build static libraries
Default: ON

`CMAKE_BUILD_TYPE` - Choose the type of build, options are: None (CMAKE_C_FLAGS used) Debug Release RelWithDebInfo MinSizeRel
Default:

`CMAKE_C_COMPILER` - C compiler
Default: /usr/bin/cc

`CMAKE_C_FLAGS` - Flags for C compiler
Default:

`CMAKE_C_FLAGS_DEBUG` - Flags used by the compiler during debug builds
Default: -g

`CMAKE_C_FLAGS_MINSIZEREL` - Flags used by the compiler during release minsize builds
Default: -Os -DNDEBUG

`CMAKE_C_FLAGS_RELEASE` - Flags used by the compiler during release builds
Default: -O3 -DNDEBUG

`CMAKE_Fortran_COMPILER` - Fortran compiler
Default: /usr/bin/gfortran
Note: Fortran support (and all related options) are triggered only if either Fortran-C support is enabled (`FCMIX_ENABLE` is ON) or Blas/Lapack support is enabled (`LAPACK_ENABLE` is ON).

`CMAKE_Fortran_FLAGS` - Flags for Fortran compiler
Default:

`CMAKE_Fortran_FLAGS_DEBUG` - Flags used by the compiler during debug builds
Default:

`CMAKE_Fortran_FLAGS_MINSIZEREL` - Flags used by the compiler during release minsize builds
Default:

`CMAKE_Fortran_FLAGS_RELEASE` - Flags used by the compiler during release builds
Default:

`CMAKE_INSTALL_PREFIX` - Install path prefix, prepended onto install directories
Default: /usr/local
Note: The user must have write access to the location specified through this option. Exported SUNDIALS header files and libraries will be installed under subdirectories `include` and `lib` of `CMAKE_INSTALL_PREFIX`, respectively.

`EXAMPLES_ENABLE` - Build the SUNDIALS examples
Default: ON

`EXAMPLES_INSTALL` - Install example files
Default: ON
Note: This option is triggered only if building example programs is enabled (`EXAMPLES_ENABLE` ON). If the user requires installation of example programs then the sources and sample output files for all SUNDIALS modules that are currently enabled will be exported to the directory specified by `EXAMPLES_INSTALL_PATH`. A CMake configuration script will also be automatically generated and exported to the same directory. Additionally, if the configuration is done under a Unix-like system, makefiles for the compilation of the example programs (using the installed SUNDIALS libraries) will be automatically generated and exported to the directory specified by `EXAMPLES_INSTALL_PATH`.

`EXAMPLES_INSTALL_PATH` - Output directory for installing example files
Default: /usr/local/examples
Note: The actual default value for this option will an `examples` subdirectory created under `CMAKE_INSTALL_PREFIX`.

`FCMIX_ENABLE` - Enable Fortran-C support
Default: OFF

`KLU_ENABLE` - Enable KLU support
Default: OFF

`LAPACK_ENABLE` - Enable Lapack support
Default: OFF
Note: Setting this option to ON will trigger the two additional options see below.

`LAPACK_LIBRARIES` - Lapack (and Blas) libraries
Default: /usr/lib/liblapack.so;/usr/lib/libblas.so
Note: CMake will search for these libraries in your `LD_LIBRARY_PATH` prior to searching default system paths.

`MPI_ENABLE` - Enable MPI support
Default: OFF
Note: Setting this option to ON will trigger several additional options related to MPI.

`MPI_MPICC` - `mpicc` program
Default:

`MPI_RUN_COMMAND` - Specify run command for MPI
Default: mpirun
Note: This can either be set to `mpirun` for OpenMPI or `srun` if jobs are managed by `SLURM` - Simple Linux Utility for Resource Management as exists on LLNL's high performance computing clusters.

`MPI_MPIF77` - `mpif77` program
Default:
Note: This option is triggered only if using MPI compiler scripts (`MPI_USE_MPISCRIPTS` is ON) and Fortran-C support is enabled (`FCMIx_ENABLE` is ON).

`OPENMP_ENABLE` - Enable OpenMP support
Default: OFF
Turn on support for the OpenMP based nvector.

`PTHREAD_ENABLE` - Enable Pthreads support
Default: OFF
Turn on support for the Pthreads based nvector.

`SUNDIALS_PRECISION` - Precision used in SUNDIALS, options are: double, single or extended
Default: double

`SUPERLUMT_ENABLE` - Enable SUPERLU_MT support
Default: OFF

`USE_GENERIC_MATH` - Use generic (stdc) math libraries
Default: ON

A.1.3 Configuration examples

The following examples will help demonstrate usage of the CMake configure options.
To configure SUNDIALS using the default C and Fortran compilers, and default `mpicc` and `mpif77` parallel compilers, enable compilation of examples, and install libraries, headers, and example sources under subdirectories of `/home/myname/sundials/`, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
> /home/myname/sundials/srcdir
%
% make install
%
```

To disable installation of the examples, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
> -DEXAMPLES_INSTALL=OFF \
> /home/myname/sundials/srcdir
%
% make install
%
```

A.1.4 Working with external Libraries

The SUNDIALS Suite contains many options to enable implementation flexibility when developing solutions. The following are some notes addressing specific configurations when using the supported third party libraries.

Building with LAPACK and BLAS

To enable LAPACK and BLAS libraries, set the `LAPACK_ENABLE` option to `ON`. If the directory containing the LAPACK and BLAS libraries is in the `LD_LIBRARY_PATH` environment variable, CMake will set the `LAPACK_LIBRARIES` variable accordingly, otherwise CMake will attemp to find the LAPACK and BLAS libraries in standard system locations. To explicitly tell CMake what libraries to use, the `LAPACK_LIBRARIES` varible can be set to the desired libraries. Example:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DLAPACK_LIBRARIES=/mypath/lib/liblapack.so;/mypath/lib/libblas.so \
> /home/myname/sundials/srcdir
%
% make install
%
```

Building with KLU

The KLU libraries are part of SuiteSparse, a suite of sparse matrix software, available from the Texas A&M University website: `http://faculty.cse.tamu.edu/davis/suitesparse.html`. SUNDIALS has been tested with SuiteSparse version 4.2.1. To enable KLU, set `KLU_ENABLE` to `ON`, set `KLU_INCLUDE_DIR` to the `include` path of the KLU installation and set `KLU_LIBRARY_DIR` to the `lib` path of the KLU installation. The CMake configure will result in populating the following variables: `AMD_LIBRARY`, `AMD_LIBRARY_DIR`, `BTF_LIBRARY`, `BTF_LIBRARY_DIR`, `COLAMD_LIBRARY`, `COLAMD_LIBRARY_DIR`, and `KLU_LIBRARY`

Building with SuperLU_MT

The SuperLU_MT libraries are available for download from the Lawrence Berkeley National Laboratory website: `http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu_mt`. SUNDIALS has been tested with SuperLU_MT version 2.4. To enable SuperLU_MT, set `SUPERLUMT_ENABLE` to `ON`, set `SUPERLUMT_INCLUDE_DIR` to the `SRC` path of the SuperLU_MT installation, and set the variable `SUPERLUMT_LIBRARY_DIR` to the `lib` path of the SuperLU_MT installation. At the same time, the variable `SUPERLUMT_THREAD_TYPE` must be set to either `Pthread` or `OpenMP`.
Do not mix thread types when building SUNDIALS solvers. If threading is enabled for SUNDIALS by having either `OPENMP_ENABLE` or `PTHREAD_ENABLE` set to `ON` then SuperLU_MT should be set to use the same threading type.

A.2 Building and Running Examples

Each of the SUNDIALS solvers is distributed with a set of examples demonstrating basic usage. To build and install the examples, set both `EXAMPLES_ENABLE` and `EXAMPLES_INSTALL` to `ON`. Specify the installation path for the examples with the variable `EXAMPLES_INSTALL_PATH`. CMake will generate `CMakeLists.txt` configuration files (and `Makefile` files if on Linux/Unix) that reference the *installed* SUNDIALS headers and libraries.

Either the `CMakeLists.txt` file or the traditional `Makefile` may be used to build the examples as well as serve as a template for creating user developed solutions. To use the supplied `Makefile` simply run `make` to compile and generate the executables. To use CMake from within the installed example directory, run `cmake` (or `ccmake` to use the GUI) followed by `make` to compile the example code. Note that if CMake is used, it will overwrite the traditional `Makefile` with a new CMake-generated `Makefile`. The resulting output from running the examples can be compared with example output bundled in the SUNDIALS distribution.
NOTE: There will potentially be differences in the output due to machine architecture, compiler versions, use of third party libraries etc.

A.3 Configuring, building, and installing on Windows

CMake can also be used to build SUNDIALS on Windows. To build SUNDIALS for use with Visual Studio the following steps should be performed:

1. Unzip the downloaded tar file(s) into a directory. This will be the *srcdir*
2. Create a separate *builddir*
3. Open a Visual Studio Command Prompt and cd to *builddir*
4. Run cmake-gui ../*srcdir*
   (a) Hit Configure
   (b) Check/Uncheck solvers to be built
   (c) Change CMAKE_INSTALL_PREFIX to *instdir*

   (d) Set other options as desired
   (e) Hit Generate

5. Back in the VS Command Window:
   (a) Run msbuild ALL_BUILD.vcxproj
   (b) Run msbuild INSTALL.vcxproj

The resulting libraries will be in the *instdir*. The SUNDIALS project can also now be opened in Visual Studio. Double click on the ALL_BUILD.vcxproj file to open the project. Build the whole *solution* to create the SUNDIALS libraries. To use the SUNDIALS libraries in your own projects, you must set the include directories for your project, add the SUNDIALS libraries to your project solution, and set the SUNDIALS libraries as dependencies for your project.

A.4 Installed libraries and exported header files

Using the CMake SUNDIALS build system, the command

```
% make install
```

will install the libraries under *libdir* and the public header files under *includedir*. The values for these directories are *instdir*/`lib` and *instdir*/`include`, respectively. The location can be changed by setting the CMake variable `CMAKE_INSTALL_PREFIX`. Although all installed libraries reside under *libdir*/`lib`, the public header files are further organized into subdirectories under *includedir*/`include`.

The installed libraries and exported header files are listed for reference in Tables A.1 and A.2. The file extension *.lib* is typically `.so` for shared libraries and `.a` for static libraries. Note that, in the Tables, names are relative to *libdir* for libraries and to *includedir* for header files.

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

Table A.1: SUNDIALS libraries and header files

SHARED	Libraries	n/a	
	Header files	sundials/sundials_config.h	sundials/sundials_types.h
		sundials/sundials_math.h	
		sundials/sundials_nvector.h	sundials/sundials_fnvector.h
		sundials/sundials_direct.h	sundials/sundials_lapack.h
		sundials/sundials_dense.h	sundials/sundials_band.h
		sundials/sundials_sparse.h	
		sundials/sundials_iterative.h	sundials/sundials_spgmr.h
		sundials/sundials_spbcgs.h	sundials/sundials_sptfqmr.h
		sundials/sundials_pcg.h	sundials/sundials_spfgmr.h
NVECTOR_SERIAL	Libraries	libsundials_nvecserial.*lib*	libsundials_fnvecserial.a
	Header files	nvector/nvector_serial.h	
NVECTOR_PARALLEL	Libraries	libsundials_nvecparallel.*lib*	libsundials_fnvecparallel.a
	Header files	nvector/nvector_parallel.h	
NVECTOR_OPENMP	Libraries	libsundials_nvecopenmp.*lib*	libsundials_fnvecopenmp.a
	Header files	nvector/nvector_openmp.h	
NVECTOR_PTHREADS	Libraries	libsundials_nvecpthreads.*lib*	libsundials_fnvecpthreads.a
	Header files	nvector/nvector_pthreads.h	
CVODE	Libraries	libsundials_cvode.*lib*	libsundials_fcvode.a
	Header files	cvode/cvode.h	cvode/cvode_impl.h
		cvode/cvode_direct.h	cvode/cvode_lapack.h
		cvode/cvode_dense.h	cvode/cvode_band.h
		cvode/cvode_diag.h	
		cvode/cvode_sparse.h	cvode/cvode_klu.h
		cvode/cvode_superlumt.h	
		cvode/cvode_spils.h	cvode/cvode_spgmr.h
		cvode/cvode_sptfqmr.h	cvode/cvode_spbcgs.h
		cvode/cvode_bandpre.h	cvode/cvode_bbdpre.h
CVODES	Libraries	libsundials_cvodes.*lib*	
	Header files	cvodes/cvodes.h	cvodes/cvodes_impl.h
		cvodes/cvodes_direct.h	cvodes/cvodes_lapack.h
		cvodes/cvodes_dense.h	cvodes/cvodes_band.h
		cvodes/cvodes_diag.h	
		cvodes/cvodes_sparse.h	cvodes/cvodes_klu.h
		cvodes/cvodes_superlumt.h	
		cvodes/cvodes_spils.h	cvodes/cvodes_spgmr.h
		cvodes/cvodes_sptfqmr.h	cvodes/cvodes_spbcgs.h
		cvodes/cvodes_bandpre.h	cvodes/cvodes_bbdpre.h
ARKODE	Libraries	libsundials_arkode.*lib*	libsundials_farkode.a
	Header files	arkode/arkode.h	arkode/arkode_impl.h
		arkode/arkode_direct.h	arkode/arkode_lapack.h
		arkode/arkode_dense.h	arkode/arkode_band.h
		arkode/arkode_sparse.h	arkode/arkode_klu.h
		arkode/arkode_superlumt.h	
		arkode/arkode_spils.h	arkode/arkode_spgmr.h
		arkode/arkode_sptfqmr.h	arkode/arkode_spbcgs.h
		arkode/arkode_pcg.h	arkode/arkode_spfgmr.h
		arkode/arkode_bandpre.h	arkode/arkode_bbdpre.h

Table A.2: SUNDIALS libraries and header files (cont.)

IDA	Libraries	libsundials_ida.*lib*	libsundials_fida.a
	Header files	ida/ida.h	ida/ida_impl.h
		ida/ida_direct.h	ida/ida_lapack.h
		ida/ida_dense.h	ida/ida_band.h
		ida/ida_sparse.h	ida/ida_klu.h
		ida/ida_superlumt.h	
		ida/ida_spils.h	ida/ida_spgmr.h
		ida/ida_spbcgs.h	ida/ida_sptfqmr.h
		ida/ida_bbdpre.h	
IDAS	Libraries	libsundials_idas.*lib*	
	Header files	idas/idas.h	idas/idas_impl.h
		idas/idas_direct.h	idas/idas_lapack.h
		idas/idas_dense.h	idas/idas_band.h
		idas/idas_sparse.h	idas/idas_klu.h
		idas/idas_superlumt.h	
		idas/idas_spils.h	idas/idas_spgmr.h
		idas/idas_spbcgs.h	idas/idas_sptfqmr.h
		idas/idas_bbdpre.h	
KINSOL	Libraries	libsundials_kinsol.*lib*	libsundials_fkinsol.a
	Header files	kinsol/kinsol.h	kinsol/kinsol_impl.h
		kinsol/kinsol_direct.h	kinsol/kinsol_lapack.h
		kinsol/kinsol_dense.h	kinsol/kinsol_band.h
		kinsol/kinsol_sparse.h	kinsol/kinsol_klu.h
		kinsol/kinsol_superlumt.h	
		kinsol/kinsol_spils.h	kinsol/kinsol_spgmr.h
		kinsol/kinsol_spbcgs.h	kinsol/kinsol_sptfqmr.h
		kinsol/kinsol_bbdpre.h	kinsol/kinsol_spfgmr.h

Appendix B

IDAS Constants

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

B.1 IDAS input constants

IDAS **main solver module**		
`IDA_NORMAL`	1	Solver returns at specified output time.
`IDA_ONE_STEP`	2	Solver returns after each successful step.
`IDA_SIMULTANEOUS`	1	Simultaneous corrector forward sensitivity method.
`IDA_STAGGERED`	2	Staggered corrector forward sensitivity method.
`IDA_CENTERED`	1	Central difference quotient approximation (2^{nd} order) of the sensitivity RHS.
`IDA_FORWARD`	2	Forward difference quotient approximation (1^{st} order) of the sensitivity RHS.
`IDA_YA_YDP_INIT`	1	Compute y_a and $\dot{y}_d$, given y_d.
`IDA_Y_INIT`	2	Compute y, given $\dot{y}$.

IDAS **adjoint solver module**		
`IDA_HERMITE`	1	Use Hermite interpolation.
`IDA_POLYNOMIAL`	2	Use variable-degree polynomial interpolation.

Iterative linear solver module		
`PREC_NONE`	0	No preconditioning
`PREC_LEFT`	1	Preconditioning on the left.
`MODIFIED_GS`	1	Use modified Gram-Schmidt procedure.
`CLASSICAL_GS`	2	Use classical Gram-Schmidt procedure.

B.2 IDAS output constants

IDAS **main solver module**		
`IDA_SUCCESS`	0	Successful function return.
`IDA_TSTOP_RETURN`	1	`IDASolve` succeeded by reaching the specified stopping point.

IDA_ROOT_RETURN	2	IDASolve succeeded and found one or more roots.
IDA_WARNING	99	IDASolve succeeded but an unusual situation occurred.
IDA_TOO_MUCH_WORK	-1	The solver took mxstep internal steps but could not reach tout.
IDA_TOO_MUCH_ACC	-2	The solver could not satisfy the accuracy demanded by the user for some internal step.
IDA_ERR_FAIL	-3	Error test failures occurred too many times during one internal time step or minimum step size was reached.
IDA_CONV_FAIL	-4	Convergence test failures occurred too many times during one internal time step or minimum step size was reached.
IDA_LINIT_FAIL	-5	The linear solver's initialization function failed.
IDA_LSETUP_FAIL	-6	The linear solver's setup function failed in an unrecoverable manner.
IDA_LSOLVE_FAIL	-7	The linear solver's solve function failed in an unrecoverable manner.
IDA_RES_FAIL	-8	The user-provided residual function failed in an unrecoverable manner.
IDA_REP_RES_FAIL	-9	The user-provided residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.
IDA_RTFUNC_FAIL	-10	The rootfinding function failed in an unrecoverable manner.
IDA_CONSTR_FAIL	-11	The inequality constraints were violated and the solver was unable to recover.
IDA_FIRST_RES_FAIL	-12	The user-provided residual function failed recoverably on the first call.
IDA_LINESEARCH_FAIL	-13	The line search failed.
IDA_NO_RECOVERY	-14	The residual function, linear solver setup function, or linear solver solve function had a recoverable failure, but IDACalcIC could not recover.
IDA_MEM_NULL	-20	The ida_mem argument was NULL.
IDA_MEM_FAIL	-21	A memory allocation failed.
IDA_ILL_INPUT	-22	One of the function inputs is illegal.
IDA_NO_MALLOC	-23	The IDAS memory was not allocated by a call to IDAInit.
IDA_BAD_EWT	-24	Zero value of some error weight component.
IDA_BAD_K	-25	The k-th derivative is not available.
IDA_BAD_T	-26	The time t is outside the last step taken.
IDA_BAD_DKY	-27	The vector argument where derivative should be stored is NULL.
IDA_NO_QUAD	-30	Quadratures were not initialized.
IDA_QRHS_FAIL	-31	The user-provided right-hand side function for quadratures failed in an unrecoverable manner.
IDA_FIRST_QRHS_ERR	-32	The user-provided right-hand side function for quadratures failed in an unrecoverable manner on the first call.
IDA_REP_QRHS_ERR	-33	The user-provided right-hand side repeatedly returned a recoverable error flag, but the solver was unable to recover.
IDA_NO_SENS	-40	Sensitivities were not initialized.
IDA_SRES_FAIL	-41	The user-provided sensitivity residual function failed in an unrecoverable manner.
IDA_REP_SRES_ERR	-42	The user-provided sensitivity residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.

`IDA_BAD_IS`	-43	The sensitivity identifier is not valid.
`IDA_NO_QUADSENS`	-50	Sensitivity-dependent quadratures were not initialized.
`IDA_QSRHS_FAIL`	-51	The user-provided sensitivity-dependent quadrature right-hand side function failed in an unrecoverable manner.
`IDA_FIRST_QSRHS_ERR`	-52	The user-provided sensitivity-dependent quadrature right-hand side function failed in an unrecoverable manner on the first call.
`IDA_REP_QSRHS_ERR`	-53	The user-provided sensitivity-dependent quadrature right-hand side repeatedly returned a recoverable error flag, but the solver was unable to recover.

IDAS adjoint solver module

`IDA_NO_ADJ`	-101	The combined forward-backward problem has not been initialized.
`IDA_NO_FWD`	-102	`IDASolveF` has not been previously called.
`IDA_NO_BCK`	-103	No backward problem was specified.
`IDA_BAD_TB0`	-104	The desired output for backward problem is outside the interval over which the forward problem was solved.
`IDA_REIFWD_FAIL`	-105	No checkpoint is available for this hot start.
`IDA_FWD_FAIL`	-106	`IDASolveB` failed because `IDASolve` was unable to store data between two consecutive checkpoints.
`IDA_GETY_BADT`	-107	Wrong time in interpolation function.

IDADLS linear solver modules

`IDADLS_SUCCESS`	0	Successful function return.
`IDADLS_MEM_NULL`	-1	The `ida_mem` argument was `NULL`.
`IDADLS_LMEM_NULL`	-2	The IDADLS linear solver has not been initialized.
`IDADLS_ILL_INPUT`	-3	The IDADLS solver is not compatible with the current NVECTOR module.
`IDADLS_MEM_FAIL`	-4	A memory allocation request failed.
`IDADLS_JACFUNC_UNRECVR`	-5	The Jacobian function failed in an unrecoverable manner.
`IDADLS_JACFUNC_RECVR`	-6	The Jacobian function had a recoverable error.
`IDADLS_NO_ADJ`	-101	The combined forward-backward problem has not been initialized.
`IDADLS_LMEMB_NULL`	-102	The linear solver was not initialized for the backward phase.

IDASLS linear solver module

`IDASLS_SUCCESS`	0	Successful function return.
`IDASLS_MEM_NULL`	-1	The `ida_mem` argument was `NULL`.
`IDASLS_LMEM_NULL`	-2	The IDASLS linear solver has not been initialized.
`IDASLS_ILL_INPUT`	-3	The IDASLS solver is not compatible with the current NVECTOR module or other input is invalid.
`IDASLS_MEM_FAIL`	-4	A memory allocation request failed.
`IDASLS_JAC_NOSET`	-5	The Jacobian evaluation routine was not been set before the linear solver setup routine was called.

IDASLS_PACKAGE_FAIL	-6	An external package call return a failure error code.
IDASLS_JACFUNC_UNRECVR	-7	The Jacobian function failed in an unrecoverable manner.
IDASLS_JACFUNC_RECVR	-8	The Jacobian function had a recoverable error.
IDASLS_NO_ADJ	-101	The combined forward-backward problem has not been initialized.
IDASLS_LMEMB_NULL	-102	The linear solver was not initialized for the backward phase.

IDASPILS linear solver modules

IDASPILS_SUCCESS	0	Successful function return.
IDASPILS_MEM_NULL	-1	The `ida_mem` argument was `NULL`.
IDASPILS_LMEM_NULL	-2	The IDASPILS linear solver has not been initialized.
IDASPILS_ILL_INPUT	-3	The IDASPILS solver is not compatible with the current NVECTOR module.
IDASPILS_MEM_FAIL	-4	A memory allocation request failed.
IDASPILS_PMEM_NULL	-5	The preconditioner module has not been initialized.
IDASPILS_NO_ADJ	-101	The combined forward-backward problem has not been initialized.
IDASPILS_LMEMB_NULL	-102	The linear solver was not initialized for the backward phase.

SPGMR generic linear solver module

SPGMR_SUCCESS	0	Converged.
SPGMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPGMR_CONV_FAIL	2	Failure to converge.
SPGMR_QRFACT_FAIL	3	A singular matrix was found during the QR factorization.
SPGMR_PSOLVE_FAIL_REC	4	The preconditioner solve function failed recoverably.
SPGMR_ATIMES_FAIL_REC	5	The Jacobian-times-vector function failed recoverably.
SPGMR_PSET_FAIL_REC	6	The preconditioner setup routine failed recoverably.
SPGMR_MEM_NULL	-1	The SPGMR memory is `NULL`
SPGMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPGMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPGMR_GS_FAIL	-4	Failure in the Gram-Schmidt procedure.
SPGMR_QRSOL_FAIL	-5	The matrix R was found to be singular during the QR solve phase.
SPGMR_PSET_FAIL_UNREC	-6	The preconditioner setup routine failed unrecoverably.

SPFGMR generic linear solver module (only available in KINSOL and ARKODE)

SPFGMR_SUCCESS	0	Converged.
SPFGMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPFGMR_CONV_FAIL	2	Failure to converge.
SPFGMR_QRFACT_FAIL	3	A singular matrix was found during the QR factorization.
SPFGMR_PSOLVE_FAIL_REC	4	The preconditioner solve function failed recoverably.
SPFGMR_ATIMES_FAIL_REC	5	The Jacobian-times-vector function failed recoverably.
SPFGMR_PSET_FAIL_REC	6	The preconditioner setup routine failed recoverably.
SPFGMR_MEM_NULL	-1	The SPFGMR memory is `NULL`
SPFGMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.

`SPFGMR_PSOLVE_FAIL_UNREC`	-3	The preconditioner solve function failed unrecoverably.
`SPFGMR_GS_FAIL`	-4	Failure in the Gram-Schmidt procedure.
`SPFGMR_QRSOL_FAIL`	-5	The matrix R was found to be singular during the QR solve phase.
`SPFGMR_PSET_FAIL_UNREC`	-6	The preconditioner setup routine failed unrecoverably.

SPBCG generic linear solver module

`SPBCG_SUCCESS`	0	Converged.
`SPBCG_RES_REDUCED`	1	No convergence, but the residual norm was reduced.
`SPBCG_CONV_FAIL`	2	Failure to converge.
`SPBCG_PSOLVE_FAIL_REC`	3	The preconditioner solve function failed recoverably.
`SPBCG_ATIMES_FAIL_REC`	4	The Jacobian-times-vector function failed recoverably.
`SPBCG_PSET_FAIL_REC`	5	The preconditioner setup routine failed recoverably.
`SPBCG_MEM_NULL`	-1	The SPBCG memory is `NULL`
`SPBCG_ATIMES_FAIL_UNREC`	-2	The Jacobian-times-vector function failed unrecoverably.
`SPBCG_PSOLVE_FAIL_UNREC`	-3	The preconditioner solve function failed unrecoverably.
`SPBCG_PSET_FAIL_UNREC`	-4	The preconditioner setup routine failed unrecoverably.

SPTFQMR generic linear solver module

`SPTFQMR_SUCCESS`	0	Converged.
`SPTFQMR_RES_REDUCED`	1	No convergence, but the residual norm was reduced.
`SPTFQMR_CONV_FAIL`	2	Failure to converge.
`SPTFQMR_PSOLVE_FAIL_REC`	3	The preconditioner solve function failed recoverably.
`SPTFQMR_ATIMES_FAIL_REC`	4	The Jacobian-times-vector function failed recoverably.
`SPTFQMR_PSET_FAIL_REC`	5	The preconditioner setup routine failed recoverably.
`SPTFQMR_MEM_NULL`	-1	The SPTFQMR memory is `NULL`
`SPTFQMR_ATIMES_FAIL_UNREC`	-2	The Jacobian-times-vector function failed.
`SPTFQMR_PSOLVE_FAIL_UNREC`	-3	The preconditioner solve function failed unrecoverably.
`SPTFQMR_PSET_FAIL_UNREC`	-4	The preconditioner setup routine failed unrecoverably.

Bibliography

[1] KLU Sparse Matrix Factorization Library. http://faculty.cse.tamu.edu/davis/suitesparse.html.

[2] SuperLU_MT Threaded Sparse Matrix Factorization Library. http://crd-legacy.lbl.gov/ xiaoye/-SuperLU/.

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

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

[5] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems. *SIAM J. Sci. Comput.*, 15:1467–1488, 1994.

[6] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Consistent Initial Condition Calculation for Differential-Algebraic Systems. *SIAM J. Sci. Comput.*, 19:1495–1512, 1998.

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

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

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

[10] Y. Cao, S. Li, L. R. Petzold, and R. Serban. Adjoint Sensitivity Analysis for Differential-Algebraic Equations: The Adjoint DAE System and its Numerical Solution. *SIAM J. Sci. Comput.*, 24(3):1076–1089, 2003.

[11] M. Caracotsios and W. E. Stewart. Sensitivity Analysis of Initial Value Problems with Mixed ODEs and Algebraic Equations. *Computers and Chemical Engineering*, 9:359–365, 1985.

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

[13] A. M. Collier, A. C. Hindmarsh, R. Serban, and C.S. Woodward. User Documentation for KINSOL v2.7.0. Technical Report UCRL-SM-208116, LLNL, 2011.

[14] T. A. Davis and P. N. Ekanathan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3), 2010.

[15] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[16] W. F. Feehery, J. E. Tolsma, and P. I. Barton. Efficient Sensitivity Analysis of Large-Scale Differential-Algebraic Systems. *Applied Numer. Math.*, 25(1):41–54, 1997.

[17] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.

[18] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.

[19] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS, suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, (31):363–396, 2005.

[20] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v2.7.0. Technical Report UCRL-SM-208108, LLNL, 2011.

[21] A. C. Hindmarsh and R. Serban. User Documentation for CVODES v2.6.0. Technical report, LLNL, 2011. UCRL-SM-208111.

[22] A. C. Hindmarsh, R. Serban, and A. Collier. User Documentation for IDA v2.7.0. Technical Report UCRL-SM-208112, LLNL, 2011.

[23] 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.

[24] S. Li, L. R. Petzold, and W. Zhu. Sensitivity Analysis of Differential-Algebraic Equations: A Comparison of Methods on a Special Problem. *Applied Num. Math.*, 32:161–174, 2000.

[25] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[26] T. Maly and L. R. Petzold. Numerical Methods and Software for Sensitivity Analysis of Differential-Algebraic Systems. *Applied Numerical Mathematics*, 20:57–79, 1997.

[27] D.B. Ozyurt and P.I. Barton. Cheap second order directional derivatives of stiff ODE embedded functionals. *SIAM J. of Sci. Comp.*, 26(5):1725–1743, 2005.

[28] 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.

[29] R. Serban and A. C. Hindmarsh. CVODES, the sensitivity-enabled ODE solver in SUNDIALS. In *Proceedings of the 5th International Conference on Multibody Systems, Nonlinear Dynamics and Control*, Long Beach, CA, 2005. ASME.

[30] R. Serban and A. C. Hindmarsh. Example Programs for IDAS v1.1.0. Technical Report LLNL-TR-437091, LLNL, 2011.

[31] H. A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.

Index