User Documentation for IDAS v1.0.0

Radu Serban and Cosmin Petra
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

August 22, 2007

UCRL-SM-xxxxxx

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

Contents

List of Tables

List of Figures

Chapter 1

Introduction

IDAS is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [15]. This suite consists of CVODE, KINSOL, and IDAS, and variants of these with sensitivity analysis capabilities.

IDAS is a general purpose solver for the initial value problem for systems of differential-algebraic equations (DAEs). The name IDAS stands for Implicit Differential-Algebraic solver with Sensitivity capabilities. IDAS is based on DASPK [3, 4], but is written in ANSI-standard C rather than FORTRAN77. Its most notable feature is that, in the solution of the underlying nonlinear system at each time step, it offers a choice of Newton/direct methods and a choice of Inexact Newton/Krylov (iterative) methods. Thus IDAS shares significant modules previously written within CASC at LLNL to support the ordinary differential equation (ODE) solvers CVODE [17, 10] and PVODE [6, 7], and also the nonlinear system solver KINSOL [11].

The Newton/Krylov methods in IDAS are: the GMRES (Generalized Minimal RESidual) [21], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [22], and TFQMR (Transpose-Free Quasi-Minimal Residual) linear iterative methods [13]. 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 sensititivity 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 is apparent. Second, the pointer, structure, and dynamic memory allocation features in C are extremely useful in software of this complexity, with the great variety of method options offered. Finally, we prefer C over C++ for 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.

The structure of this document is as follows:

- In Chapter A we begin with instructions for the installation of IDAS, within the structure of SUNDIALS.
- 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 an overview of the mathematical aspects of sensitivity analysis, both forward (§2.2) and adjoint (§2.3), and conclude with short descriptions of preconditioning (§2.4) and rootfinding (§2.5).
- 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 two NVECTOR implementations provided with SUNDIALS: a serial implementation (§7.1) and a parallel implementation based on MPI (§7.2).
- 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.

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 [4]. For this, the user must identify sub-vectors of y (not necessarily contiguous), denoted y_d and y_a, which are its differential and algebraic parts, respectively, such that F depends on $\dot{y}_d$ 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 risk 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 [1]. The method order ranges from 1 to 5, with the BDF of order q given by the multistep formula

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

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. The linear systems are solved by one of five methods:

- dense direct solver (serial version only),
- band direct solver (serial version only),
- diagonal approximate Jacobian solver,
- SPGMR = scaled preconditioned GMRES (Generalized Minimal Residual method) with restarts allowed,
- SPBCG = scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method), or
- SPTFQMR = scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method).

For the SPGMR, SPBCG, and SPTFQMR cases, preconditioning is allowed only on the left (see §2.4). Note that the direct linear solvers (dense and band) can only be used with serial vector representations.

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

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

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 or banded), 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 $R \leq 0.9$. The user has the option of changing the constant in the convergence test from its default value of 0.33. The quantity S is set to $S = 20$ initially and whenever J or P is updated, and it is reset to $S = 100$ on a step with $\alpha \neq \bar{\alpha}$. Note that at $m = 1$, the convergence test (2.8) 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

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

where U is the unit roundoff, h is the current step size, and W_j is the error weight for the component y_j defined by (2.7). 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.

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

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

where $\phi(k)$ is a modified divided difference of order k that is retained by IDAS (and behaves asymptotically as h^k). Thus the local truncation errors are estimated as $\mathrm{ELTE}(q') = C(q')\|\phi(q'+1)\|$ 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

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

The step/order selection begins with a test for monotonicity that is made even *before* the local error test is performed. Namely, the order is reset to $q' = q-1$ if (a) $q=2$ and $T(1) \le T(2)/2$, or (b) $q>2$ and $\max\{T(q-1), T(q-2)\} \le T(q)$; 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 $\mathrm{ELTE}(q)$, and, with safety factors, is given by

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

The value of η is adjusted so that $0.25 \le \eta \le 0.9$ 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 $T(q-1) \le \min\{T(q), T(q+1)\}$;
- else reset $q \leftarrow q+1$ if $T(q+1) < T(q)$;
- leave q unchanged otherwise [then $T(q-1) > T(q) \le T(q+1)$].

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 $\eta \le 1$, η is restricted to $0.5 \le \eta \le 0.9$; and (c) if $1 < \eta < 2$ we use $\eta = 1$. Finally h is reset to $h' = \eta h$. Thus we do not increase the step size unless it can be doubled. See [1] for details.

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: $y_i > 0$, $y_i < 0$, $y_i \ge 0$, or $y_i \le 0$. 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 $t = t_{\text{out}}$ is overtaken, and then computes $y(t_{\text{out}})$ by interpolation. However, a "one step" mode option is available, where control returns to the calling program after each step. There are also options to force IDAS not to integrate past a given stopping point $t = t_{\text{stop}}$.

2.2 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 in short *sensitivity equations*):

$$\begin{aligned} &\frac{\partial F}{\partial y} s_i + \frac{\partial F}{\partial \dot{y}} \dot{s}_i + \frac{\partial F}{\partial p_i} = 0 \\ &s_i(t_0) = \frac{\partial y_0(p)}{\partial p_i}\,, \;\; \dot{s}_i(t_0) = \frac{\partial \dot{y}_0(p)}{\partial p_i}\,, \end{aligned} \tag{2.10}$$

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 (2.10), 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 the desired sensitivities ($N_s \leq N_p$). 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.

2.2.1 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 [9], 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 (2.10) 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 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 [19]. 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 [20], the discretization is applied simultaneously to both the original equations (2.2) and the sensitivity systems (2.10) resulting in an "extended" nonlinear system $\hat{G}(\hat{y}_n) = 0$ where $\hat{y}_n = [y, \ldots, s_i, \ldots]$. This combined nonlinear system can be solved

using a modified Newton method as in (2.5) by solving the corrector equation

$$\hat{J}[\hat{y}_{n(m+1)} - \hat{y}_{n(m)}] = -\hat{G}(\hat{y}_{n(m)}) \tag{2.11}$$

at each iteration, where

$$\hat{J} = \begin{bmatrix} J & & & & \\ J_1 & J & & & \\ J_2 & 0 & J & & \\ \vdots & \vdots & \ddots & \ddots & \\ J_{N_s} & 0 & \dots & 0 & J \end{bmatrix},$$

J is defined as in (2.6), and $J_i = (\partial/\partial y)\,[F_y s_i + F_{\dot{y}} \dot{s}_i + F_{p_i}]$. It can be shown that 2-step quadratic convergence can be attained by using only the block-diagonal portion of $\hat{J}$ in the corrector equation (2.11). 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 (2.11) to update the sensitivity portions of the residual $\hat{G}$.

- *Staggered corrector* In this approach [12], 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 (2.10):

$$\begin{aligned} &J[\xi_{n(m+1)} - \xi_{n(m)}] = \\ &-\left[F_y(t_n, y_n, \dot{y}_n)\xi_{n(m)} + F_{\dot{y}}(t_n, y_n, \dot{y}_n) \cdot h_n^{-1} \left(\alpha_{n,0}\xi_{n(m)} + \sum_{i=1}^{q} \alpha_{n,i}\xi_{n-i}\right) + F_{p_i}(t_n, y_n, \dot{y}_n)\right]. \end{aligned} \tag{2.12}$$

  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 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 (2.12) will theoretically converge after one iteration.

2.2.2 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.

2.2.3 Evaluation of the sensitivity right-hand side

There are several methods for evaluating the right-hand side of the sensitivity systems (2.10): 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 (2.10) can be evaluated separately:

$$\frac{\partial F}{\partial y}s_i + \frac{\partial F}{\partial \dot{y}}\dot{s}_i \approx \frac{F(t, y+\sigma_y s_i, \dot{y}+\sigma_y \dot{s}_i, p) - F(t, y-\sigma_y s_i, \dot{y}-\sigma_y \dot{s}_i, p)}{2\,\sigma_y}\,, \tag{2.13}$$

$$\frac{\partial F}{\partial p_i} \approx \frac{F(t, y, \dot{y}, p+\sigma_i e_i) - F(t, y, \dot{y}, p-\sigma_i e_i)}{2\,\sigma_i}\,, \tag{2.13'}$$

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

simultaneously:

$$\frac{\partial F}{\partial y}s_i + \frac{\partial F}{\partial \dot{y}}\dot{s}_i + \frac{\partial F}{\partial p_i} \approx \frac{F(t, y+\sigma s_i, \dot{y}+\sigma \dot{s}_i, p+\sigma e_i) - F(t, y-\sigma s_i, \dot{y}-\sigma \dot{s}_i, p-\sigma e_i)}{2\,\sigma}\,, \tag{2.14}$$

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

or by adaptively switching between (2.13)+(2.13') and (2.14), depending on the relative size of the estimated finite difference increments σ_i and σ_y.

These procedures for choosing the perturbations (δ_i, δ_y, δ) and switching ($\rho_{\max}$) between finite difference and directional derivative formulas have also been implemented for first-order 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 (2.14) 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.

2.3 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

$$G(p) = \int_{t_0}^{T} g(t, y, p)dt\,, \tag{2.15}$$

or, alternatively, the gradient dg/dp of the function $g(t, y, p)$ at 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 [8].

2.3.1 Sensitivity of $G(p)$

We focus first on solving the sensitivity problem for $G(p)$ defined by (2.15). Introducing a Lagrange multiplier λ, we form the augmented objective function

$$I(p) = G(p) - \int_0^T \lambda^* F(t, y, \dot{y}, p) dt.$$

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

$$\frac{dG}{dp} = \frac{dI}{dp} = \int_0^T (g_p + g_y y_p) dt - \int_0^T \lambda^* (F_p + F_y y_p + F_{\dot{y}} \dot{y}_p) dt, \tag{2.16}$$

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

$$\int_0^T \lambda^* F_{\dot{y}} \dot{y}_p dt = (\lambda^* F_{\dot{y}} y_p)|_0^T - \int_0^T (\lambda^* F_{\dot{y}})' y_p dt.$$

Thus equation (2.16) becomes

$$\frac{dG}{dp} = \int_0^T (g_p - \lambda^* F_p)\, dt - \int_0^T \left[-g_y + \lambda^* F_y - (\lambda^* F_{\dot{y}})'\right] y_p dt - (\lambda^* F_{\dot{y}} y_p)|_0^T. \tag{2.17}$$

Now letting

$$(\lambda^* F_{\dot{y}})' - \lambda^* F_y = -g_y \tag{2.18}$$

we obtain

$$\frac{dG}{dp} = \int_0^T (g_p - \lambda^* F_p)\, dt - (\lambda^* F_{\dot{y}} y_p)|_0^T. \tag{2.19}$$

Note that y_p at $t = 0$ 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

$$\lambda^* F_{\dot{y}}|_{t=T} = 0, \tag{2.20}$$

yielding the sensitivity equation for dG/dp

$$\frac{dG}{dp} = \int_0^T (g_p - \lambda^* F_p)\, dt + (\lambda^* F_{\dot{y}} y_p)|_{t=0}. \tag{2.21}$$

This choice will not suffice for a Hessenberg index-2 DAE system. For derivation of proper final conditions in such cases, see [8].

The first thing to notice about the adjoint system (2.18) is that there is no explicit specification of the parameters p; this implies that, once the solution λ is found, the formula (2.19) 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 (2.18) 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 (2.18). The approach adopted in IDAS, based on *checkpointing*, is described in §2.3.3 below.

2.3.2 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 (2.19), we have

$$\frac{dg}{dp} = (g_p - \lambda^* F_p)(T) - \int_0^T \lambda_T^* F_p dt + (\lambda_T^* F_{\dot{y}} y_p)|_{t=0} - \frac{d(\lambda^* F_{\dot{y}} y_p)}{dT} \tag{2.22}$$

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

$$(\lambda_T^* F_{\dot{y}})' - \lambda_T^* F_y = 0. \tag{2.23}$$

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

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

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 (2.18) yield

$$(\lambda_T^* F_{\dot{y}})|_{t=T} = [g_y - \lambda^* F_y] \, |_{t=T}. \tag{2.24}$$

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 (2.18), the above selection for $\lambda_T(T)$ is not sufficient for index-two Hessenberg DAEs (see [8] for details).

2.3.3 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.

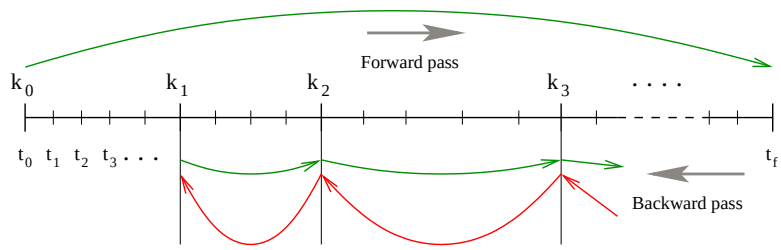


Figure 2.1: Illustration of the checkpointing algorithm for generation of the forward solution during the integration of the adjoint system.

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 from t_1 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 N_d the 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 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 (2.15).

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 (2.18) or (2.23), as well as any other quadrature ODEs that may be needed in evaluating the integrals in (2.19). 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.

[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.

2.4 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 [2] 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 inverse 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.

2.5 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 $g_i(t, y, y')$ that depend on t, the solution vector $y = y(t)$, and its $t-$derivative $y'(t)$. 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 $g_i(t, y(t), y'(t))$, 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 [14]. 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 $\tau/2$.

Chapter 3

Code Organization

3.1 SUNDIALS organization

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

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

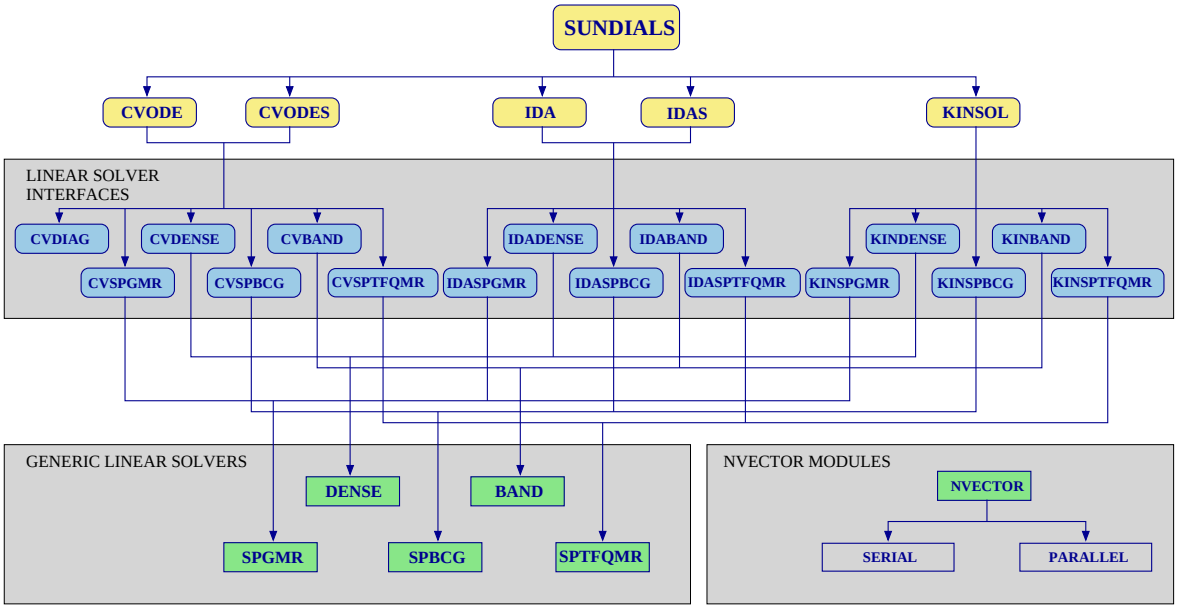
3.2 IDAS organization

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

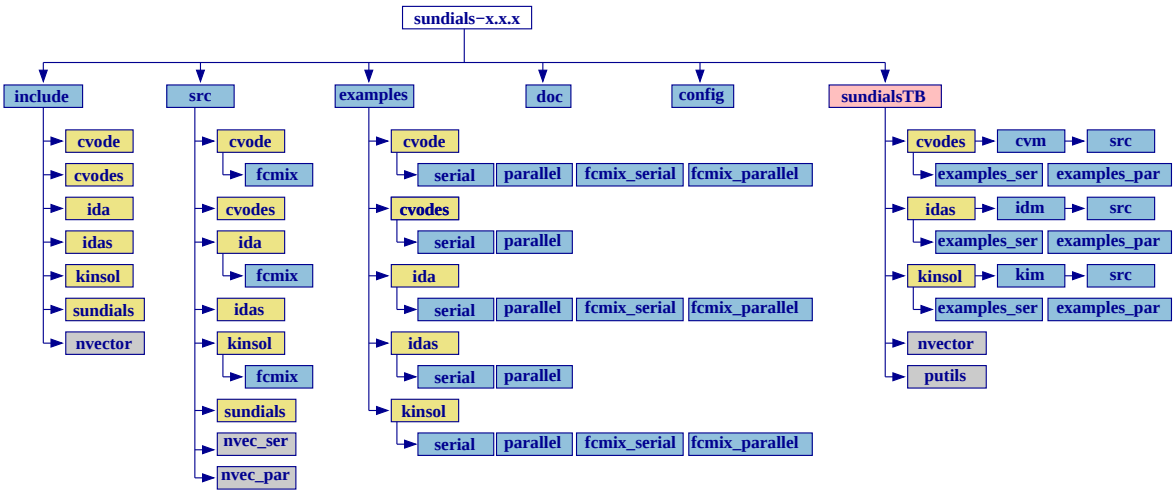
The overall organization of the IDA package is shown in Figure 3.2. The central integration module, implemented in the files `ida.h`, `ida_impl.h`, and `ida.c`, deals with the evaluation of integration coefficients, the Newton iteration process, estimation of local error, selection of stepsize and order, and interpolation to user output points, among other issues. Although this module contains logic for the basic Newton iteration algorithm, it has no knowledge of the method being used to solve the linear systems that arise. For any given user problem, one of the linear system modules is specified, and is then invoked as needed during the integration.

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

- IDADENSE: LU factorization and backsolving with dense matrices;
- IDABAND: LU factorization and backsolving with banded matrices;
- IDASPGMR: scaled preconditioned GMRES method;
- IDASPBCG: scaled preconditioned Bi-CGStab method;
- IDASPTFQMR: scaled preconditioned TFQMR method.



(a) High-level diagram



(b) Directory structure of the source tree

Figure 3.1: Organization of the SUNDIALS suite

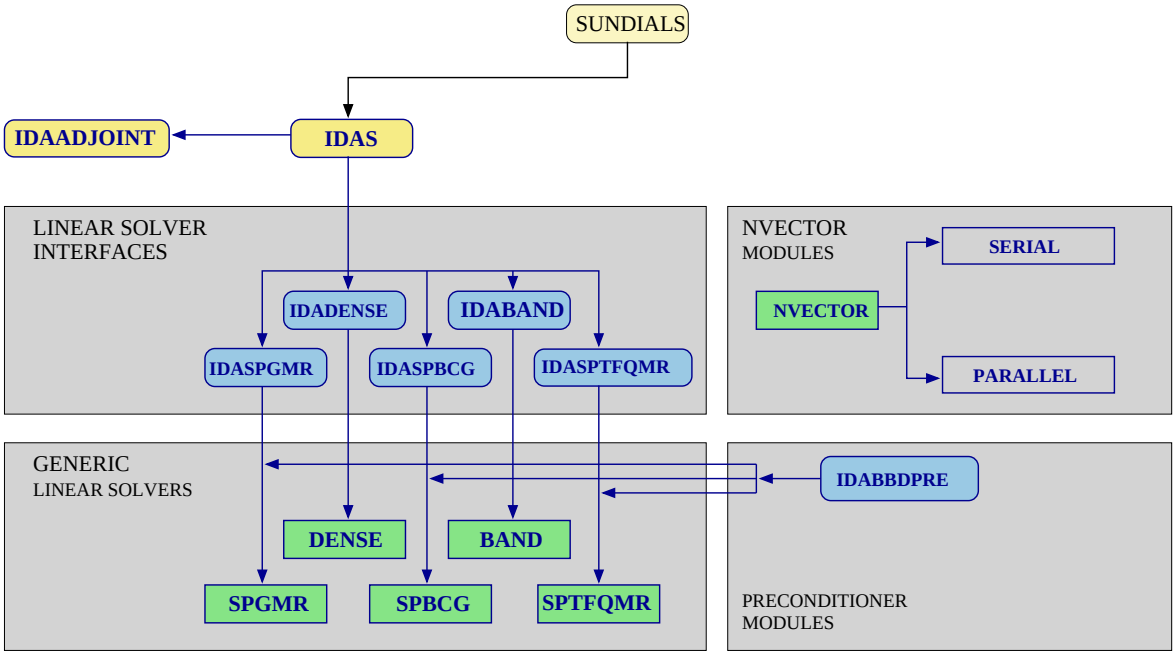


Figure 3.2: Overall structure diagram of the IDA package. Modules specific to IDA are distinguished by rounded boxes, while generic solver and auxiliary modules are in square boxes.

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

In the case of the direct methods 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. 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 [2, 5], together with the example and demonstration programs included with IDA, offer considerable assistance in building preconditioners.

Each IDA linear solver module consists of five routines, devoted to (1) memory allocation and initialization, (2) setup of the matrix data involved, (3) solution of the system, (4) monitoring performance, and (5) freeing of memory. The setup and solution phases are separate because the evaluation of Jacobians and preconditioners is done only periodically during the integration, as required to achieve convergence. The call list within the central IDA module to each of the five associated functions is fixed, thus allowing the central module to be completely independent of the linear system method.

These modules are also decomposed in another way. Each of the modules IDADENSE, IDABAND, IDASPGMR, IDASPBCG, and IDASPTFQMR is a set of interface routines built on top of a generic solver module, named DENSE, BAND, SPGMR, SPBCG, and SPTFQMR, respectively. The interfaces deal with the use of these methods in the IDA context, whereas the generic solver is independent of the context. While the generic solvers here were generated with SUNDIALS in mind, our intention is that they be usable in other applications as general-purpose solvers. This separation also allows for any generic solver to be replaced by an improved version, with no necessity to revise the IDA package elsewhere.

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

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

Chapter 4

Using IDAS for C Applications

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. The listings of the sample programs in the companion document [16] 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 or direct band linear solvers, since these linear solver modules need to form the complete system Jacobian. The IDADENSE and IDABAND modules (using either the internal implementation or Lapack) can only be used with NVECTOR_SERIAL. 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 Chapter 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 Chapter 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 or two files),

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

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

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

4.2 Data types

The sundials_types.h file contains the definition of the type realtype, which is used by the SUNDIALS solvers for all floating-point data. The type realtype can be float, double, or long double, with the default being double. The user can change the precision of the SUNDIALS solvers arithmetic at the configuration stage (see §A.1.1).

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

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

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

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

A user program which uses the type realtype and the RCONST macro to handle floating-point constants is precision-independent except for any calls to precision-specific standard math library functions. (Our example programs use both realtype and RCONST.) Users can, however, use the type double, float, or long double in their code (assuming the typedef for realtype matches this choice). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use realtype, so long as the SUNDIALS libraries use the correct precision (for details see §A.1.1).

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 two 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.

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_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 `idas_direct.h` which defines common functions and which in turn includes a header 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 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.

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

1. **[P] Initialize MPI**

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

2. **Set problem dimensions**

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

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

3. **Set vectors of initial values**

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

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

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

   Otherwise, make the calls:

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

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

   and load initial values into the structure defined by:

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

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

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

The initial conditions for y' are set similarly.

4. **Create 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 a scalar relative tolerance and scalar absolute tolerance or scalar relative tolerance and a vector of absolute tolerances, respectively. 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**] `flag = IDADense(...);`

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

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

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

   `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 §4.5.7.3 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 §4.5.5. Also see §4.5.7.2 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 §4.5.4 for details.

12. **Advance solution in time**

    For each point at which output is desired, call `flag = IDASolve(ida_mem, tout, &tret, yret, ypret, itask);` Set `itask` to specify the return mode. The vector `yret` (which can be the same

as the vector `y0` above) will contain $y(t)$, while the vector `ypret` will contain $y'(t)$. See §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);`

    [**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.

TODO: say something about error handling (i.e. sending error messages to the handler)

On error, each user-callable function returns a negative value and sends an error message to the error handler routine, which prints the message on standard error by default. However, the user can set a file as error output or can provide his own error handler function (see §4.6).

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 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 F in the DAE. This function has the form `res(t, yy, yp, resval, user_data)` (for full details see §4.6).

`t0` (`realtype`) is the initial value of t.

`y0` (`N_Vector`) is the initial value of y.

`yp0` (`N_Vector`) is the initial value of y'.

Return value The return flag `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 `IDAInit`.

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 the weigths 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 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_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 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_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 DAE.

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 flag 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 idasdenx in the IDAS package, and the discussion of it in the IDAS Examples document [16]. In that problem, the three components vary betwen 0 and 1, and have different noise levels; hence the abstol vector. It is impossible to give any general advice on abstol values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.

(3) Finally, it is important to pick all the tolerance values conservately, because they control the error committed on each individual time step. The final (global) errors are some sort of accumulation of those per-step errors. A good rule of thumb is to reduce the tolerances by a factor of .01 from the actual desired limits on errors. So if you want .01% accuracy (globally), a good choice is reltol= 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 $F(t, y)$.

(4) IDAS provides the option of enforcing positivity or non-negativity on components. But these constraint options should only be exercised if the use of absolute tolerances to control the computed values is unsuccessful, because they involve some extra overhead cost.

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 five IDAS linear solvers currently available for this task: IDADENSE, IDABAND, IDASPGMR, IDASPBCG, and IDASPTFQMR.

The first two are direct solvers and derive their name from the type of approximation used for the Jacobian $J = \partial F/\partial y + c_j \partial F/\partial y'$. IDADENSE and IDABAND work with dense and banded approximations to J, respectively. The SUNDIALS suite includes both internal implementations of these two linear solvers and interfaces to Lapack implementations. Together, these linear solvers are refered to as IDADLS (from direct 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 refered 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`, `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 (with the exception of the Lapack linear solvers), 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, 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.
	The user's main function 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

IDADIRECT_SUCCESS The IDADENSE initialization was successful.

IDADIRECT_MEM_NULL The ida_mem pointer is NULL.

IDADIRECT_ILL_INPUT The IDADENSE solver is not compatible with the current NVECTOR module.

IDADIRECT_MEM_FAIL A memory allocation request failed.

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

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 function must include the idas_lapack.h header file.

Arguments The input arguments are identical to those of IDADense.

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

IDABand

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

Description The function IDABand selects the IDABAND linear solver.

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

`IDABand`

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 function must include the `idas_lapack.h` header file.

Arguments The input arguments are identical to those of `IDABand`.

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

`IDASpgmr`

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

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

The user's main function 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_SPGMR_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 function 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_SPBCG_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 function 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_SPTFQMR_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 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 root finding algorithm, call the following function:

`IDARootInit`

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

Description The function `IDARootInit` specifies that the roots of a set of functions $g_i(t, y, y')$ 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 $g_i(t, y, y')$ 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`$= 0$.

4.5.5 Initial condition calculation function

`IDACalcIC` calculates corrected initial conditions for the DAE system for a class of index-one problems of semi-implicit form. (See §2.1 and Ref. [4].) It uses Newton iteration combined with a linesearch algorithm. Calling `IDACalcIC` is optional. It is only necessary when the initial conditions do not solve the given system. Thus if `y0` and `yp0` are known to satisfy $F(t_0, y_0, y_0') = 0$, then a call to `IDACalcIC` is generally *not* necessary.

A call to `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 y', given the differential components of y. This option requires that the `N_Vector id` was set through `IDASetId`, specifying the differential and algebraic components.

  `icopt=IDA_Y_INIT` directs `IDACalcIC` to compute all components of y, given y'. 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 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 $y(t_0)$ and $y'(t_0)$ which were specified in the previous call to IDAInit or IDAReInit. To obtain the corrected values, call IDAGetconsistentIC (see §4.5.9.2).

4.5.6 IDAS solver function

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

IDASolve

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

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

Arguments

ida_mem	(void *) pointer to the IDAS memory block.
tout	(realtype) the next time at which a computed solution is desired.
tret	(realtype *) the time reached by the solver.
yret	(N_Vector) the computed solution vector y.
ypret	(N_Vector) the computed solution vector y'.
itask	(int) a flag indicating the job of the solver for the next user step. The IDA_NORMAL task is to have the solver take internal steps until it has reached or just passed the user specified tout parameter. The solver then interpolates in order to return approximate values of y(tout) and y'(tout). The IDA_ONE_STEP option tells the solver to just take one internal step and return the solution at the point reached by that step. The IDA_NORMAL_TSTOP and IDA_ONE_STEP_TSTOP modes are similar to IDA_NORMAL and IDA_ONE_STEP, respectively, except that the integration never proceeds past the value tstop, specified through the function IDASetStopTime (see §4.5.7.1).

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

In IDA_NORMAL mode with no errors, *tret will be equal to tout and yret $= y$(tout), ypret $= y'$(tout).

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

IDA_SUCCESS	IDASolve succeeded.
IDA_TSTOP_RETURN	IDASolve succeeded by reaching the stop point specified through the optional input function IDASetStopTime.
IDA_ROOT_RETURN	IDASolve succeeded and found one or more roots. 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 is illegal. This includes the situation where a root of one of the root functions was found both at a point t and also very near t. It also includes the situation when a component of the error weight vectors becomes negative during internal time-stepping. The IDA_ILL_INPUT flag will also be returned if the linear solver function initialization (called by the user after calling IDACreate) failed to set the linear solver-specific lsolve field in ida_mem. In any case, the user should see the printed error message for more details.
IDA_TOO_MUCH_WORK	The solver took mxstep internal steps but could not reach tout. The default value for mxstep is MXSTEP_DEFAULT = 500.
IDA_TOO_MUCH_ACC	The solver could not satisfy the accuracy demanded by the user for some internal step.
IDA_ERR_FAIL	Error test failures occurred too many times (MXNEF = 10) during one internal time step or occurred with $\|h\| = h_{min}$.
IDA_CONV_FAIL	Convergence test failures occurred too many times (MXNCF = 10) during one internal time step or occurred with $\|h\| = h_{min}$.
IDA_LINIT_FAIL	The linear solver's initialization function failed.
IDA_LSETUP_FAIL	The linear solver's setup function failed in an unrecoverable manner.
IDA_LSOLVE_FAIL	The linear solver's solve function failed in an unrecoverable manner.
IDA_CONSTR_FAIL	The inequality constraints were violated and the solver was unable to recover.
IDA_REP_RES_ERR	The user's residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.
IDA_RES_FAIL	The user's residual function returned a nonrecoverable error flag.
IDA_RTFUNC_FAIL	The rootfinding function failed.

Notes

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

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

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

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

IDAS provides an extensive list of functions that can be used to change various optional input parameters that control the behavior of the IDAS solver 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 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 `IDASetErrHandlerFn` or `IDASetErrFile` are to be called, that call should be first, in order to take effect for any later error message.

`IDASetErrHandlerFn`

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

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 C error handler function (see §4.6.2).

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

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

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

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

`IDASetErrFile`

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

Description The function `IDASetErrFile` specifies the pointer to the file where all IDAS messages should be directed in case 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`).

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.

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

Optional input	Function name	Default
IDA main solver		
Error handler function	`IDASetErrHandlerFn`	internal fn.
Pointer to an error file	`IDASetErrFile`	`stderr`
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`
IDA initial conditions calculation		
Coeff. in the nonlinear convergence test	`IDASetNonlinConvCoefIC`	0.0033
Maximum no. of steps	`IDASetMaxNumStepsIC`	5
Maximum no. of Jacobian/precond. evals.	`IDASetMaxNumJacsIC`	4
Maximum no. of Newton iterations	`IDASetMaxNumItersIC`	10
Turn off linesearch	`IDASetLineSearchOffIC`	`FALSE`
Lower bound on Newton step	`IDASetStepToleranceIC`	$\mathrm{uround}^{2/3}$
IDADLS linear solvers		
Dense Jacobian function	`IDADlsSetDenseJacFn`	DQ
Band Jacobian function	`IDADlsSetBandJacFn`	DQ
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

`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.

`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.

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

`IDA_SUCCESS` The optional value has been successfully set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT` The specified value `maxord` is negative, or larger than its previous value.

Notes The default value is 5. Since `maxord` affects the memory requirements for the internal IDAS memory block, its value can not be increased past its previous value.

`IDASetMaxNumSteps`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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`.
`IDA_ILL_INPUT` `mxsteps` is non-positive.

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

`IDASetInitStep`

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

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

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

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

`IDA_SUCCESS` The optional value has been successfully set.

IDA_MEM_NULL The ida_mem pointer is NULL.

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

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.

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.

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.

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.

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.

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

Notes The default value is 0.33.

`IDASetSuppressAlg`

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

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

Arguments `ida_mem` (`void *`) pointer to the IDAS memory block.
`suppresslag` (`booleantype`) indicates whether to suppress (`TRUE`) or not (`FALSE`) the algebraic variables in the local error test.

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

Notes The default value is `FALSE`.

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

`IDASetId`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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 §4.5.5).

`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 successfuly set.

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

`IDA_ILL_INPUT` The constraints vector contains illegal values.

Notes The presence of a non-`NULL` constraints vector that is not 0.0 in all components will cause constraint checking to be performed.

4.5.7.2 Initial condition calculation optional input functions

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

`IDASetNonlinConvCoefIC`

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

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

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

`epiccon` (`realtype`) coefficient in the Newton convergence test.

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

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

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

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

`IDASetMaxNumStepsIC`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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 successfuly set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT` `maxnh` is non-positive.

Notes The default value is 5.

`IDASetMaxNumJacsIC`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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 successfuly set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT` `maxnj` is non-positive.

Notes The default value is 4.

`IDASetMaxNumItersIC`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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 successfuly set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.
`IDA_ILL_INPUT` `maxnit` is non-positive.

Notes The default value is 10.

`IDASetLineSearchOffIC`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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 successfuly set.

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

Notes The default value is `FALSE`.

`IDASetStepToleranceIC`

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

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

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

`steptol` (`int`) Newton step tolerance.

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

`IDA_SUCCESS` The optional value has been successfuly set.

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

`IDA_ILL_INPUT` The `steptol` tolerance is negative (illegal).

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

4.5.7.3 Direct linear solvers optional input functions

The IDADENSE solver needs a function to compute a dense approximation to the Jacobian matrix $J(t, y, y')$. This function must be of type `IDADlsDenseJacFn`. The user can supply his/her own dense Jacobian function, or use the default difference quotient approximation function 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 its dense Jacobian function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `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

`IDADIRECT_SUCCESS` The optional value has been successfully set.

`IDADIRECT_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDADIRECT_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 $J(t, y, y')$. 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 its banded Jacobian approximation function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer 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

IDADIRECT_SUCCESS	The optional value has been successfully set.
IDADIRECT_MEM_NULL	The ida_mem pointer is NULL.
IDADIRECT_LMEM_NULL	The IDABAND linear solver has not been initialized.

Notes By default, IDABAND uses the 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.

4.5.7.4 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 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 difference quotient function IDASpilsDQJtimes 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 §4.6.7 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 and the pointer to user data.

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

`psetup` (`IDASpilsPrecSetupFn`) user-defined preconditioner setup function.

`psolve` (`IDASpilsPrecSolveFn`) 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` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.

Notes The function type `IDASpilsPrecSolveFn` is described in §4.6.8. The function type `IDASpilsPrecSetupFn` is described in §4.6.9.

`IDASpilsSetJacTimesVecFn`

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

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

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 `IDASpilsDQJtimes`. If `NULL` is passed to `jtimes`, this default function is used.

The function type `IDASpilsJacTimesVecFn` is described in §4.6.7.

`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 successfuly 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 Gram-Schmidt orthogonalization type `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 successfuly 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 GMRES convergence test constant is reduced from the Newton iteration test constant. (See §2).

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

`eplifac` (`realtype`)

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

`IDASPILS_SUCCESS`	The optional value has been successfuly 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 factor `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).

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 successfuly 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.

`IDASpbcgSetMaxl`

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

Description The function `IDASpbcgSetMaxl` specifies maximum of the Krylov subspace dimension for the Bi-CGStab method.

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 successfuly set.
`IDASPILS_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPILS linear solver has not been initialized.

Notes The default value is 5. Passing `maxl` = 0 also results in the default value.

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

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 kth 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 $t_n - h_u \leq t \leq t_n$, where t_n denotes the current internal time reached, and h_u is the last internal step size used successfully.

Arguments `ida_mem` (`void *`) pointer to the 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_DKY`	`k` is not one of the $\{0, 1, \ldots, kused\}$.

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

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.

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

Optional output	Function name
IDA 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`
IDA initial conditions calculation	
Number of backtrack operations	`IDAGatNumBacktrackops`
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`
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`

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 successfuly 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 §4.5.4), the actual size of the real workspace, in `realtype` words, is given by the following:

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

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

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

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

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

For the default value of `maxord`, with no rootfinding, no `id`, no constraints, and with no call to `IDASVtolerances`, these lengths are given roughly by: `lenrw` $= 55 + 11N$, `leniw` $= 38$.

`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 successfuly set.

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

`IDAGetNumResEvals`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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 successfuly set.

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

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

`IDAGetNumLinSolvSetups`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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 successfuly set.

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

`IDAGetNumErrTestFails`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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 successfuly 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 successfuly 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 successfuly set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetLastStep`

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

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

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

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

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

`IDAGetCurrentStep`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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 successfuly set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetActualInitStep`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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 successfuly set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

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

`IDAGetCurrentTime`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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 successfuly set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetTolScaleFactor`

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

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

Arguments `ida_mem` (`void *`) pointer to the 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 successfuly set.
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetErrWeights`

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

Description The function `IDAGetErrWeights` returns the solution error weights at the current time. These are the 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 successfuly 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.
`eweight` (`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 successfuly 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 positive 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 the two vectors. Thus, for example, if there were recent error test failures, the components causing the failures are those with largest values for the products, denoted loosely as `eweight[i]*ele[i]`.

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

`IDAGetNumNonlinSolvIters`

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

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

Arguments
- `ida_mem` (`void *`) pointer to the 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 successfuly set.
- `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetNumNonlinSolvConvFails`

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

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

Arguments
- `ida_mem` (`void *`) pointer to the 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 successfuly set.
- `IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDAGetNonlinSolvStats`

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

Description The function `IDAGetNonlinSolvStats` returns the 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 successfuly 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 a 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 successfuly 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 successfuly set.
- `IDA_ILL_INPUT` The fucntion 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`$[i] = 1$ 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 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 Direct linear solvers optional output functions

The following optional outputs are available from the IDADLS module: workspace requirements, number of calls to the Jacobian routine, number of calls to the residual routine for finite-difference Jacobian approximation, and last return value from 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 IDADENSE real and integer workspaces.

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

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

`IDADIRECT_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDADIRECT_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 the problem size 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

`IDADIRECT_SUCCESS`	The optional output value has been successfuly set.
`IDADIRECT_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDADIRECT_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

`IDADIRECT_SUCCESS`	The optional output value has been successfuly set.
`IDADIRECT_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDADIRECT_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, &flag);`

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

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

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

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

`IDADIRECT_SUCCESS`	The optional output value has been successfully set.
`IDADIRECT_MEM_NULL`	The `ida_mem` pointer is `NULL`.
`IDADIRECT_LMEM_NULL`	The IDADENSE linear solver has not been initialized.

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

`IDADlsGetReturnFlagName`

Call `name = IDADlsGetReturnFlagName(flag);`

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

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

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

4.5.9.5 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 IDASPGMR 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 successfuly 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 successfuly 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 successfuly 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 successfuly 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 successfuly 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 successfuly 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 successfuly 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, &flag);`

Description The function `IDASpilsGetLastFlag` returns the last return value from an IDASPILS routine.

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

Description The function `IDASpilsGetReturnFlagName` returns the name of the IDASPILS constant corresponding to `flag`.

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

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 `Set` calls, as described in §4.5.3.

`IDAReInit`

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

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

Arguments
- `ida_mem` (`void *`) pointer to the IDAS memory block.
- `res` (`IDAResFn`) is the C function which computes F. This function has the form `f(t, y, yp, r, user_data)` (for full details see §4.6).
- `t0` (`realtype`) is the initial value of t.
- `y0` (`N_Vector`) is the initial value of y.
- `yp0` (`N_Vector`) is the initial value of y'.

Return value The return flag `flag` (of type `int`) will be one of the following:
- `IDA_SUCCESS` The call to `IDAReInit` was successful.
- `IDA_MEM_NULL` The 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 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 y'.

Arguments
- `tt` is the current value of the independent variable.
- `yy` is the current value of the dependent variable vector, $y(t)$.
- `yp` is the current value of $y'(t)$.
- `rr` is the output residual vector $F(t, y, y')$.
- `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 occured (e.g. `yy` has an illegal value), or a negative value if a nonrecoverable error occured.

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

Notes Allocation of memory for `yp` is handled within 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 *user_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.
- `user_data` is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.

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 returning a pointer to memory (e.g. `IDABBDPrecAlloc`) encounters an error, it sets `error_code` to 0 before returning `NULL`.

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 vector for which the WRMS norm must 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 successfuly set the error weights and -1 otherwise. In case of failure, a message is printed and the integration stops.

Notes Allocation of memory for `ewt` is handled within 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 ODE 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 $g(t, y, y')$ such that the roots of the `nrtfn` components $g_i(t, y, y')$ 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 $y'(t)$, the $t-$derivative of y.
- `gout` is the output array, of length `nrtfn`, with components $g_i(t, y, y')$.
- `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 occured (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. `IDADense` 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 c_j,
                                N_Vector yy, N_Vector yp, N_Vector rr,
                                void *user_data, DlsMat Jac,
                                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.
- `c_j` 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 $y'(t)$.
- `rr` is the current value of the residual vector $F(t, y, y')$.

`user_data` is a pointer to user data — the same as the `user_data` parameter passed to `IDASetUserData`.

`Jac` is the output Jacobian matrix.

`tmp1`
`tmp2`
`tmp3` are pointers to memory allocated for variables of type `N_Vector` which can be used by `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 occured, or a negative value if a nonrecoverable error occured.

In the case of a recoverable eror return, the integrator will attempt to recover by reducing the stepsize, and hence changing α in (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 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, use the `IDAGet*` functions described in §4.5.9.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

4.6.6 Jacobian information (direct method with banded Jacobian)

If the direct linear solver with banded treatment of the Jacobian is used (i.e. `IDABand` is called in Step 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,
                               N_Vector yy, N_Vector yp, N_Vector rr,
                               realtype c_j, void *user_data, DlsMat Jac,
                               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.

`mlower`
`mupper` are the lower and upper half bandwidth of the Jacobian.

tt	is the current value of the independent variable.
yy	is the current value of the dependent variable vector, $y(t)$.
yp	is the current value of $y'(t)$.
rr	is the current value of the residual vector $F(t, y, y')$.
c_j	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.
Jac	is the output Jacobian matrix.
tmp1	
tmp2	
tmp3	are pointers to memory allocated for variables of type N_Vector which can be used by 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 occured, or a negative value if a nonrecoverable error occured.

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

Notes A user-supplied band Jacobian function must load the band matrix Jac of type DlsMat with the elements of the Jacobian $J(t, y, y')$ 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 jth column of Jac, and if we assign this address to realtype *col_j, then the ith element of the jth column is given by BAND_COL_ELEM(col_j, i, j), counting from 0. Thus for (m, n) within the band, $J_{m,n}$ can be loaded by setting col_n = BAND_COL(Jac, n-1); BAND_COL_ELEM(col_n, m-1, n-1) = $J_{m,n}$. The elements of the jth column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type 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 §9.2.

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, use the IDAGet* functions described in §4.5.9.1. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

4.6.7 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 in the following form:

`IDASpilsJacTimesVecFn`

Definition
```
typedef int (*IDASpilsJacTimesVecFn)(realtype tt, N_Vector yy,
                                     N_Vector yp, N_Vector rr,
                                     N_Vector v, N_Vector Jv,
                                     realtype c_j, 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 $y'(t)$.
- `rr` is the current value of the residual vector $F(t, y, y')$.
- `v` is the vector by which the Jacobian must be multiplied to the right.
- `Jv` is the output vector computed.
- `c_j` is the scalar in the system Jacobian, proportional to the inverse of the step size (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.

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, 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 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 $J = \partial F/\partial y + c_j \partial F/\partial y'$. 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 c_j, 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 $y'(t)$.
- `rr` is the current value of the residual vector $F(t, y, y')$.
- `rvec` is the right-hand side vector r of the linear system to be solved.
- `zvec` is the output vector computed.
- `c_j` 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).

4.6.9 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 c_j, 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 $y'(t)$.

`rr` is the current value of the residual vector $F(t, y, y')$.

`c_j` 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, use the `IDAGet*` functions described in §4.5.9.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

foo

4.7 Integration of pure quadrature equations

If the system of DAEs contains *pure quadratures*, it is more efficient to treat them 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`. 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 occured, `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 by:

- 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 by:

- 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 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 occured, `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.

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 tests 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 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, &t, 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`.
`t` (`realtype`) the time reached by the solver.
`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.

Notes In case of an error return, an error message is also sent to the error handler function.

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, 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.
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 $0, 1, ..., kused$.
IDA_BAD_T	The time t is not in the allowed range.

Notes In case of an error return, an error message is also sent to the error handler function.

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 should be used in the step size control mechanism. If so, the user must call 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. This function must be of type `IDAQuadRhsFn` defined as follows:

`IDAQuadRhsFn`

Definition
```
typedef int (*IDAQuadRhsFn)(realtype t, N_Vector y, 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 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 vector, $y'(t)$.
- `rhsQ` is the output vector $f_Q(t, y, y')$.
- `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 `rhsQt` 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 are two situations in which recovery is not possible even if `IDAQuadRhsFn` function returns a recoverable error flag. This include the situation when this occurrs at the very first call to the `IDAQuadRhsFn` (in which case IDAS returns `IDA_FIRST_QRHS_ERR`) or if a recoverable error is reported when `IDAQuadRhsFn` is called after an error test failure, while the linear multistep method order is equal to 1 (in which case IDAS returns `IDA_UNREC_QRHSFUNC_ERR`).

TODO: Fix the last proposition.

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 [18] 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 $G(t, y, y')$ which approximates the

function $F(t, y, y')$ 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 y' into M disjoint blocks y_m and y'_m, and a decomposition of G into blocks G_m. The block G_m depends on y_m and y'_m, and also on components of $y_{m'}$ and $y'_{m'}$ associated with neighboring sub-domains (so-called ghost-cell data). Let $\bar{y}_m$ and $\bar{y'}_m$ denote y_m and y'_m (respectively) augmented with those other components on which G_m depends. Then we have

$$G(t, y, y') = [G_1(t, \bar{y}_1, \bar{y}'_1), G_2(t, \bar{y}_2, \bar{y}'_2), \ldots, G_M(t, \bar{y}_M, \bar{y}'_M)]^T \ , \tag{4.1}$$

and each of the blocks $G_m(t, \bar{y}_m, \bar{y}'_m)$ is uncoupled from the others.

The preconditioner associated with this decomposition has the form

$$P = diag[P_1, P_2, \ldots, P_M] \tag{4.2}$$

where

$$P_m \approx \partial G_m / \partial y_m + \alpha \partial G_m / \partial y'_m \tag{4.3}$$

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 $G(t, y, y') \approx F(t, y, y')$ 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`, and neither function has a return value. 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 is not expected to 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 function computes $G(t, y, y')$. It loads the vector `gval` as a function of `tt`, `yy`, and `yp`.

Arguments `Nlocal` is the local vector length.

`tt`	is the value of the independent variable.
`yy`	is the dependent variable.
`yp`	is the derivative of the dependent variable.
`gval`	is the output vector.
`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 assumes 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 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 comunication was done in `res`, then `Gcomm` = `NULL` can be passed in the call to `IDABBDPrecAlloc` (see below).

Besides the header files required for the integration of the DAE problem (see §4.3), to use the IDABBDPRE module, the main program must include the header file `ida_bbdpre.h` which declares the needed function prototypes.

The following is a summary of the usage of this module and describes the sequence of calls in the user main program. Steps that are unchanged from the user main program presented in §4.4 are grayed-out.

1. **Initialize MPI**
2. **Set problem dimensions**
3. **Set vector of initial values**
4. **Create 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 for and initialize a data structure `bbd_data`, of type `void *`, to be passed to any of the Krylov linear solvers. The last two arguments of `IDABBDPrecAlloc` are the two user-supplied functions described above.

9. **Set linear solver optional inputs**

   Note that the user should not overwrite the preconditioner data, setup function, or solve function through calls to IDASPGMR, IDASPBCG, or IDASPTFQMR 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 – `IDABBDPreconGetWorkSpace` and `IDABBDPreconGetNumGfnEvals`.

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

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

Return value **MORE HRE!!!!**

The return value of `IDABBDPrecReInit` is `IDABBDPRE_SUCCESS` indicating success, or `IDABBDPRE_PDATA_NULL` if `bbd_data` is `NULL`.

Notes If one of the half-bandwidths `mudq` or `mldq` to be used in the difference-quotient calculation of the approximate Jacobian is negative or exceeds the value `Nlocal`-1, it is replaced by 0 or `Nlocal`-1 accordingly.

The half-bandwidths `mudq` and `mldq` need not be the true half-bandwidths of the Jacobian of the local block of G, when smaller values may provide a greater efficiency.

Also, the half-bandwidths `mukeep` and `mlkeep` of the retained banded approximate Jacobian block may be even smaller, to reduce storage and computation costs further.

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

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, Gres, Gcomm);`

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` $= 0.0$.

`Gres` (`IDABBDLocalFn`) the C function which computes the local residual approximation $G(t, y, y')$.

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

Return value **MORE HERE!!!**

The return value of `IDABBDPrecReInit` is `IDABBDPRE_SUCCESS` indicating success, or `IDABBDPRE_PDATA_NULL` if `bbd_data` is `NULL`.

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 **FIX THIS!!!**

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

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

`IDABBDPRE_PDATA_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 **FIX THIS!!!**

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

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

`IDABBDPRE_PDATA_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 §4.5.9.5).

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 (2.10). The only departure from this philosophy is due to the `IDAResFn` type definition (§4.6). 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 Chapter 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 §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 two implementations provided with IDAS: steps marked with [**P**] correspond to NVECTOR_PARALLEL, while steps marked with [**S**] correspond to NVECTOR_SERIAL. Differences between the user main program in §4.4 and the one below start only at step (10).

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. **Define the sensitivity problem**

    - Number of sensitivities (required)

      Set `Ns`, the number of parameters with respect to which sensitivities are to be computed.

    - Problem parameters (optional)

      If IDAS will 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 will evaluate the sensitivity residuals, set `plist`, an array of `Ns` integer flags to specify the parameters `p` with respect to which solution sensitivities are to be computed. If sensitivities with respect to the j-th problem parameter 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 estimates tolerances for the sensitivity solution vectors (based on tolerances for the state solution vector) or if IDAS will 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 $\bar{p}_{\text{plist}_i} = |p_i|$ 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.

11. **Set sensitivity initial conditions**

    To set the sensitivities vectors `yS0` and `ypS0` to initial values use functions defined by a particular NVECTOR implementation.

    For example, for sensitivities vector `yS0`, set the `Ns` vectors `yS0[i]` of `N` initial values for sensitivities (for $i = 0, \ldots, Ns - 1$).

    First, create an array of `Ns` vectors by making the call

    [**S**] `yS0 = N_VNewVectorArray_Serial(Ns, N);`

    [**P**] `yS0 = N_VNewVectorArray_Parallel(Ns, N);`

    and, for each $i = 1, \ldots,$`Ns`, load initial values for the i-th sensitivity vector into the structure defined by:

    [**S**] `NV_DATA_S(yS0[i])`

[**P**] `NV_DATA_P(yS0[i])`

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_VNewVectorArrayEmpty_Serial(Ns, N);`

[**P**] `yS0 = N_VNewVectorArrayEmpty_Parallel(Ns, N);`

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]);`

[**P**] `N_VSetArrayPointer_Parallel(yS0_i, yS0[i]);`

The initial conditions for sensitivities `ypS0` of y' are set similarly.

12. **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).

13. **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.

14. **Advance solution in time**

15. **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`, respectively `ypS0`) by calling one of the following routines: `IDAGetSens`, `IDAGetSens1`, `IDAGetSensDky1` or `IDAGetSensDky1` (see §5.2.5).

16. **Deallocate memory for solutions vector**

17. **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);`

    [**P**] `N_VDestroyVectorArray_Parallel(yS0, Ns);`

    and similarly for `ypS0`.

    If `yS` (or `ypS`) were created from `realtype` arrays `yS_i`, it is the user's responsibility to also free the space for the arrays `yS_i`.

18. **Free user data structure**

19. **Free solver memory**

20. **Free vector specification memory**

5.2 User-callable routines for forward sensitivity analysis

This section describes the IDAS functions, additional to those presented in §4.5, that are called by the user to setup 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 and can be `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;
- `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 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 If an error occured, `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 by:

- Base value: `lenrw` = `lenrw` + (`maxord+5`)$N_s N$
- With `itolS = IDA_SV` (see `IDASetSensTolerances`): `lenrw` = `lenrw` $+N_s N$

the size of the integer workspace is increased by:

- Base value: `leniw` = `leniw` + (`maxord+5`)$N_s N$
- With `itolS = IDA_SV`: `leniw` = `leniw` $+N_s N$

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 and can be `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 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 occured, `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 After a call to `IDASensFree`, forward sensitivity computations can be reactivated only by calling again `IDASensInit`.

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 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 allocation function for sensitivities `IDASensInit` has not been called.
`IDA_ILL_INPUT`	One of the input tolerances was negative.

`IDASVtolerances`

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 allocation function for sensitivities `IDASensInit` 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 DAE.

`IDAEEtolerances`

Call `flag = IDAEEtolerances(ida_mem);`

Description When this function is used, 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 `IDAWFtolerances` 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. Anyhow, more restrictions apply for initial conditions calculation of the state variables, see §4.5.5.

Calling IDACalcIC is optional. It is only necessary when the initial conditions do not solve 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 §4.5.5 for a list of possible return values.

5.2.4 IDASolve 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 y' in y, respectively in y'. 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.
- yS (N_Vector *) the computed forward sensitivity vectors of y.

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	yQ is NULL.

Notes In case of an error return, an error message is also printed.

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 *`) the vectors containing the derivatives. 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` The pointer to `ida_mem` was `NULL`.
- `IDA_NO_SENS` Forward sensitivity analysis was not initialized.
- `IDA_BAD_DKY` One of the vectors `dkyS` is `NULL`.
- `IDA_BAD_K` `k` is not in the range $0, 1, ..., kused$.
- `IDA_BAD_T` The time `t` is not in the allowed range.

Notes In case of an error return, an error message is also printed.

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, t, 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`.
- `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.
- `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` `yQ` is `NULL`.
- `IDA_BAD_T` The time `t` is not in the allowed range.

Notes In case of an error return, an error message is also printed.

`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`.

Table 5.1: Forward sensitivity optional inputs

Optional input	Routine name	Default
Sensitivity scaling factors	IDASetSensParams	NULL
DQ approximation method	IDASetSensDQMethod	0.0
Error control strategy	IDASetSensErrCon	FALSE
Maximum no. of nonlinear iterations	IDASetSensMaxNonlinIters	3

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. 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 The pointer to ida_mem was NULL.

IDA_NO_SENS Forward sensitivity analysis was not initialized.

IDA_BAD_DKY One of the vectors dkyS is NULL.

IDA_BAD_IS The index is is not in the allowed range.

IDA_BAD_K k is not in the range $0, 1, ..., q_u$.

IDA_BAD_T The time t is not in the allowed range.

Notes In case of an error return, an error message is also printed.

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 $f(t, y, p)$. If non-NULL, p must point to a field in the user's data structure user_data passed to the righ-hand side 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 flags to specify which parameters 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 preceeded 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 one of 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 a factor of DQrhomax and the separate approximation is used otherwise. Note that a value DQrhomax < 1.0 will effectively disable switching. See §2.2 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.

Table 5.2: Forward sensitivity optional outputs

Optional output	Routine name
No. of calls to sensitivity residual function	IDAGetNumSensResEvals
No. of calls to residual function for sensitivity	IDAGetNumResEvalsSens
No. of sensitivity local error test failures	IDAGetNumSensErrTestFails
No. of calls to lin. solv. setup routine for sens.	IDAGetNumSensLinSolvSetups
Error weight vector for sensitivity variables	IDAGetSensErrWeights
No. of sens. nonlinear solver iterations	IDAGetNumSensNonlinSolvIters
No. of sens. convergence failures	IDAGetNumSensNonlinSolvConvFails
No. of staggered nonlinear solver iterations	IDAGetNumStgrSensNonlinSolvIters
No. of staggered convergence failures	IDAGetNumStgrSensNonlinSolvConvFails

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.

`IDAGetNumSensResEvals`

Call `flag = IDAGetNumSensResEvals(ida_mem, &nfSevals);`

Description The function `IDAGetNumSensResEvals` 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.

Notes In order to accommodate any of the three possible sensitivity solution methods, the default internal finite difference quotient functions evaluate the sensitivity residuals one at a time. Therefore, `nfSevals` will always be a multiple of the number of sensitivity parameters (the same as the case in which the user supplies a routine of type `IDASensRhs1Fn`).

`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.

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.

`IDAGetNumSensErrTestFails`

Call `flag = IDAGetNumSensErrTestFails(ida_mem, &nSetfails);`

Description The function `IDAGetNumSensErrTestFails` returns the number of local error test failures for the sensitivity variables that have occured.

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.

`IDAGetNumSensLinSolvSetups`

Call `flag = IDAGetNumSensLinSolvSetups(ida_mem, &nlinsetupsS);`

Description The function `IDAGetNumSensLinSolvSetups` 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.

`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 if either the `ism`=`IDA_STAGGERED` or the `ism`=`IDA_STAGGERED1` sensitivity solution method has been 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 weights 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`.

`IDAGetNumSensNonlinSolvIters`

Call `flag = IDAGetNumSensNonlinSolvIters(ida_mem, &nSniters);`

Description The function `IDAGetNumSensNonlinSolvIters` 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 the `ism` was `IDA_STAGGERED` or `IDA_STAGGERED1` in the call to `IDASensInit` (see §5.2.1).

In the `IDA_STAGGERED1` case, the value of `nSniters` is the sum of the number of nonlinear iterations performed for each sensitivity equation. These individual counters can be obtained through a call to `IDAGetNumStgrSensNonlinSolvIters` (see below).

`IDAGetNumSensNonlinSolvConvFails`

Call `flag = IDAGetNumSensNonlinSolvConvFails(ida_mem, &nSncfails);`

Description The function `IDAGetNumSensNonlinSolvConvFails` 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 the `ism` was `IDA_STAGGERED` or `IDA_STAGGERED1` in the call to `IDASensInit` (see §5.2.1).

In the `IDA_STAGGERED1` case, the value of `nSncfails` is the sum of the number of nonlinear convergence failures that occured for each sensitivity equation. These individual counters can be obtained through a call to `IDAGetNumStgrSensNonlinConvFails` (see below).

`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 (2.10).

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 (2.10), 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 *yyS, 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 $(\partial F/\partial y)s_i(t)+(\partial F/\partial \dot{y})s_i'(t)+(\partial F/\partial p_i)$ 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 the $y'(t)$.
`yS`	contains the current values of the sensitivities of y.
`ypS`	contains the current values of the sensitivities of y'.
`resvalS`	contains the output sensitivities vectors.
`user_data`	is a pointer to user data.
`tmp1`	
`tmp2`	
`tmp3`	are `N_Vector`s which can be used as temporary storage.

Return value A `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 TODO : modify the below text

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

There are two situations in which recovery is not possible even if `IDASensResFn` function returns a recoverable error flag. This include the situation when this occurrs at the very first call to the `IDASensResFn` (in which case IDAS returns `IDA_FIRST_SRHSFUNC_ERR`) or if a recoverable error is reported when `IDASensResFn` is called after an error test failure, while the linear multistep method order is equal to 1 (in which case IDAS returns `IDA_UNREC_SRHSFUNC_ERR`).

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. **Define the sensitivity problem**
10. **Set sensitivity initial conditions**
11. **Activate sensitivity calculations**
12. **Set sensitivity analysis optional inputs**
13. **Set vector of initial values for quadrature variables**

    Typically, the quadrature variables should be initialized to 0.
14. **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.
15. **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.
16. **Advance solution in time**
17. **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.
18. **Get optional outputs**
19. **Extract sensitivity solution**
20. **Get sensitivities-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.
21. **Deallocate memory for solutions vector**
22. **Deallocate memory for sensitivity vectors**
23. **Deallocate memory for sensitivity-dependent quadrature variables**
24. **Free solver memory**
25. **[P] Finalize MPI**

`IDAQuadSensInit` (step 14 above) can be called and quadrature-related optional inputs (step 15 above) can be set, anywhere between steps 9 and 16.

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. 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 occured, `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 by:

- Base value: `lenrw` = `lenrw` + (`maxord+5`)N_q
- if `IDAQuadSensSVtolerances` is called: `lenrw` = `lenrw` $+N_qN_s$

and the size of the integer workspace is increased by:

- Base value: `leniw` = `leniw` + (`maxord+5`)N_q
- if `IDAQuadSensSVtolerances` is called: `leniw` = `leniw` $+N_qN_s$

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 occured, 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.

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 tests 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 both a solution, sensitivites and quadratures depending on sensitivities at time t. However, IDASolve will still return only the solutions y and y'. Sensitivity-dependent quadratures can be obtained using one of the following functions:

IDAGetQuadSens

Call `flag = IDAGetQuadSens(ida_mem, &t, yQS);`

Description The function IDAGetQuadSens returns 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 reached by the solver.
yQS	(N_Vector *) the computed solution 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 is NULL.

Notes In case of an error return, an error message is also sent to the error handler function.

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 *) the vector containing the derivatives. This vector must be allocated by the user.

Return value The return value flag of IDAGetQuadSensDky is one of:

IDA_SUCCESS	IDAGetQuadSensDky succeeded.
IDA_MEM_NULL	The pointer to 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	The vector dkyQ is NULL.
IDA_BAD_K	k is not in the range $0, 1, ..., kused$.
IDA_BAD_T	The time t is not in the allowed range.

Notes In case of an error return, an error message is also sent to the error handler function.

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 = IDAQuadGetSens1(ida_mem, &t, 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.
t	(realtype) the time reached by the solver.
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	yQ is NULL.

Notes In case of an error return, an error message is also printed.

`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`	The pointer to `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`	One of the vectors `dkyS` is `NULL`.
`IDA_BAD_IS`	The index `is` is not in the allowed range.
`IDA_BAD_K`	`k` is not in the range $0, 1, ..., kused$.
`IDA_BAD_T`	The time `t` is not in the allowed range.

Notes In case of an error return, an error message is also printed.

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 should be used in the step size control mechanism. If so, the user must call `IDAQuadSensSStolerances` or `IDAQuadSensSVtolerances` to specify the integration tolerances for the quadrature variables.

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 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, reltolQS, abstolQS);`

Description The function `IDAQuadSensEEtolerances` specifies that tolerances for 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`) 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_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 `resQS` 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 integration of quadrature equations, the user must provide a function that defines the right-hand side of the quadrature equations. This function must be of type `IDAQuadSensRhsFn` defined as follows:

`IDAQuadSensRhsFn`

Definition
```
typedef int (*IDAQuadSensRhsFn)(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
- `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, $y'(t)$.
- `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 vectors s_i'.
- `rrQ` is the current value quadrature right-hand side.
- `rhsvalQS` contains the output vectors.
- `user_data` is the `user_data` pointer passed to `IDASetUserData`.
- `tmp1`
- `tmp2`
- `tmp3` are `N_Vector`s which can be used as temporary storage.

Return value A `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 are two situations in which recovery is not possible even if IDAQuadSensRhsFn function returns a recoverable error flag. This include the situation when this occurrs at the very first call to the IDAQuadSensRhsFn (in which case IDAS returns IDA_FIRST_QRHS_ERR) or if a recoverable error is reported when IDAQuadSensRhsFn is called after an error test failure, while the linear multistep method order is equal to 1 (in which case IDAS returns IDA_UNREC_QRHSFUNC_ERR).

TODO: Fix the last proposition.

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 (§2.2), 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 IDASPGMR). 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 method 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 final correction), 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 (2.18) or (2.23), 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 Chapter 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 §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 with [**P**] correspond to NVECTOR_PARALLEL, while steps marked with [**S**] correspond to NVECTOR_SERIAL.

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` or `nvector_parallel.h` for the two 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. **Allocate space for the adjoint computation**

    Call `IDAAdjInit()` to allocate memory for the combined forward-backward problem (see §6.2.1 for more details). This call requires `Nd`, the number of steps between two consecutive checkpoints. `IDAAdjInit` also specifies the type of interpolation used (see §2.3.3).

12. **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 §6.2.2)). The final value of `tret`, denoted `tfinal`, is then the maximum allowable value for the endpoint t_1.

Backward problem

13. **Set problem dimensions for the backward problem**

    [**S**] set `NB`, the number of variables in the backward problem
    [**P**] set `NB` and `NBlocal`

14. **Create the backward problem**

    Call `IDACreateB`, a wrapper for `IDACreate`, to create the IDAS memory block the new backward problem. Unlike `IDACreate`, the function `IDACreateB` does not return a pointer to the newly created memory block (see §6.2.3). Instead, this pointer is attached to the adjoint memory block (created by `IDAAdjInit` and returns an identifier that user must later specify in any of his/her actions on the newly created backward problem.

15. **Allocate memory for the backward problem**

    Call `IDAInitB` or `IDAInitBS` (when the backward problem depends on the forward sensitivities). The two function are actually wrappers for `IDAInit` and allocate internal memory, specify problem and initialize IDAS at `tB0` for the backward problem (see §6.2.3).

16. **Specify integration tolerances for backward problem**

    Call `IDASStolerancesB(...);` or `IDASvtolerancesB(...);` to specify a scalar relative tolerance and scalar absolute tolerance or 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 §6.2.4 for more information.

17. **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).

18. **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`, `IDADiagB`, `IDASpgmrB`, `IDASpbcgB`, or `IDASptfqmr` (see §6.2.5). 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.

19. **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 wrapper 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.

20. **Integrate backward problem**

    Call `IDASolveB`, a second wrapper around the IDAS main integration function `IDASolve`, to integrate the backward problem from `tB0` (see §6.2.7). 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 of the forward problem.

21. **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`.

22. **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 `IDASolveFree` to free the IDAS memory block for the forward problem, and a call to `IDAAdjFree` (see §6.2.1) to free the memory allocated for the combined problem. Note that `IDAAdjFree` also deallocates the IDAS memory for the backward problems.

23. **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 (13)-(21) 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.

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 N_d interpolation data points and a linked list of checkpoints is initialized.

Arguments `ida_mem` (`void *`) is 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 §2.3.3).

Return value The return value `flag` of `IDAAdjInit` 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 (2`Nd`+3) variables of type `N_Vector`.

If an error occured, `IDAAdjInit` also prints an error message to the file specified by the optional input `errfp`.

`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 is the IDAS memory block 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.

6.2.2 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 in `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.

`yret` (`N_Vector`) the computed solution vector y.

`ypret` (`N_Vector`) the computed solution vector y'.

`itask` (`int`) a flag indicating the job of the solver for the next 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 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.

`ncheck` (`int *`) the number of check points 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 $y'(t)$. Additionally, it returns in `ncheck` the number of checkpoints saved. 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 check points 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 checkpoints data.

6.2.3 Backward problem initialization functions

The functions `IDACreateB` and `IDAInitB` (or `IDAInitBS`) must be called in the order listed. They instantiate a 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 a IDAS solver object and specifies the solution method for the backward problem.

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

`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.

The function `IDAInitB` is essentially a call to `IDAInit` with some particularization for backward integration as described below. It is essentially a wrapper for `IDAInit` and so all details given for `IDAReInit` in §4.5.10 apply.

`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 adjoint memory block returned by `IDAInit`.
- `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).
- `tB0` (`realtype`) specifies the endpoint where final conditions are provided for the backward problem.
- `yB0` (`N_Vector`) is the final value of the backward problem.
- `ypB0` (`N_Vector`) is the derivative final value of the backward problem.

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, `yB0`, `ypB0` or `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 from that of another.

`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 adjoint memory block returned by `IDAInit`.
- `which` (`int`) represents the identifier of the backward problem.
- `resBS` (`IDAResFnBS`) is the C function which computes fB, the residual of the backward DAE problem. This function has the form `resBS(t, y, yp, yB, ypB, yS, ypS, resvalB, user_dataB)` (for full details see §6.3).

tB0	(realtype) specifies the endpoint where final conditions are provided for the backward problem.
yB0	(N_Vector) is the final value of the backward problem.
ypB0	(N_Vector) is the derivative final value of the backward problem.

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, yB0, ypB0 or resB was NULL or sensitivities has not been active during the forward integration.

Notes The memory allocated by IDAInitBS is deallocated by the function IDAAdjFree.

Note that 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 the backward problem, even it has been initialized with the sensitivity-dependent version routine 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 IDAInit.
which	(int) represents the identifier of the backward problem.
tB0	(realtype) specifies the endpoint where final conditions are provided for the backward problem.
yB0	(N_Vector) is the final value of the backward problem.
ypB0	(N_Vector) is the derivative final value of the backward problem.

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 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, yB0 or ypB0 was NULL.

6.2.4 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 `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 `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.

6.2.5 Linear solver initialization functions for backward problem

All linear solver modules in IDAS 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 six 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 their second argument which must be the identifier of the backward problem.

```
flag = IDADenseB(ida_mem, which, nB);
flag = IDABandB(ida_mem, which, nB, mupperB, mlowerB);
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 `IDADENSE_MEM_NULL`, `IDADIAG_MEM_NULL`, `IDABAND_MEM_NULL`, or `IDASPILS_MEM_NULL`. Also, if `which` is not a valid identifier, the functions will return `IDADENSE_ILL_INPUT`, `IDADIAG_ILL_INPUT`, `IDABAND_ILL_INPUT`, or `IDASPILS_ILL_INPUT`

6.2.6 Initial condition calculation functions for backward problem

IDAA provides support for calculation of consistent initial conditions for 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 solve the adjoint system.

The above functions provide the same functionality for backward problem as IDACalcIC with parameter icopt = IDA_YA_YDP_INIT provides for forward problem (see §4.5.5): compute the algebraic components of yB and differential components of yB', 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 final time $tB0$. IDACalcICBS also needs forward sensitivities at final time $tB0$.

IDACalcICB

Call `flag = IDACalcICB(ida_mem, which, tout1, N_Vector y0, N_Vector yp0);`

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.
- tout1 (realtype) is the first value of t at which a solution will be requested (from IDASolveB).This value is needed here to determine the direction of integration and rough scale in the independent variable t.
- y0 (N_Vector) the forward solution at final time $tB0$.
- yp0 (N_Vector) the forward derivative solution at final time $tB0$.

Return value The return value flag (of type int) can be any that is returned by IDACalcIC (see §4.5.5). 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 $yB(tB_0)$ and $yB'(tB_0)$ 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 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, tout1,
                   N_Vector y0, N_Vector yp0,
                   N_Vector yS0, N_Vector ypS0);
```

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.
- tout1 (realtype) is the first value of t at which a solution will be requested (from IDASolveB).This value is needed here to determine the direction of integration and rough scale in the independent variable t.
- y0 (N_Vector) the forward solution at final time $tB0$.
- yp0 (N_Vector) the forward derivative solution at final time $tB0$.

`yS` (`N_Vector *`) a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution at final time $tB0$.

`ypS` (`N_Vector *`) a pointer to an array of `Ns` vectors containing the sensitivities of the forward derivative solution at final time $tB0$.

Return value The return value `flag` (of type `int`) can be any that is returned by `IDACalcIC` (see §4.5.5). 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 $yB(tB_0)$ and $yB'(tB_0)$ which were specified in the previous call to `IDAInitBS` or `IDAReInitBS`. To obtain the corrected values, call `IDAGetconsistentICB` (see §6.2.10.2).

6.2.7 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 integrations between consecutive checkpoints. 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 call to this function 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 `IDAInit`.

`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 `tBout` parameter. The solver then interpolates in order to return an approximate value of yB(`tBout`). 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 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 through 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 occured 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.

6.2.8 Adjoint sensitivity optional input

User can disable anytime during the integration of the forward problem 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 anymore.

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.

6.2.9 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 specifies 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 function:

`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 `IDAInit`.
- `which` (`int`) represents the identifier of the backward problem.
- `djacB` (`IDADlsDenseJacFnB`) user-defined dense Jacobian approximation function.

Return value The return value `flag` (of type `int`) is one of:
- `IDADIRECT_SUCCESS` `IDADlsSetDenseJacFnB` succeeded.
- `IDADIRECT_MEM_NULL` The `ida_mem` was `NULL`.
- `IDADIRECT_NO_ADJ` The function `IDAAdjInit` has not been previously called.
- `IDADIRECT_LMEM_NULL` The IDADENSE linear solver has not been initialized through a call to `IDADenseB`.
- `IDADIRECT_ILL_INPUT` The parameter `which` represented an invalid identifier.

Notes The function type `IDADlsDenseJacFnB` is described in §6.3.

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 function:

`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 `IDAInit`.
- `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:
- `IDADIRECT_SUCCESS` `IDADlsSetBandJacFnB` succeeded.
- `IDADIRECT_MEM_NULL` The `ida_mem` was `NULL`.
- `IDADIRECT_NO_ADJ` The function `IDAAdjInit` has not been previously called.
- `IDADIRECT_LMEM_NULL` The IDADENSE linear solver has not been initialized through a call to `IDABandB`.
- `IDADIRECT_ILL_INPUT` The parameter `which` represented an invalid identifier.

Notes The function type `IDABandJacFnB` is described in §6.3.

6.2.9.4 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 successfuly set.
- `IDASPILS_MEM_NULL` The `ida_mem` memory block was `NULL`.
- `IDASPILS_LMEM_NULL` The IDASPGMR 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.

`IDASpilsSetJacTimesVecFnB`

Call `flag = IDASpilsSetJacTimesVecFnB(ida_mem, which, jtvB);`

Description The function `IDASpilsSetJacTimesFnB` 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 successfuly set.
- `IDASPILS_MEM_NULL` The `ida_mem` memory block was `NULL`.
- `IDASPILS_LMEM_NULL` The IDASPGMR 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.

`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 successfuly set.
`IDASPILS_MEM_NULL`	The `ida_mem` memory block was `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPGMR 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 Gram-Schmidt orthogonalization type `gstypeB` is 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 successfuly set.
`IDASPILS_MEM_NULL`	The `ida_mem` memory block was `NULL`.
`IDASPILS_LMEM_NULL`	The IDASPGMR 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

Notes The maximum subspace dimension is initially specified in the call to `IDASpbcgB` or `IDASptfqmrB`. The call to `IDASpilsSetMaxlB` is needed only if `maxl` is being changed from its previous value.

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

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 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 a pointer to this memory block:

`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 using `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 in any way `ida_memB`.

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 successfuly 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 `yy0_mod` and `yp0_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. While one of the `IDAQuadInitB` or `IDAQuadInitBS` should be used to allocate internal memory and to initialize backward quadratures, the same function should be called for any other operation (extraction, optional input/output, reinitialization, deallocation).

6.2.11.1 Backward quadrature initialization functions

The function `IDAQuadInitB` initializes and allocates memory for the backward integration of quadrature equations. 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` The `ida_mem` memory block 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 dependes on the forward sensitivites.

`IDAQuadInitBS`

Call `flag = IDAQuadInitB(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.
- `rhsQB` (`IDAQuadRhsFnBS`) is the C function which computes $fQBS$, the residual of the backward quadrature equations. This function has the form `rhsQB(t, y, yp, yB, ypB, yS, ypS, 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` The `ida_mem` memory block 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 `IDAReInitB` was successful.
- `IDA_MEM_NULL` The `ida_mem` memory block 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 not only after a call to `IDAQuadInitB` but also to `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, &t, 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.
- `t` (`realtype`) the time reached by the solver.
- `yQB` (`N_Vector`) the computed quadrature vector.

Return value 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`	`yQ` is `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 specifies the identifier `which` of the backward problem for which the optional values are specified.

```
flag = IDASetQuadErrConB(ida_mem, which, errconQ);
flag = IDAQuadSVtolerancesB(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 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).

6.2.12 Optional output from the adjoint module

6.2.12.1 Checkpoint information function

For debugging purposes, IDAS provides a function `IDAAdjGetCheckPointsInfo` which returns partial information from the linked list of checkpoints generated by `IDASolveF`. The call to this function has the form:

`IDAGetAdjCheckPointsInfo`

Call `flag = IDAGetAdjCheckPointsInfo(ida_mem, ckpnt);`

Description The function `IDAGetAdjCheckPointsInfo` returns a structure array with checkpoint information.

Arguments `ida_mem` (`void *`) pointer to the adjoint memory returned by `IDAAdjInit`.

`ckpnt` (`IDAadjCheckPointRec *`) an array of `ncheck`+1 structures with checkpoint information, where `ncheck` is the numebr of checkpoints returned by `IDASolveF`.

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

`IDA_SUCCESS`	`IDAGetAdjCheckPointsInfo` was successful.
`IDA_MEM_NULL`	`ida_mem` is `NULL`.
`IDA_NO_ADJ`	The function `IDAAdjInit` has not been previously called.

Notes The user must allocate space for ckpnt (ncheck+1 structures).

For an example of using `IDAAdjGetCheckPointsInfo`, see the `idaadjdenx` example.

The type `IDAadjCheckPointRec` is defined in the header file `idas.h`:

```
typedef struct {
  void *my_addr;
  void *next_addr;
  realtype t0;
  realtype t1;
  long int nstep;
  int order;
  realtype step;
} IDAadjCheckPointRec;
```

The fields in this structure have the following meanings:

`my_addr`	Address of current checkpoint.
`next_addr`	Address of next checkpoint.
`t0`	
`t1`	Time interval between current and next checkpoint.
`nstep`	Step number at which the current checkpoint was saved.
`order`	Linear multistep method order at the current checkpoint.
`step`	Integration stepsize at current checkpoint.

6.2.12.2 Interpolation data

Fo debugging purposes, IDAA provides two extraction functions which return the data stored for interpolation purposes.

`IDAAdjGetDataPointHermite`

Call `int = IDAAdjGetDataPointHermite(ida_mem, which, &t, y, yd);`

Description The function `IDAAdjGetDataPointHermite` returns the time and two vectors associated with the `which` interpolation data point.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`which`	(`long int`) index of the intepolation data point.
`t`	(`realtype *`)
`y`	(`N_Vector`)
`yd`	(`N_Vector`) time, solution, and solution derivative for the forward problem stored for interpolation purposes at the `which` data point.

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

`IDA_SUCCESS`	`IDAAdjGetDataPointHermite` was successful.
`IDA_MEM_NULL`	`ida_mem` is `NULL`.
`IDA_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDA_ILL_INPUT`	The interpolation type was not cubic Hermite.

Notes It is the user's responsibility to allocate space for `y` and `yd`.

`IDAAdjGetDataPointPolynomial`

Call `int = IDAAdjGetDataPointPolynomial(ida_mem, which, &t, order, y);`

Description The function `IDAAdjGetDataPointPolynomial` returns the time and two vectors associated with the `which` interpolation data point.

Arguments

`ida_mem`	(`void *`) pointer to the IDAS memory block.
`which`	(`long int`) index of the intepolation data point.
`t`	(`realtype *`)

`order`	(`int`)
`yd`	(`N_Vector`) time, method order, and solution of the forward problem stored for interpolation purposes at the `which` data point.

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

`IDA_SUCCESS`	`IDAAdjGetDataPointHermite` was successful.
`IDA_MEM_NULL`	`ida_mem` is `NULL`.
`IDA_NO_ADJ`	The function `IDAAdjInit` has not been previously called.
`IDA_ILL_INPUT`	The interpolation type was not variable-order polynomial.

Notes It is the user's responsibility to allocate space for `y`.

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 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 (2.18) or (2.23).

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 derivative solution vector.
`yB`	is the current value of the dependent variable vector.
`ypB`	is the current value of the dependent derivative variable vector.
`resvalB`	is the output vector containing the residual of the backward DAE problem.
`user_dataB`	is a pointer to user data, same as passed to `IDASetUserDataB`.

Return value A `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 it failed unrecoverably (in which case the integration is halted and `IDASolveB` returns `IDA_RHSFUNC_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 IDAResFnB, IDAA needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurrs in the interpolation, IDAA triggers an unrecoverable failure in the righ-hand side function which will halt the integration and IDASolveB will return IDA_RHSFUNC_FAIL.

6.3.2 DAE residual for the backward problem depending on the forward sensitivities

The user must provide a 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 (2.18) or (2.23).

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 derivative solution 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 sensitivities of the forward derivative solution.
yB	is the current value of the dependent variable vector.
ypB	is the current value of the dependent derivative variable vector.
resvalB	is the output vector containing the residual of the backward DAE problem.
user_dataB	is a pointer to user data, same as passed to IDASetUserDataB.

Return value A 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 it failed unrecoverably (in which case the integration is halted and IDASolveB returns IDA_RHSFUNC_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 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 IDAResFnBS, IDAA needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurrs in the interpolation, IDAA triggers an unrecoverable failure in the righ-hand side function which will halt the integration and IDASolveB will return IDA_RHSFUNC_FAIL.

6.3.3 Quadrature right-hand side for the backward problem

The user must provide a 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 derivative solution vector.
`yB`	is the current value of the dependent variable vector.
`ypB`	is the current value of the dependent variable vector.
`rhsvalBQ`	is the output vector containing the residual of the backward quadrature equations.
`user_dataB`	is a pointer to user data, same as passed to `IDASetUserDataB`.

Return value A `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 `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 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 `IDAQuadRhsFnB`, IDAA needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurrs in the interpolation, IDAA triggers an unrecoverable failure in the quadrature righ-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 a function of type `IDAQuadRhsFnBS` defined by

`IDAQuadRhsFnBS`

Definition
```
typedef int (*IDAQuadRhsFnBS)(realtype t, N_Vector y, N_Vector yp,
                              N_Vector yB, N_Vector ypB,
                              N_Vector *yS, N_Vector *ypS,
                              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 derivative solution 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 sensitivities of the forward derivative solution.
yB	is the current value of the dependent variable vector.
ypB	is the current value of the dependent variable vector.
rhsvalBQ	is the output vector containing the residual of the backward quadrature equations.
user_dataB	is a pointer to user data, same as passed to IDASetUserDataB.

Return value A 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 rhsvalQS 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 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 IDAQuadRhsFnBS, IDAA needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurrs in the interpolation, IDAA triggers an unrecoverable failure in the quadrature righ-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 is called in step 18 of §6.1), the user may provide, through a call to IDADlsSetDenseJacFnB (see §6.2.9), a function of the following type:

IDADlsDenseJacFnB

Definition

```
typedef int (*IDADlsDenseJacFnB)(long int NeqB, realtype tt, realtype c_jB
                                 N_Vector yy, N_Vector yp,
                                 N_Vector yyB, 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.
c_jB	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 derivative solution vector.
yyB	is the current value of the dependent variable vector.

ypB	is the current value of the dependent derivative variable vector.
resvalB	is the current value of the residual of the backward problem.
user_dataB	is a pointer to user data - the same as 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 A 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 on IDADENSE_JACFUNC_REIDAR), 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 on IDADENSE_JACFUNC_UNREIDAR).

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,yyB), where yy is the solution of the original IVP at time tt and yyB 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 DenseMat. The user is referred to §4.6.5 for details regarding accessing a DenseMat object.

Before calling the user's IDADlsDenseJacFnB, IDAA needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurrs in the interpolation, IDAA triggers an unrecoverable failure in the Jacobian function which will halt the integration (IDASolveB returns IDA_LSETUP_FAIL and IDADENSE sets last_flag on IDADENSE_JACFUNC_UNREIDAR).

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 is called in step 18 of §6.1), the user may provide, through a call to IDADlsSetBandJacFnB (see §6.2.9), a function the following type:

IDADlsBandJacFnB

Definition
```
typedef int (*IDABandJacFnB)(long int NeqB, int mupperB, int mlowerB,
                             realtype tt, realtype c_jB
                             N_Vector yy, N_Vector yp,
                             N_Vector yyB, 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.
c_jB	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 derivative solution vector.

yyB	is the current value of the dependent variable vector.
ypB	is the current value of the dependent derivative variable vector.
resvalB	is the current value of the residual of the backward problem.
user_dataB	is a pointer to user data - the same as the parameter passed to IDASetUserDataB.
tmp1B	
tmp2B	
tmp3B	are pointers to memory allocated for variables of type N_Vector which can be used by IDABandJacFnB as temporary storage or work space.

Return value A IDABandJacFnB 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 on IDABAND_JACFUNC_REIDAR), 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 on IDABAND_JACFUNC_UNREIDAR).

Notes A user-supplied band Jacobian function must load the band matrix JacB (of type BandMat) with the elements of the Jacobian at the point (tt,yy,yyB), where yy is the solution of the original IVP at time tt and yyB 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 BandMat object and on the rest of the arguments passed to a function of type IDABandJacFnB are given in §4.6.6.

Before calling the user's IDABandJacFnB, IDAA needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurrs in the interpolation, IDAA triggers an unrecoverable failure in the Jacobian function which will halt the integration (IDASolveB returns IDA_LSETUP_FAIL and IDABAND sets last_flag on IDABAND_JACFUNC_UNREIDAR).

6.3.7 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 18 of §6.1), the user may provide a function of type IDASpilsJacTimesVecFnB in the following form:

IDASpilsJacTimesVecFnB

Definition

```
typedef int (*IDASpilsJacTimesVecFnB)(realtype t,
                                      N_Vector yy, N_Vector yp,
                                      N_Vector yyB, N_Vector yyB,
                                      N_Vector resvalB,
                                      N_Vector vB, N_Vector JvB,
                                      realtype c_jB, void *user_dataB,
                                      N_Vector tmp1B, N_Vector tmp2B);
```

Purpose This function computes the action of the Jacobian on a given vector vB for the backward problem (or an approximation to it).

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 derivative solution vector.
yB	is the current value of the dependent variable vector.
ypB	is the current value of the dependent derivative variable vector.
resvalB	is the current value of the residual of the backward problem.
vB	is the vector by which the Jacobian must be multiplied to the right.

JvB	is the output vector computed.
c_j	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 IDASpilsJacTimesVecFn 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 result of the product between 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 §4.6.7). If the backward problem is the adjoint of $\dot{y} = f(t, y)$, then this function is to compute $-(\partial f/\partial y)^T v_B$.

6.3.8 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 may be either a left or a right preconditioner matrix. This function must be of type IDASpilsPrecSolveFnB defined by

IDASpilsPrecSolveFnB

Definition
```
typedef int (*IDASpilsPrecSolveFnB)(realtype t,
                                    N_Vector y, N_Vector yp,
                                    N_Vector yB, N_Vector ypB,
                                    N_Vector resvalB,
                                    N_Vector rvecB, N_Vector zvecB,
                                    realtype c_jB, 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.
y	is the current value of the forward solution vector.
yp	is the current value of the forward derivative solution vector.
yB	is the current value of the dependent variable vector.
ypB	is the current value of the dependent derivative variable vector.
resvalB	is the current value of the residual of the backward problem.
rvecB	is the right-hand side vector r of the linear system to be solved.
zvecB	is the output vector computed.
c_jB	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).

6.3.9 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 type `IDASpilsPrecSetupFnB` defined by

`IDASpilsPrecSetupFnB`

Definition

```
typedef int (*IDASpilsPrecSetupFnB)(realtype t,
                                    N_Vector y, N_Vector yp,
                                    N_Vector yB, N_Vector ypB,
                                    N_Vector resvalB,
                                    realtype c_jB, 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 a `IDASpilsPrecSetupFnB` are as follows:

- `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 vector.
- `yB` is the current value of the dependent variable vector.
- `ypB` is the current value of the dependent variable vector.
- `resvalB` is the current value of the residual of the backward problem.
- `c_jB` 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 IDABBDPRE 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 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

`IDABBDPrecInitB`

Call
```
flag = IDABBDPrecInitB(ida_mem, int which, NlocalB,
                       mudqB, mldqB,
                       mukeepB, mlkeepB,
                       dqrelyB, GreB, 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.
- `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`$= 0.0$.
- `GreB` (`IDABBDLocalFnB`) the C function which computes the approximation $g_B(t, y)$ to the right-hand side of the backward problem.
- `GcommB` (`IDABBDCommFnB`) the optional C function which performs all interprocess communication required for the computation of $g_B(t, y)$.

Return value If successful, `IDABBDPrecInitB` stores 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 specify the use of the IDASPGMR linear solver module with the IDABBDPRE preconditioner module, make the following call:

`IDABBDSpgmrB`

Call
```
flag = IDABBDSpgmrB(ida_mem, which, maxlB);
```

Description The function `IDABBDSpgmrB` links the IDABBDPRE data to the IDASPGMR linear solver and attaches the latter to the IDAS memory block for the backward problem.

Arguments
- `ida_mem` (`void *`) pointer to the IDAS memory block returned by `IDAAdjInit`.
- `which` (`int`) The identifier of the backward problem.
- `maxlB` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `IDASPILS_MAXL`$= 5$.

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

`IDASPILS_SUCCESS`	The IDASPGMR initialization was successful.

IDASPILS_MEM_FAIL	A memory allocation request has failed.
IDASPILS_MEM_NULL	The ida_mem argument was NULL.
IDASPILS_NO_ADJ	The function IDAAdjInit has not been previously called.
IDASPILS_LMEM_NULL	No linear solver has been attached.
IDASPILS_ILL_INPUT	An invalid parameter has been passed.

To specify the use of the IDASPBCG linear solver module with the IDABBDPRE preconditioner module, make the following call:

IDABBDSpbcgB

Call `flag = IDABBDSpbcgB(ida_mem, which, maxlB);`

Description The function IDABBDSpbcgB links the IDABBDPRE data to the IDASPBCG linear solver and attaches the latter to the IDAS memory block for the backward problem.

Arguments

ida_mem	(void *) pointer to the IDAS memory block returned by IDAAdjInit.
which	(int) The identifier of the backward problem.
maxlB	(int) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value IDASPILS_MAXL= 5.

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

IDASPILS_SUCCESS	The IDASPGMR initialization was successful.
IDASPILS_MEM_FAIL	A memory allocation request has failed.
IDASPILS_MEM_NULL	The ida_mem argument was NULL.
IDASPILS_NO_ADJ	The function IDAAdjInit has not been previously called.
IDASPILS_LMEM_NULL	No linear solver has been attached.
IDASPILS_ILL_INPUT	An invalid parameter has been passed.

To specify the use of the IDASPTFQMR linear solver module with the IDABBDPRE preconditioner module, make the following call:

IDABBDSptfqmrB

Call `flag = IDABBDSptfqmrB(ida_mem, which, maxlB);`

Description The function IDABBDSptfqmrB links the IDABBDPRE data to the IDASPTFQMR linear solver and attaches the latter to the IDAS memory block for the backward problem.

Arguments

ida_mem	(void *) pointer to the IDAS memory block returned by IDAAdjInit.
which	(int) The identifier of the backward problem.
maxlB	(int) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value IDASPILS_MAXL= 5.

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

IDASPILS_SUCCESS	The IDASPGMR initialization was successful.
IDASPILS_MEM_FAIL	A memory allocation request has failed.
IDASPILS_MEM_NULL	The ida_mem argument was NULL.
IDASPILS_NO_ADJ	The function IDAAdjInit has not been previously called.
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 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 `IDAAdjInit`.
- `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 `IDABBDPrecInitB` 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 `glocB` (of type `IDABBDLocalFnB`) which approximates the residual of the backward problem and which is computed locally, and an optional function `cfnB` (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)(int NlocalB, realtype t,
                              N_Vector y, N_Vector yp,
                              N_Vector yB, N_Vector ypB,
                              N_Vector gB,
                              void *user_dataB);
```

Purpose This function loads the vector `gB` as a function of `t`, `y`, and `yB`.

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 derivative solution vector.
- `yB` is the current value of the dependent variable vector.
- `ypB` is the current value of the dependent derivatice variable vector.
- `gB` is the output vector.
- `user_dataB` is a pointer to user data - the same as the `user_dataB` parameter passed to `IDASetUserDataB`.

Return value A `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 assumes 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, IDAA needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurrs in the interpolation, IDAA 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 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 derivative solution vector.
yB	is the current value of the dependent variable vector.
ypB	is the current value of the dependent derivatice variable vector.
user_dataB	is a pointer to user data - the same as the user_dataB parameter passed to IDASetUserDataB.

Return value A IDABBDCommFn 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 two provided within SUNDIALS, a serial and an MPI parallel implementations.

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

```
typedef struct _generic_N_Vector *N_Vector;

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

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

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

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

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

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

Table 7.1 contains a complete list of all vector operations defined by the generic NVECTOR module.

Finally, note that the generic NVECTOR module defines the functions `N_VCloneVectorArray` and `N_VCloneEmptyVectorArray`. Both functions create (by cloning) an array of `count` variables of type `N_Vector`, each of the same type as an existing `N_Vector`. Their prototypes are

```
N_Vector *N_VCloneVectorArray(int count, N_Vector w);
N_Vector *N_VCloneEmptyVectorArray(int count, N_Vector w);
```

and their definitions are based on the implementation-specific `N_VClone` and `N_VCloneEmpty` operations, respectively.

An array of variables of type `N_Vector` can be destroyed by calling `N_VDestroyVectorArray`, whose prototype is

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

and whose definition is based on the implementation-specific `N_VDestroy` operation.

A particular implementation of the NVECTOR module must:

- Specify the *content* field of `N_Vector`.
- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different `N_Vector` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free an `N_Vector` with the new *content* field and with *ops* pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `N_Vector` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the *content* field of the newly defined `N_Vector`.

Table 7.1: Description of the NVECTOR operations

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

continued on next page

continued from last page

Name	Usage and Description
N_VScale	`N_VScale(c, x, z);` Scales the `N_Vector x` by the scalar `c` and returns the result in `z`: $z_i = cx_i,\ i = 0, \ldots, n-1$.
N_VAbs	`N_VAbs(x, z);` Sets the components of the `N_Vector z` to be the absolute values of the components of the `N_Vector x`: $y_i = \lvert x_i \rvert,\ i = 0, \ldots, n-1$.
N_VInv	`N_VInv(x, z);` Sets the components of the `N_Vector z` to be the inverses of the components of the `N_Vector x`: $z_i = 1.0/x_i,\ i = 0, \ldots, n-1$. This routine may not check for division by 0. It should be called only with an `x` which is guaranteed to have all nonzero components.
N_VAddConst	`N_VAddConst(x, b, z);` Adds the scalar `b` to all components of `x` and returns the result in the `N_Vector z`: $z_i = x_i + b,\ i = 0, \ldots, n-1$.
N_VDotProd	`d = N_VDotProd(x, y);` Returns the value of the ordinary dot product of `x` and `y`: $d = \sum_{i=0}^{n-1} x_i y_i$.
N_VMaxNorm	`m = N_VMaxNorm(x);` Returns the maximum norm of the `N_Vector x`: $m = \max_i \lvert x_i \rvert$.
N_VWrmsNorm	`m = N_VWrmsNorm(x, w)` Returns the weighted root-mean-square norm of the `N_Vector x` with weight vector `w`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i)^2\right)/n}$.
N_VWrmsNormMask	`m = N_VWrmsNormMask(x, w, id);` Returns the weighted root mean square norm of the `N_Vector x` with weight vector `w` built using only the elements of `x` corresponding to nonzero elements of the `N_Vector id`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i \mathrm{sign}(id_i))^2\right)/n}$.
N_VMin	`m = N_VMin(x);` Returns the smallest element of the `N_Vector x`: $m = \min_i x_i$.
N_VWL2Norm	`m = N_VWL2Norm(x, w);` Returns the weighted Euclidean ℓ_2 norm of the `N_Vector x` with weight vector `w`: $m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}$.
N_VL1Norm	`m = N_VL1Norm(x);` Returns the ℓ_1 norm of the `N_Vector x`: $m = \sum_{i=0}^{n-1} \lvert x_i \rvert$.
N_VCompare	`N_VCompare(c, x, z);` Compares the components of the `N_Vector x` to the scalar `c` and returns an `N_Vector z` such that: $z_i = 1.0$ if $\lvert x_i \rvert \geq c$ and $z_i = 0.0$ otherwise.

continued on next page

continued from last page

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

7.1 The NVECTOR_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of *data*.

```
struct _N_VectorContent_Serial {
  long int length;
  booleantype own_data;
  realtype *data;
};
```

The following five macros are provided to access the content of an NVECTOR_SERIAL vector. The suffix `_S` in the names denotes serial version.

- `NV_CONTENT_S`

  This routine gives access to the contents of the serial vector `N_Vector`.

  The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial `N_Vector` content structure.

  Implementation:

  ```
  #define NV_CONTENT_S(v) ( (N_VectorContent_Serial)(v->content) )
  ```

- `NV_OWN_DATA_S`, `NV_DATA_S`, `NV_LENGTH_S`

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

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

  The assignment `v_len = NV_LENGTH_S(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_S(v) = len_v` sets the length of `v` to be `len_v`.

Implementation:

```
#define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
#define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
#define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
```

- `NV_Ith_S`

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

  The assignment `r = NV_Ith_S(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_S(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
  ```

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in Table 7.1. Their names are obtained from those in Table 7.1 by appending the suffix `_Serial`. The module NVECTOR_SERIAL provides the following additional user-callable routines:

- `N_VNew_Serial`

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

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

- `N_VNewEmpty_Serial`

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

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

- `N_VMake_Serial`

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

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

- `N_VCloneVectorArray_Serial`

  This function creates (by cloning) an array of `count` serial vectors.

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

- `N_VCloneVectorArrayEmpty_Serial`

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

  ```
  N_Vector *N_VCloneVectorArrayEmpty_Serial(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Serial`

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

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

- `N_VPrint_Serial`

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

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

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_S(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_S(v,i)` within the loop.
- `N_VNewEmpty_Serial`, `N_VMake_Serial`, and `N_VCloneVectorArrayEmpty_Serial` set the field *own_data* = `FALSE`. `N_VDestroy_Serial` and `N_VDestroyVectorArray_Serial` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.
- To maximize efficiency, vector operations in the NVECTOR_SERIAL implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

7.2 The NVECTOR_PARALLEL implementation

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

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

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

- `NV_CONTENT_P`

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

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

  Implementation:

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

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

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

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

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

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

  Implementation:

  ```
  #define NV_OWN_DATA_P(v)   ( NV_CONTENT_P(v)->own_data )
  #define NV_DATA_P(v)       ( NV_CONTENT_P(v)->data )
  ```

```
#define NV_LOCLENGTH_P(v)  ( NV_CONTENT_P(v)->local_length )
#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )
```

- NV_COMM_P

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

  Implementation:

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

- NV_Ith_P

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

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

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

  Implementation:

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

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Table 7.1 Their names are obtained from those in Table 7.1 by appending the suffix _Parallel. The module NVECTOR_PARALLEL provides the following additional user-callable routines:

- N_VNew_Parallel

  This function creates and allocates memory for a parallel vector.

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

- N_VNewEmpty_Parallel

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

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

- N_VMake_Parallel

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

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

- N_VCloneVectorArray_Parallel

  This function creates (by cloning) an array of count parallel vectors.

  ```
  N_Vector *N_VCloneVectorArray_Parallel(int count, N_Vector w);
  ```

- N_VCloneVectorArrayEmpty_Parallel

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

```
N_Vector *N_VCloneVectorArrayEmpty_Parallel(int count, N_Vector w);
```

- `N_VDestroyVectorArray_Parallel`

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

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

- `N_VPrint_Parallel`

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

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

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the local component array via `v_data = NV_DATA_P(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_P(v,i)` within the loop.
- `N_VNewEmpty_Parallel`, `N_VMake_Parallel`, and `N_VCloneVectorArrayEmpty_Parallel` set the field *own_data* = `FALSE`. `N_VDestroy_Parallel` and `N_VDestroyVectorArray_Parallel` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.
- To maximize efficiency, vector operations in the NVECTOR_PARALLEL implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

7.3 NVECTOR functions used by IDAS

In Table 7.2 below, we list the vector functions 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 (IDASPILS stands for any of IDASPGMR, IDASPBCG, or IDASPTFQMR), the IDABBDPRE preconditioner module, and the FIDA module.

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 implementation file `ida_spgmr.c` for the IDASPGMR solver and within the implementation files `sundials_spgmr.c` and `sundials_iterative.c` for the generic SPGMR solver upon which the IDASPGMR solver is implemented. Also, although `N_VDiv` and `N_VProd` are not called within the implementation file `ida_spgmr.c`, they are called within the implementation file `sundials_spgmr.c` and so are required by the IDASPGMR solver module. 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	IDADENSE	IDABAND	IDASPILS	IDABBDPRE	FIDA
N_VClone	✓			✓	✓	
N_VDestroy	✓			✓	✓	
N_VSpace	✓					
N_VGetArrayPointer		✓	✓		✓	✓
N_VSetArrayPointer		✓				✓
N_VLinearSum	✓	✓		✓		
N_VConst	✓			✓		
N_VProd	✓			✓		
N_VDiv	✓			✓		
N_VScale	✓	✓	✓	✓	✓	
N_VAbs	✓					
N_VInv	✓					
N_VAddConst	✓					
N_VDotProd				✓		
N_VMaxNorm	✓					
N_VWrmsNorm	✓					
N_VMin	✓					
N_VMinQuotient	✓					
N_VConstrMask	✓					
N_VWrmsNormMask	✓					
N_VCompare	✓					

Chapter 8

Providing Alternate Linear Solver Modules

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

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

A linear solver module must also provide a user-callable specification routine (like those described in §4.5.3) which will attach the above five routines 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 a `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 four interface routines, only the `lsolve` routine is required. The `lfree` routine must be provided only if the solver specification routine makes any memory allocation. For consistency with the existing IDAS linear solver modules, we recommend that the return value of the specification function be 0 for a successful return or a negative value if an error occurs (the pointer to the main IDAS memory block is `NULL`, an input is illegal, the NVECTOR implementation is not compatible, a memory allocation fails, etc.)

To facilitate data exchange between the five interface functions, the field `ida_lmem` in the IDAS memory block can be used to attach a linear solver-specific memory block.

These five routines, 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 routines. Note that the call list of each routine includes a pointer to the main IDAS memory block, by which the routine can access various data related to the IDAS solution. The contents of this memory block are given in the file `idas.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 a specific linear solver, such as counters and statistics.

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 routine

The type definition of `lsetup` is

`lsetup`

Definition
```
int (*lsetup)(IDAMem IDA_mem, N_Vector yyp, N_Vector ypp,
              N_Vector resp,
              N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);
```

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

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

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

8.3 Solve routine

The type definition of `lsolve` is

`lsolve`

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

Purpose The routine `lsolve` must solve the linear equation $Mx = b$, where M is some approximation to $J = \partial F/\partial y + c_j \partial F/\partial y'$ (see Eqn. (2.6)), and the right-hand side vector b is input.

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).
- `ycur` is a vector that contains the solver's current approximation to $y(t_n)$.
- `ypcur` is a vector that contains the solver's current approximation to $y'(t_n)$.
- `rescur` is a vector that contains $F(t_n, y_{cur}, y'_{cur})$.

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

8.4 Performance monitoring routine

The type definition of `lperf` is

`lperf`

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

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

Arguments `IDA_mem` is the 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.

Return value The `lperf` return value is ignored.

8.5 Memory deallocation routine

The type definition of `lfree` is

`lfree`

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

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

Arguments The argument `IDA_mem` is the IDAS memory pointer of type `IDAMem`.

Return value This routine has no return value.

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

Chapter 9

Generic Linear Solvers in SUNDIALS

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

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

For reasons related to installation, the names of the files involved in these generic solvers begin with the prefix `sundials_`. But despite this, each of the solvers is in fact generic, in that it is usable completely independently of SUNDIALS.

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

9.1 The DENSE module

Relative to the SUNDIALS *srcdir*, the files comprising the DENSE generic linear solver are as follows:

- header files (located in *srcdir*`/include/sundials`)
  `sundials_dense.h` `sundials_smalldense.h`
  `sundials_types.h` `sundials_math.h` `sundials_config.h`
- source files (located in *srcdir*`/src/sundials`)
  `sundials_dense.c` `sundials_smalldense.c` `sundials_math.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are relevant to the DENSE package by itself (see §A.1.3 for details):

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  ```
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
  ```
- (optional) use of generic math functions: `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the `ABS` macro and `RAbs` function.

The eight files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into a DENSE library or into a larger user code.

9.1.1 Type DenseMat

The type `DenseMat` is defined to be a pointer to a structure with the number of rows, number of columns, and a data field:

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

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

9.1.2 Accessor Macros

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

- `DENSE_ELEM`

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

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

- `DENSE_COL`

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

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

9.1.3 Functions

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

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

9.1.4 Small Dense Matrix Functions

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

- `denalloc`

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

- `denallocpiv`

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

- `denGETRF`

  `denGETRF(a,m,n,p)` factors the `m` by `n` dense matrix `a`, using Gaussian elimination with row pivoting. It overwrites the elements of `a` with its LU factors and keeps track of the pivot rows chosen in the pivot array `p`.

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

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

     `denGETRF` returns 0 if successful. Otherwise it encountered a zero diagonal element during the factorization, indicating that the matrix `a` does not have full column rank. In this case it returns the column index (numbered from one) at which it encountered the zero.

- `denGETRS`

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

- `denzero`

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

- `dencopy`

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

- `denscale`

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

- `denaddI`

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

- `denfreepiv`

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

- `denfree`

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

- `denprint`

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

9.2 The BAND module

Relative to the SUNDIALS *srcdir*, the files comprising the BAND generic linear solver are as follows:

- header files (located in *srcdir*`/include/sundials`)
  `sundials_band.h`
  `sundials_types.h` `sundials_math.h` `sundials_config.h`

- source files (located in *srcdir*`/src/sundials`)
  `sundials_band.c` `sundials_math.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are required to use the BAND package by itself (see §A.1.3 for details):

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  ```
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
  ```

- (optional) use of generic math functions:
  ```
  #define SUNDIALS_USE_GENERIC_MATH 1
  ```

The `sundials_types.h` header file defines of the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the `MIN`, `MAX`, and `ABS` macros and `RAbs` function.

The six files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into a BAND library or into a larger user code.

9.2.1 Type BandMat

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

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

The fields in the above structure are:

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

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

  - *data[0]* is a pointer to $(smu+ml+1)*size$ contiguous locations which hold the elements within the band of A
  - *data[j]* is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from $smu-mu$ (to access the uppermost element within the band in the j-th column) to $smu+ml$ (to access the lowest element within the band in the j-th column). Indices from 0 to $smu-mu-1$ give access to extra storage elements required by `BandGBTRF`.
  - *data[j][i-j+smu]* is the (i, j)-th element, $j-mu \leq i \leq j+ml$.

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

See Figure 9.1 for a diagram of the `BandMat` type.

9.2.2 Accessor Macros

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

- `BAND_ELEM`

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

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

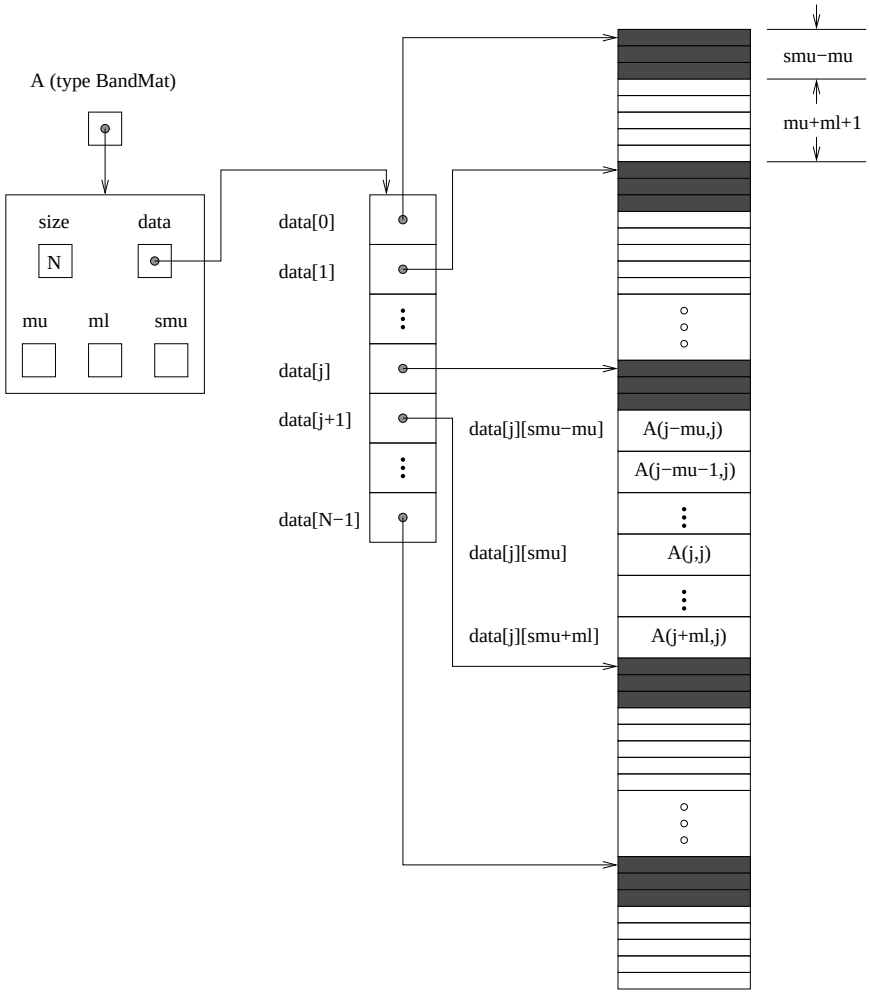


Figure 9.1: Diagram of the storage for a band matrix of type `BandMat`. Here `A` is an $N \times N$ band matrix of type `BandMat` with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to $N-1$ and the (i,j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the `BandGBTRF` and `BandGBTRS` routines.

- BAND_COL

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

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

- BAND_COL_ELEM

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

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

9.2.3 Functions

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

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

9.3 The SPGMR module

The SPGMR package, in the files sundials_spgmr.h and sundials_spgmr.c, includes an implementation of the scaled preconditioned GMRES method. A separate code module, implemented in sundials_iterative.(h,c), contains auxiliary functions that support SPGMR, as well as the other Krylov solvers in SUNDIALS (SPBCG and SPTFQMR). For full details, including usage instructions, see the header files sundials_spgmr.h and sundials_iterative.h.

Relative to the SUNDIALS *srcdir*, the files comprising the SPGMR generic linear solver are as follows:

- header files (located in *srcdir*/include/sundials)
  sundials_spgmr.h sundials_iterative.h sundials_nvector.h
  sundials_types.h sundials_math.h sundials_config.h
- source files (located in *srcdir*/src/sundials)
  sundials_spgmr.c sundials_iterative.c sundials_nvector.c

Only two of the preprocessing directives in the header file sundials_config.h are required to use the SPGMR package by itself (see §A.1.3 for details):

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  ```
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
  ```
- (optional) use of generic math functions:
  ```
  #define SUNDIALS_USE_GENERIC_MATH 1
  ```

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the `MAX` and `ABS` macros and `RAbs` and `RSqrt` functions.

The generic NVECTOR files, `sundials_nvector.(h,c)` are needed for the definition of the generic `N_Vector` type and functions. The NVECTOR functions used by the SPGMR module are: `N_VDotProd`, `N_VLinearSum`, `N_VScale`, `N_VProd`, `N_VDiv`, `N_VConst`, `N_VClone`, `N_VCloneVectorArray`, `N_VDestroy`, and `N_VDestroyVectorArray`.

The SPGMR package can only be used in conjunction with an actual NVECTOR implementation library, such as the NVECTOR_SERIAL or NVECTOR_PARALLEL provided with SUNDIALS.

The nine files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into an SPGMR library or into a larger user code.

9.3.1 Functions

The following functions are available in the SPGMR package:

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

The following functions are available in the support package `sundials_iterative.(h,c)`:

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

9.4 The SPBCG module

The SPBCG package, in the files `sundials_spbcgs.h` and `sundials_spbcgs.c`, includes an implementation of the scaled preconditioned Bi-CGStab method. For full details, including usage instructions, see the file `sundials_spbcgs.h`.

The SPBCG package can only be used in conjunction with an actual NVECTOR implementation library, such as the NVECTOR_SERIAL or NVECTOR_PARALLEL provided with SUNDIALS.

The files needed to use the SPBCG module by itself are the same as for the SPGMR module, with `sundials_spbcgs.(h,c)` replacing `sundials_spgmr.(h,c)`.

9.4.1 Functions

The following functions are available in the SPBCG package:

- `SpbcgMalloc`: allocation of memory for `SpbcgSolve`;
- `SpbcgSolve`: solution of $Ax = b$ by the SPBCG method;
- `SpbcgFree`: free memory allocated by `SpbcgMalloc`.

9.5 The SPTFQMR module

The SPTFQMR package, in the files `sundials_sptfqmr.h` and `sundials_sptfqmr.c`, includes an implementation of the scaled preconditioned TFQMR method. For full details, including usage instructions, see the file `sundials_sptfqmr.h`.

The SPTFQMR package can only be used in conjunction with an actual NVECTOR implementation library, such as the NVECTOR_SERIAL or NVECTOR_PARALLEL provided with SUNDIALS.

The files needed to use the SPTFQMR module by itself are the same as for the SPGMR module, with `sundials_sptfqmr.(h,c)` replacing `sundials_spgmr.(h,c)`.

9.5.1 Functions

The following functions are available in the SPTFQMR package:

- `SptfqmrMalloc`: allocation of memory for `SptfqmrSolve`;
- `SptfqmrSolve`: solution of $Ax = b$ by the SPTFQMR method;
- `SptfqmrFree`: free memory allocated by `SptfqmrMalloc`.

Appendix A

IDAS Installation Procedure

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

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`, `ida`, or `kinsol`, and `x.y.z` represents the version number (of the SUNDIALS suite or of the individual solver).

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

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

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

A few observations:

- starting with version `2.5.0`, two installation methods are provided: in addition to the previous autotools-based method, SUNDIALS now provides a method based on CMake. Both approaches are described in detail in the following sections.
- The installation of examples follows a new philosophy: If examples are enabled by the user, "make" will build all pertinent examples together with the SUNDIALS libraries, but "make install" will export (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 (and which can therefore be used as "templates" for your own problems). The configure script will install makefiles. CMake installs `CMakeLists.txt` files and also (as an option available only under Unix/Linux) makefiles. Note that the exported example makefiles are generated from templates (also included in the attached tarball).
- No matter the installtion procedure, even if generation of shared libraries is enabled, only static libraries are created for the FCMIX modules (due to the use of fixed names for the Fortran user-provided subroutines, FCMIX shared libraries would result in "undefined symbol" errors at link time).
- Since the CMake based build system is fairly new and has only been tested under Linux and Windows (and there only for VC8 and the Intel Compiler) any feedback on how it works on other platforms is appreciated

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

- *srcdir*

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

- *builddir*

  is the directory under which SUNDIALS is built. This can be the same as *srcdir*, although out-of-source builds are recommended.

- *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.

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

A.1 Autotools-based installation

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

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

from within *srcdir*.

The installation steps for SUNDIALS can be as simple as

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

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

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

```
% make clean
```

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

```
% make distclean
```

A.1.1 Configuration options

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

General options

`--prefix=PREFIX`

Location for architecture-independent files.

Default: `PREFIX=`/usr/local

`--exec-prefix=EPREFIX`

Location for architecture-dependent files.

Default: `EPREFIX=`/usr/local

`--includedir=DIR`

Alternate location for installation of header files.

Default: `DIR=PREFIX/include`

`--libdir=DIR`

Alternate location for installation of libraries.

Default: `DIR=EPREFIX/lib`

`--disable-`*solver*

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

`--enable-examples`

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

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

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

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

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

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

`--with-examples-instdir=EXINSTDIR`

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

Default: `DIR=EPREFIX/examples`

`--with-cppflags=ARG`

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

`--with-cflags=ARG`

Specify additional C compilation flags.

`--with-ldflags=ARG`

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

`--with-libs=ARG`

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

`--with-precision=ARG`

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

Default: `ARG=double`

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

Options for Fortran support

`--disable-fcmix`

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

`--with-fflags=ARG`

Specify additional FORTRAN compilation flags.

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

`--with-f77underscore=ARG`

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

Default: `ARG=one`

`--with-f77case=ARG`

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

Default: `ARG=lower`

Options for MPI support

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

`--disable-mpi`

Using this option will completely disable MPI support.

`--with-mpicc=ARG`

`--with-mpif77=ARG`

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

`--with-mpi-root=MPIDIR`

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

`--with-mpi-incdir=INCDIR`

`--with-mpi-libdir=LIBDIR`

`--with-mpi-libs=LIBS`

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

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

Default: `INCDIR=MPIDIR/include` and `LIBDIR=MPIDIR/lib`

`--with-mpi-flags=ARG`

Specify additional MPI-specific flags.

Options for library support

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

`--enable-shared`

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

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

Options for Matlab support

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

`--enable-sundialsTB`

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

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

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

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

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

`--with-sundialsTB-instdir=STBINSTDIR`

Alternate location for the installed SUNDIALSTB toolbox (valid only if SUNDIALSTB is enabled). As for the example programs, installation of SUNDIALSTB can be completely disabled by issuing

STBINSTDIR=no (in case building the toolbox is desired but its installtion will be done manually afterwards). Otherwise, all required SUNDIALSTB files will be installed under the directory STBINSTDIR/sundialsTB.

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

--with-matlab=MATLAB

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

--with-mexopts=ARG

Specify the mex options file to be used.

Default: Standard Matlab mex options file.

--with-mexflags=ARG

Specify the mex compiler flags to be used.

Default: ARG=-O

--with-mexldadd=ARG

Specify additional mex linker flags.

Default: none

Environment variables

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

CC

F77

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

CFLAGS

FFLAGS

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

A.1.2 Configuration examples

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

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

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

To disable installation of the examples, use:

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

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

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

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

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

A.1.3 Building SUNDIALS without the configure script

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

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

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

A sample header file that, appropriately modified, can be used as `sundials_config.h` (otherwise created automatically by the `configure` script) is provided below.

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

```

/* Define SUNDIALS version number */
#define SUNDIALS_PACKAGE_VERSION "2.3.0"

/* FCMIX: Define Fortran name-mangling macro
 * Depending on the inferred scheme, one of the following
 * six macros will be defined:
 *     #define F77_FUNC(name,NAME) name
 *     #define F77_FUNC(name,NAME) name ## _
 *     #define F77_FUNC(name,NAME) name ## __
 *     #define F77_FUNC(name,NAME) NAME
 *     #define F77_FUNC(name,NAME) NAME ## _
 *     #define F77_FUNC(name,NAME) NAME ## __
 */
#define F77_FUNC(name,NAME) name ## _
#define F77_FUNC_(name,NAME) name ## _

/* Define precision of SUNDIALS data type 'realtype'
 * Depending on the precision level, one of the following
 * three macros will be defined:
 *     #define SUNDIALS_SINGLE_PRECISION 1
 *     #define SUNDIALS_DOUBLE_PRECISION 1
 *     #define SUNDIALS_EXTENDED_PRECISION 1
 */
#define SUNDIALS_DOUBLE_PRECISION 1

/* Use generic math functions
 * If it was decided that generic math functions can be used, then
 *     #define SUNDIALS_USE_GENERIC_MATH 1
 * otherwise
 *     #define SUNDIALS_USE_GENERIC_MATH 0
 */
#define SUNDIALS_USE_GENERIC_MATH 1

/* FNVECTOR: Allow user to specify different MPI communicator
 * If it was found that the MPI implementation supports MPI_Comm_f2c, then
 *      #define SUNDIALS_MPI_COMM_F2C 1
 * otherwise
 *      #define SUNDIALS_MPI_COMM_F2C 0
 */
#define SUNDIALS_MPI_COMM_F2C 1
```

The various preprocessor macros defined within `sundials_config.h` have the following uses:

- Precision of the SUNDIALS `realtype` type

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

- Use of generic math functions

  If `SUNDIALS_USE_GENERIC_MATH` is defined, then the functions in `sundials_math.(h,c)` will use the `pow`, `sqrt`, `fabs`, and `exp` functions from the standard math library (see `math.h`), regardless of the definition of `realtype`. Otherwise, if `realtype` is defined to be an alias for the `float`

C-type, then SUNDIALS will use `powf`, `sqrtf`, `fabsf`, and `expf`. If `realtype` is instead defined to be a synonym for the `long double` C-type, then `powl`, `sqrtl`, `fabsl`, and `expl` will be used.

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

- FORTRAN name-mangling scheme

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

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

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

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

  in the `sundials_config.h` header file.

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

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

A.2 CMake-based installation

Using CMake as a build system for the SUNDIALS library has the advantage that GUI based build configuration is possible. Also build files for Windows development environments can be easily generated. On the Windows platform compilers such as the Borland C++ compiler or Visual C++ compiler are natively supported.

The installation options are very similar to the options mentioned above. Note, however, that CMake may not support all features and plattforms that are supported by the autotools build system.

A.2.1 Prerequisites

You may need to get CMake if it isn't available on your system already. In order to use the CMake build system, you need a fairly recent CMake version. You can download it from `http://www.cmake.org` `www.cmake.org`. If you are building cmake from sources on Linux/Unix, make sure to have curses (including development libraries) installed, so that ccmake gets compiled as well. Build instructions for cmake (only necessary for *nix systems) can be found on the CMake page. Once you have CMake installed, you should be able to use `CMakeSetup.exe` on Windows and `ccmake` on Linux/Unix.

A.2.2 Building on Linux/Unix

We assume that the SUNDIALS sources have been uncompressed in *srcdir*. Next, create the *builddir*, for example inside *srcdir* (you can also create the build directory anywhere else, simply substitute the `..` in the next command with the path to *srcdir*). Next change into that directory and run `ccmake`.

```
% mkdir build
% cd build
% ccmake ..
```

You should now see the ccmake curses interface. Press 'c' to configure your build with the default options. You should see an option listing like in the following screenshot. [[¡image ccmake1.png size="medium"]] (If you don't have curses on your system and cannot use ccmake, you can configure cmake with command line options very similar to ./configure of the autotools. You can read about this on the cmake webpage.)

In the dialog you can adjust the build options. For details see the options above in the autotools section. To adjust advanced options press 't' to show all the options and settings CMake offers.

After adjusting some options, for instance enabling the examples by turning `ENABLE_EXAMPLES` to `ON`, you need to press 'c' again. Depending on the options, you will see new options at the top of the list, marked with a star. After adjusting the new options, press 'c' again. Once all options have been set, you can press 'g' to generate the make files.

Now you can build and install the sundials library:

```
% make
% make install
```

A.2.3 Building on Windows

The first part in this section is a step-by-step compilation and usage guide for Visual Studio users (namely Visual Studio 2005 aka VC8). In the second part we describe how to build SUNDIALS using Mingw32 GCC or other compilers, which is very similar to the method used on Linux/Unix system.

The first common part is the extraction of the archives, which, due to a lack of native support of `tag.gz` files, is not quite as trivial as on Linux.

Extracting the archives

Begin by placing the SUNDIALS tarball in one directory on your harddrive, for instance C:\sundials.

If you simply double-click the files, chances are that you will get the helpful "unknown file type" window. It is futile searching for a program on a standard install, `tar.gz` files are not supported natively on Windows. Instead, download one of the many zip utilities (e.g. the open source tool `7-zip` available from `http://www.7-zip.org/`, which will be used below.)

Use the "Extract here" option to uncompress the `sundials-x.y.z.tar.gz` archive. Use the "Extract here" option again on the `sundials-x.y.z.tar` file and all the files will be extracted into a new `sundials-x.y.z` subdirectory (the *srcdir*).

Building and using Sundials with Visual Studio

Configuration of the SUNDIALS build. Now you need to open a console window. Note that, in order for the correct path variables to be set, you need to open the command line via the link provided as Visual Studio start menu option.

Inside that command window change into the source directory and run `CMakeSetup.exe`. In case `CMakeSetup.exe` does not start the CMake configuration utility, check that the CMake bin directory is in your path environment variable.

Now you should see the CMake configuration dialog. First select the *srcdir* `sundials-x.y.z` as the source directory. Then copy this directory over in the "build" directory **and** append the directory name 'build'.

Now you can start the configuration process by clicking on 'configure'. You will be prompted with a number of make file generator options. Select 'Visual Studio 8 2005" (or select a different make file generator, if you work with a different version). The `CMakeSetup.exe` tool will now verify the build tool chain and determine the compilation options. Once completed, you will see a list of build options to select. (If the build chain detection fails, there is probably some problem with the VC8 installation or some missing/wrong environmental variables).

You can now adjust the listed options to your needs. Once you are finished, simply click 'configure' again. All options, that are fully configured will turn grey, indicating that you are ready to generate the makefiles. For instance, you can turn compilation of the examples on.

Finally press 'OK' to generate the VC project files. The content of the build directory will now show the following files:

MORE HERE...

Building the library (and examples). Now simply double click on the `ALL_BUILD.vcproj` project file and Visual Studio will open with a (long) list of generated projects.

Simply build the whole solution and after a while the build log will confirm that everything went all right.

Having the libraries compiled, now it is fairly easy to use them in your own projects.

Using the Sundials library. Three tasks are involved in order to use a solver of the Sundials library in your own code:

- set include directories in your project
- add required SUNDIALS library project files to your solution
- set the SUNDIALS libraries as Dependencies

Here's a small step-by-step example which illustrates the process of building a CVODE based application using one of the included examples.

- First close the current solution and create a new Win32 console project. In the "New project" wizard set the following Application options to create an empty project.
- We need the CVODE and NVECTOR_SERIAL libraries. Use the "Add-¿Existing project..." option from the solution context menu and select the project files

  `C:\Sundials\sundials-2.5.0\build\src\cvode\sundials_cvode_static.vcproj`

  and

  `C:\Sundials\sundials-2.3.0\build\src\nvec_ser\sundials_nvecserial_static.vcproj`

- Also add the example program

  `C:\Sundials\sundials-2.3.0\examples\cvode\serial\cvbanx.c`

  to your empty console project.

- Now we still need to adjust some of our project settings, because compilation of the example project still fails because the Sundials headers cannot be found. Open the project properties and go to the C++ options.
- Here you need to select the include directory in the SUNDIALS source directory **and** the include directory in the SUNDIALS build directory.
- Now also verify that the linker options show that "Link Library Dependencies" is enabled.
- Last but not least set the project dependencies. Open "Project Dependencies" from the context menu of your example project, and check both SUNDIALS libraries.
- Finally, compile your project and run it.

Building Sundials with Mingw32-GCC or other compilers

This requires a console window with a correctly set path variable for `mingw`.

First create a subdirectory 'build' inside the sundials source directory (you can also create the build directory anywhere else). Next run `CMakeSetup.exe` with the `sundials` source directory as command line argument. Alternatively, simply run `CMakeSetup.exe` and select the source and build directories from the directory combo boxes.

When pressing 'configure' for the first time you will be prompted with a choice of build systems. Select the compiler/build system you want to use and press 'OK'.

In the dialog you can now adjust the build options. For details see the options above in the autotools section. Check the "Show advanced values" checkbox if you want to set advanced options.

After adjusting all options press configure again (all lines should become grey) and OK to generate make files and quit the GUI.

Now you can run your build systems make tool, e.g. mingw32-make for mingw, make for Borland C++. To use the Intel Compiler you need to create nmake build files and you must start the console window from the provided start menu entry.

A.3 Installed libraries and exported header files

Using the standard SUNDIALS build system, the command

```
% make install
```

will install the libraries under *libdir* and the public header files under *includedir*. The default values for these directories are *instdir*`/lib` and *instdir*`/include`, respectively, but can be changed using the configure script options `--prefix`, `--exec-prefix`, `--includedir` and `--libdir` (see §A.1.1). For example, a global installation of SUNDIALS on a *NIX system could be accomplished using

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

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

The installed libraries and exported header files are listed for reference in Table A.1. The file extension *.lib* is typically `.so` for shared libraries and `.a` for static libraries (see *Options for library support* for additional details). Note that, in Table A.1, 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*`/sundials` directory since they are explicitly included by the appropriate solver header files (*e.g.*, `cvode_dense.h` includes `sundials_dense.h`). However, it is both legal and safe to do so (*e.g.*, the functions declared in `sundials_smalldense.h` could be used in building a preconditioner).

Table A.1: SUNDIALS libraries and header files

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

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_NORMAL_TSTOP	3	Solver returns at specified output time, but does not proceed past the specified stopping time.
IDA_SIMULTANEOUS	1	Simultaneous corrector forward sensitivity method.
IDA_STAGGERED	2	Staggered corrector forward sensitivity method.
IDA_CENTERED	1	Central difference quotient approximation (2nd order) of the sensitivity RHS.
IDA_FORWARD	2	Forward difference quotient approximation (1st order) of the sensitivity RHS.
IDA_YA_YDP_INIT	1	Compute y_a and y'_d, given y_d.
IDA_Y_INIT	2	Compute y, given y'.

IDAA **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.
PREC_RIGHT	2	Preconditioning on the right.
PREC_BOTH	3	Preconditioning on both the left and the right.
MODIFIED_GS	1	Use modified Gram-Schmidt procedure.
CLASSICAL_GS	2	Use classical Gram-Schmidt procedure.

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 occured.
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 IDAMalloc.
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 s outside the last step taken.
IDA_BAD_DKY	-26	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.

IDAA **adjoint solver module**

IDA_NO_ADJ	-100	The combined forward-backward problem has not been initialized.
IDA_BAD_TBO	-101	The desired output for backward problem is outside the interval over which the forward problem was solved.
IDA_REIFWD_FAIL	-102	No checkpoint is available for this hot start.
IDA_FWD_FAIL	-103	IDASolveB failed because IDASolve was unable to store data between two consecutive checkpoints.
IDA_GETY_BADT	-104	Wrong time in interpolation function.
IDA_NO_BCK	-105	No backward problem was specified.
IDA_NO_FWD	-106	IDASolveF has not been previously called.

IDADLS **linear solver modules**

IDADIRECT_SUCCESS	0	Successful function return.
IDADIRECT_MEM_NULL	-1	The ida_mem argument was NULL.
IDADIRECT_LMEM_NULL	-2	The IDADLS linear solver has not been initialized.
IDADIRECT_ILL_INPUT	-3	The IDADLS solver is not compatible with the current NVECTOR module.
IDADIRECT_MEM_FAIL	-4	A memory allocation request failed.
IDADIRECT_JACFUNC_UNRECVR	-5	The Jacobian function failed in an unrecoverable manner.
IDADIRECT_JACFUNC_RECVR	-6	The Jacobian function had a recoverable error.

IDASPILS **linear solver modules**

IDASPILS_SUCCESS	0	Successful function return.
IDASPILS_MEM_NULL	-1	The ida_mem argument was NULL.
IDASPILS_LMEM_NULL	-2	The linear solver has not been initialized.
IDASPILS_ILL_INPUT	-3	The 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	-100	The combined forward-backward problem has not been initialized.

SPGMR generic linear solver module

SPGMR_SUCCESS	0	Converged.
SPGMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPGMR_CONV_FAIL	2	Failure to converge.
SPGMR_QRFACT_FAIL	3	A singular matrix was found during the QR factorization.
SPGMR_PSOLVE_FAIL_REC	4	The preconditioner solve function failed recoverably.
SPGMR_ATIMES_FAIL_REC	5	The Jacobian-times-vector function failed recoverably.
SPGMR_PSET_FAIL_REC	6	The preconditioner setup function failed recoverably.
SPGMR_MEM_NULL	-1	The SPGMR memory is NULL
SPGMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPGMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPGMR_GS_FAIL	-4	Failure in the Gram-Schmidt procedure.
SPGMR_QRSOL_FAIL	-5	The matrix R was found to be singular during the QR solve phase.
SPGMR_PSET_FAIL_UNREC	-6	The preconditioner setup function failed unrecoverably.

SPBCG generic linear solver module

SPBCG_SUCCESS	0	Converged.
SPBCG_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPBCG_CONV_FAIL	2	Failure to converge.
SPBCG_PSOLVE_FAIL_REC	3	The preconditioner solve function failed recoverably.
SPBCG_ATIMES_FAIL_REC	4	The Jacobian-times-vector function failed recoverably.
SPBCG_PSET_FAIL_REC	5	The preconditioner setup function failed recoverably.
SPBCG_MEM_NULL	-1	The SPBCG memory is NULL
SPBCG_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPBCG_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPBCG_PSET_FAIL_UNREC	-4	The preconditioner setup function failed unrecoverably.

SPTFQMR generic linear solver module

SPTFQMR_SUCCESS	0	Converged.
SPTFQMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPTFQMR_CONV_FAIL	2	Failure to converge.
SPTFQMR_PSOLVE_FAIL_REC	3	The preconditioner solve function failed recoverably.
SPTFQMR_ATIMES_FAIL_REC	4	The Jacobian-times-vector function failed recoverably.
SPTFQMR_PSET_FAIL_REC	5	The preconditioner setup function failed recoverably.
SPTFQMR_MEM_NULL	-1	The SPTFQMR memory is NULL
SPTFQMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed.
SPTFQMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPTFQMR_PSET_FAIL_UNREC	-4	The preconditioner setup function failed unrecoverably.

Bibliography

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

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

[3] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems. *SIAM J. Sci. Comput.*, 15:1467–1488, 1994.

[4] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Consistent Initial Condition Calculation for Differential-Algebraic Systems. *SIAM J. Sci. Comput.*, 19:1495–1512, 1998.

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

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

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

[8] 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.

[9] 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.

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

[11] A. M. Collier, A. C. Hindmarsh, R. Serban, and C.S. Woodward. User Documentation for KINSOL v2.4.0. Technical Report UCRL-SM-208116, LLNL, 2006.

[12] 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.

[13] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.

[14] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.

[15] 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.

[16] A. C. Hindmarsh and R. Serban. Example Programs for IDA v2.4.0. Technical Report UCRL-SM-208113, LLNL, 2005.

[17] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v2.4.0. Technical Report UCRL-SM-208108, LLNL, 2005.

[18] 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.

[19] 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.

[20] T. Maly and L. R. Petzold. Numerical Methods and Software for Sensitivity Analysis of Differential-Algebraic Systems. *Applied Numerical Mathematics*, 20:57–79, 1997.

[21] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.

[22] H. A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.

Index