SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers

ALAN C. HINDMARSH, PETER N. BROWN, KEITH E. GRANT, STEVEN L. LEE, RADU SERBAN, DAN E. SHUMAKER, and CAROL S. WOODWARD
Lawrence Livermore National Laboratory

SUNDIALS is a suite of advanced computational codes for solving large-scale problems that can be modeled as a system of nonlinear algebraic equations, or as initial-value problems in ordinary differential or differential-algebraic equations. The basic versions of these codes are called KINSOL, CVODE, and IDA, respectively. The codes are written in ANSI standard C and are suitable for either serial or parallel machine environments. Common and notable features of these codes include inexact Newton-Krylov methods for solving large-scale nonlinear systems; linear multistep methods for time-dependent problems; a highly modular structure to allow incorporation of different preconditioning and/or linear solver methods; and clear interfaces allowing for users to provide their own data structures underneath the solvers. We describe the current capabilities of the codes, along with some of the algorithms and heuristics used to achieve efficiency and robustness. We also describe how the codes stem from previous and widely used Fortran 77 solvers, and how the codes have been augmented with forward and adjoint methods for carrying out first-order sensitivity analysis with respect to model parameters or initial conditions.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]; G.1.7 [**Numerical Analysis**]: Ordinary Differential Equations—*Differential-algebraic equations*; *multistep and multivalue methods*; *stiff equations*; G.1.5 [**Numerical Analysis**]: Roots of Nonlinear Equations—*Iterative methods*; *convergence*

General Terms: Algorithms, Design

Additional Key Words and Phrases: ODEs, DAEs, nonlinear systems, sensitivity analysis

1. INTRODUCTION

With the ever-increasing capabilities of modern computers, simulation code developers are challenged to develop fast and robust software capable of

This work was performed 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.
Authors' addresses: Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94551; email: {alanh,pnbrown,keg,slee,radu,shumaker,cswoodward}@llnl.gov.

solving problems with increasingly higher resolutions and modeling more complex physical phenomena. At the heart of many numerical simulation codes lie systems of nonlinear algebraic or time-dependent equations, and simulation scientists continue to require efficient solvers for these systems.

To meet this need, Lawrence Livermore National Laboratory (LLNL) has a long history of research and development in ordinary differential equation (ODE) methods and software, as well as closely related areas, with emphasis on applications to partial differential equations (PDEs). Among the popular Fortran 77 solvers written at LLNL are the following:

—VODE: a solver for ODE initial-value problems for stiff/nonstiff systems, with direct solution of linear systems, by Brown, Byrne, and Hindmarsh [Brown et al. 1989].
—VODPK: a variant of VODE with preconditioned Krylov (GMRES iteration [Saad and Schultz 1986]) solution of the linear systems in place of direct methods, by Brown, Byrne, and Hindmarsh [Byrne 1992].
—NKSOL: a Newton-Krylov (GMRES) solver for nonlinear algebraic systems, by Brown and Saad [Brown and Saad 1990].
—DASPK: a solver for differential-algebraic equation (DAE) systems (a variant of DASSL) with both direct and preconditioned Krylov solution methods for the linear systems, by Brown, Hindmarsh, and Petzold [Brown et al. 1994].

Starting in 1993, the push to solve large systems in parallel motivated work to write or rewrite solvers in C. Moving to the C language was done to exploit features of C not present in Fortran 77 while using languages with stable compilers (F90/95 were not yet stable when this work started); achieve a more object-oriented design; facilitate the use of the codes with other object-oriented codes being written in C and C++; maximize the reuse of code modules; and facilitate the extension from a serial to a parallel implementation. The first result of the C effort was CVODE. This code was a rewrite in ANSI standard C of the VODE and VODPK solvers combined, for serial machines [Cohen and Hindmarsh 1994, 1996]. The next result of this effort was PVODE, a parallel extension of CVODE [Byrne and Hindmarsh 1998, 1999]. Similar rewrites of NKSOL and DASPK followed, using the same general design as CVODE and PVODE. The resulting solvers are called KINSOL and IDA, respectively. More recently, we have merged the PVODE and CVODE codes into a single solver, CVODE.

The main numerical operations performed in these codes are operations on data vectors, and the codes have been written in terms of interfaces to these vector operations. The result of this design is that users can relatively easily provide their own data structures to the solvers by telling the solver about their structures and providing the required operations on them. The codes also come with default vector structures with predefined operation implementations for both serial and distributed memory parallel environments in case a user prefers to not supply their own structures. In addition, all parallelism is contained within specific vector operations (norms, dot products, etc.). No other operations within the solvers require knowledge of parallelism. Thus, using a

solver in parallel consists of using a parallel vector implementation, either the one provided with SUNDIALS, or the user's own parallel vector structure, underneath the solver. Hence, we no longer make a distinction between parallel and serial versions of the codes.

These codes have been combined into the core of SUNDIALS, the SUite of Nonlinear and DIfferential/Algebraic equation Solvers. This suite, consisting of CVODE, KINSOL, and IDA (along with current and future augmentations to include forward and adjoint sensitivity analysis capabilities), was implemented with the goal of providing robust time integrators and nonlinear solvers that can easily be incorporated into existing simulation codes. The primary design goals were to require minimal information from the user, allow users to easily supply their own data structures underneath the solvers, and allow for easy incorporation of user-supplied linear solvers and preconditioners.

As simulations have increased in size and complexity, the relationship between computed results and problem parameters has become harder to establish. Scientists have a greater need to understand answers to questions like the following. Which model parameters are most influential? How can parameters be adjusted to better match experimental data? What is the uncertainty in solutions given uncertainty in data? Sensitivity analysis provides information about the relationship between simulation results and model data, which can be critical to answering these questions. In addition, for a given parameter, sensitivities can be computed in a modest multiple of the computing time of the simulation itself.

SUNDIALS is being expanded to include forward and adjoint sensitivity versions of the solvers. The first of these, CVODES, is complete. A brief description of the strategy for adding sensitivity analysis in a way that respects the user interfaces of the SUNDIALS codes is contained in this article, and a more thorough description of the CVODES package (which is distinct from but built on the same core code as CVODE) is contained in a companion article [Serban and Hindmarsh 2005]. The second sensitivity solver will be IDAS and is currently under development. An extension, to be called KINSOLS, to KINSOL for sensitivity analysis will be completed if need arises.

The rest of this article is organized as follows. In Section 2, the algorithms in the three core solvers of SUNDIALS are presented. We have attempted to identify many of the heuristics related to stopping criteria and finite-difference parameter selection where appropriate, as these items can sometimes affect algorithm performance significantly. In Section 3, the preconditioning packages supplied with SUNDIALS are described. Section 4 overviews the CVODES package and strategies for adding sensitivity capabilities to the codes. Sections 5 and 6 describe the organization of the codes within the suite and the philosophy of the user interface. Availability of the codes is given in Section 7. Finally, comments on applications of SUNDIALS, concluding remarks, and indications of future development are contained in the last section.

2. THE BASIC SOLVERS

In this section we overview each of the three core solvers in SUNDIALS, giving a detailed summary of the methods and algorithms used in each. Although many

of the algorithmic features are common to the three codes (e.g., finite-difference Jacobian-vector approximations and stopping criteria), we still outline them with respect to each package, as some of the details in their implementation are different.

All three of the solver descriptions below involve a number of heuristic rules, safety factors, and the like. We do not attempt to justify or explain these heuristics here, for two reasons. First, most of them are largely arbitrary, and have little or no solid mathematical basis. For example, a safety factor less than 1 is needed in many places (such as in step size selection based on estimated local error), but we know of no good argument for preferring any particular value over another. (In a few cases, some performance testing was done to optimize heuristic parameters, but this is rare.) The second reason is that all of these heuristics were inherited from earlier solvers, and the documentation of those solvers includes some discussion of the heuristics used. Each of the three descriptions cites the appropriate earlier literature.

2.1 CVODE

CVODE solves ODE initial value problems in real N-space. We write such problems in the form

$$\dot{y} = f(t, y), \quad y(t_0) = y_0, \tag{1}$$

where $y \in \mathbf{R}^N$. Here we use $\dot{y}$ to denote dy/dt. While we use t to denote the independent variable, and usually this is time, it certainly need not be. CVODE solves both stiff and nonstiff systems. Roughly speaking, stiffness is characterized by the presence of at least one rapidly damped mode, whose time constant is small compared to the time scale of the solution itself. (See Hairer and Wanner [1991] for more on stiffness.)

The methods used in CVODE are variable-order, variable-step multistep methods, based on formulas of the form

$$\sum_{i=0}^{K_1} \alpha_{n,i}\, y_{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i}\, \dot{y}_{n-i} = 0.$$

Here the y_n are computed approximations to $y(t_n)$, and $h_n = t_n - t_{n-1}$ is the step size. The user of CVODE must choose appropriately from one of two families of multistep formulas. For nonstiff problems, CVODE includes the Adams-Moulton formulas, characterized by $K_1 = 1$ and $K_2 = q$ above, where the order q varies between 1 and 12. For stiff problems, CVODE includes the Backward Differentiation Formulas (BDFs) in so-called *fixed-leading coefficient form*, given by $K_1 = q$ and $K_2 = 0$, with order q varying between 1 and 5. The coefficients are uniquely determined by the method type, its order, the recent history of the step sizes (the last q values), and the normalization $\alpha_{n,0} = -1$. See Byrne and Hindmarsh [1975] and Jackson and Sacks-Davis [1980].

For either choice of formula, the nonlinear system

$$G(y_n) \equiv y_n - h_n \beta_{n,0} f(t_n, y_n) - a_n = 0, \tag{2}$$

where $a_n \equiv \sum_{i>0} (\alpha_{n,i}\, y_{n-i} + h_n \beta_{n,i}\, \dot{y}_{n-i})$, must be solved (approximately) at each integration step. For this, CVODE offers the choice of either *functional iteration*,

suitable only for nonstiff systems, and various versions of *Newton iteration*. If we denote the Newton iterates by $y_{n,m}$, then functional iteration, given by

$$y_{n,m+1} = h_n \beta_{n,0} f(t_n, y_{n,m}) + a_n,$$

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

$$M[y_{n,m+1} - y_{n,m}] = -G(y_{n,m}),$$

in which

$$M \approx I - \gamma J, \quad J = \partial f / \partial y, \quad \text{and} \quad \gamma = h_n \beta_{n,0}. \tag{3}$$

In either case, the initial guess for the iteration is a predicted value $y_{n,0}$ computed explicitly from the available history data (the last $q + 1$ computed values of y or $\dot{y}$). For the Newton corrections, CVODE provides a choice of four methods:

—a dense direct solver (serial version only),
—a band direct solver (serial version only),
—a diagonal approximate Jacobian solver [Radhakrishnan and Hindmarsh 1993], or
—SPGMR = Scaled Preconditioned GMRES, without restarts [Brown and Hindmarsh 1989].

(By *serial version* we mean the CVODE solver with the serial NVECTOR module attached.)

For large stiff systems, where direct methods are not feasible, the combination of a BDF integrator with the SPGMR algorithm yields a powerful tool because it combines established methods for stiff integration, nonlinear iteration, and Krylov (linear) iteration with a problem-specific treatment of the dominant source of stiffness, in the form of the user-supplied preconditioner matrix [Brown and Hindmarsh 1989].

In the process of controlling errors at various levels, CVODE uses a weighted root-mean-square norm, denoted $\| \cdot \|_{\mathrm{WRMS}}$, for all error-like quantities:

$$\|v\|_{\mathrm{WRMS}} = \sqrt{N^{-1} \sum_{1}^{N} (v_i / W_i)^2}. \tag{4}$$

The weights W_i are based on the current solution (with components denoted y^i), and on the relative tolerance RTOL and absolute tolerances ATOL_i input by the user, namely,

$$W_i = \mathrm{RTOL} \cdot |y^i| + \mathrm{ATOL}_i. \tag{5}$$

Because W_i represents a tolerance in the component y^i, a vector representing a perturbation in y and having norm of 1 is regarded as "small." For brevity, we will usually drop the subscript WRMS on norms in what follows.

In the cases of a direct solver (dense, band, or diagonal), the iteration is a Modified Newton iteration, in that the iteration matrix M is fixed throughout the nonlinear iterations. However, for SPGMR, it is an Inexact Newton

iteration, in which M is applied in a matrix-free manner, with matrix-vector products Jv obtained by either difference quotients or a user-supplied routine. The matrix M (direct cases) or preconditioner matrix P (SPGMR case) is updated as infrequently as possible, to balance the high costs of matrix operations against other costs. Specifically, this matrix update occurs when

—starting the problem,
—more than 20 time steps have been taken since the last update,
—the current value of γ and its value at the last update ($\bar{\gamma}$) satisfy $|\gamma/\bar{\gamma} - 1| > 0.3$,
—a convergence failure just occurred, or
—an error test failure just occurred.

When forced by a convergence failure, an update of M or P may or may not involve a reevaluation of J (in M) or of Jacobian data (in P), depending on whether Jacobian error was the likely cause of the failure. More generally, the decision is made to reevaluate J (or instruct the user to reevaluate Jacobian data in P) when

—starting the problem,
—more than 50 steps have been taken since the last evaluation,
—a convergence failure occurred with an outdated matrix, and the value $\bar{\gamma}$ of γ at the last update satisfies $|\gamma/\bar{\gamma} - 1| < 0.2$, or
—a convergence failure occurred that forced a step size reduction.

The stopping test for the Newton iteration is related to the subsequent local error test, with the goal of keeping the nonlinear iteration errors from interfering with local error control. The final computed iterate $y_{n,m}$ will have to satisfy a local error test $\|y_{n,m} - y_{n,0}\| \leq \epsilon$, where ϵ is an error test constant described below. Letting y_n denote the exact solution of (2), we want to ensure that the iteration error $y_n - y_{n,m}$ is small relative to ϵ, specifically that it is less than 0.1ϵ. (The safety factor 0.1 can be changed by the user.) For this, we also estimate the linear convergence rate constant R as follows. We initialize R to 1, and reset $R = 1$ when M or P is updated. After computing a correction $\delta_m = y_{n,m} - y_{n,m-1}$, we update R if $m > 1$ as

$$R \leftarrow \max\{0.3R, \|\delta_m\|/\|\delta_{m-1}\|\},$$

and we use the estimate

$$\|y_n - y_{n,m}\| \approx \|y_{n,m+1} - y_{n,m}\| \approx R\|y_{n,m} - y_{n,m-1}\| = R\|\delta_m\|.$$

Therefore the convergence (stopping) test is

$$R\|\delta_m\| < 0.1\epsilon.$$

We allow at most three iterations (but this limit can be changed by the user). We also declare the iteration to be diverging if any $\|\delta_m\|/\|\delta_{m-1}\| > 2$ with $m > 1$. If the iteration fails to converge with a current J or P, we are forced to reduce the step size, and we replace h_n by $h_n/4$. The integration is halted after a preset

number of convergence failures; the default value of this limit is 10, but this can be changed by the user.

When SPGMR is used to solve the linear system, its errors must also be controlled, and this also involves the local error test constant ϵ. The linear iteration error in the solution vector δ_m is approximated by the preconditioned residual vector. Thus to ensure (or attempt to ensure) that the linear iteration errors do not interfere with the nonlinear error and local integration error controls, we require that the norm of the preconditioned residual in SPGMR is less than $0.05 \cdot (0.1\epsilon)$.

With the direct dense and band methods, the Jacobian may be supplied by a user routine, or approximated by difference quotients, at the user's option. In the latter case, we use the usual approximation

$$J_{ij} = [f^i(t, y + \sigma_j e_j) - f^i(t, y)]/\sigma_j.$$

The increments σ_j are given by

$$\sigma_j = \max\{\sqrt{U}\ |y^j|, \sigma_0 W_j\},$$

where U is the unit roundoff, σ_0 is a dimensionless value (involving the unit roundoff and the norm of $\dot{y}$), and W_j is the error weight defined in (5). In the dense case, this scheme requires N evaluations of f, one for each column of J. In the band case, the columns of J are computed in groups, by the Curtis-Powell-Reid algorithm [Curtis et al. 1974], with the number of f evaluations equal to the bandwidth.

In the case of SPGMR, preconditioning may be used on the left, on the right, or on both, with user-supplied routines for the preconditioning setup and solve operations, and optionally also for the required matrix-vector products Jv. If a routine for Jv is not supplied, these products are computed as

$$Jv = [f(t, y + \sigma v) - f(t, y)]/\sigma. \tag{6}$$

The increment σ is $1/\|v\|$, so that σv has norm 1.

A critical part of CVODE, making it an ODE "solver" rather than just an ODE method, is its control of local error. At every step, the local error is estimated and required to satisfy tolerance conditions, and the step is redone with reduced step size whenever that error test fails. As with any linear multistep method, the local truncation error LTE, at order q and step size h, satisfies an asymptotic relation

$$\text{LTE} = Ch^{q+1}y^{(q+1)} + O(h^{q+2})$$

for some constant C, under mild assumptions on the step sizes. A similar relation holds for the error in the predictor $y_{n,0}$. These are combined to get a relation

$$\text{LTE} = C'[y_n - y_{n,0}] + O(h^{q+2}),$$

where C' is another known constant. The local error test is simply $\|\text{LTE}\| \leq 1$ (recalling that a vector of WRMS norm 1 is considered small). Using $y_n = y_{n,m}$ (the last iterate computed) above, the local error test is performed on the

predictor-corrector difference $\Delta_n \equiv y_{n,m} - y_{n,0}$, and takes the form

$$\|\Delta_n\| \leq \epsilon \equiv 1/|C'|.$$

If this test passes, the step is considered successful. If it fails, the step is rejected and a new step size h' is computed based on the asymptotic behavior of the local error, namely, by the equation

$$(h'/h)^{q+1}\|\Delta_n\| = \epsilon/6.$$

Here 1/6 is a safety factor. A new attempt at the step is made, and the error test repeated. If it fails three times, then the order q is reset to 1 (if it was >1), or (if $q = 1$) the step is restarted from a fresh value of f (discarding all history data). The ratio h'/h is restricted (during the current step only) to be ≤ 0.2 after two error test failures, and to be ≥ 0.1 after three. After seven failures, CVODE returns to the user with a give-up message.

In addition to adjusting the step size to meet the local error test, CVODE periodically adjusts the order, with the goal of maximizing the step size. The integration starts out at order 1 and varies the order dynamically after that. The basic idea is to pick the order q for which a polynomial of order q best fits the discrete data involved in the multistep method. However, if either a convergence failure or an error test failure occurs on any given step, no change in step size or order is allowed on the next step. At the current order q, selecting a new step size is done exactly as when the error test fails, giving a tentative step size ratio

$$h'/h = (\epsilon/6\|\Delta_n\|)^{1/(q+1)} \equiv \eta_q.$$

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

$$h'/h = [1/6\|\mathrm{LTE}_{q-1}\|]^{1/q} \equiv \eta_{q-1}$$

and

$$h'/h = [1/10\|\mathrm{LTE}_{q+1}\|]^{1/(q+2)} \equiv \eta_{q+1}.$$

The new order and step size are then set according to

$$\eta = \max\{\eta_{q-1}, \eta_q, \eta_{q+1}\}\ , \quad h' = \eta h,$$

with q' set to the index achieving the above maximum. However, if we find that $\eta < 1.5$, we do not bother with the change. Also, h'/h is always limited to 10, except on the first step, when it is limited to 10^4.

The various algorithmic features of CVODE described above, as inherited from VODE and VODPK, are documented in Brown et al. [1989], Byrne [1992], and Hindmarsh [2000]. A full description of the usage of CVODE is given in Hindmarsh and Serban [2004a].

There is an important additional part of the CVODE order selection algorithm that is not based on local error, but instead provides protection against

potentially unstable behavior of the BDF methods. At order 1 or 2, the BDF method is A-stable. But at orders 3 to 5 it is not, and the region of instability includes a portion of the left half-plane that is concentrated near the imaginary axis. The size of that region of instability grows as the order increases from 3 to 5. What this means is that when running BDF at these higher orders, if an eigenvalue λ of the system lies close enough to the imaginary axis, the step sizes, h, for which the method is stable are limited (at least according to the linear stability theory) to a set that prevents $h\lambda$ from leaving the stability region. System eigenvalues that are likely to cause this instability are ones that correspond to weakly damped oscillations, such as might arise from a semidiscretized advection-diffusion PDE with advection dominating over diffusion.

CVODE includes an optional algorithm called *STALD* (STAbility Limit Detection), which attempts to detect directly the presence of a stability region boundary that is limiting the step sizes in the presence of a weakly damped oscillation [Hindmarsh 1992]. Working directly with history data that is readily available, if it concludes that the step size is in fact stability-limited, it dictates a reduction in the method order, regardless of the outcome of the error-based algorithm.

STALD has been tested in combination with the VODE solver on linear advection-dominated advection-diffusion problems [Hindmarsh 1995], where it works well. The implementation in CVODE has been successfully tested on linear and nonlinear advection-diffusion problems, among others. The STALD option adds some overhead computational cost to the CVODE solution. In timing tests, these overhead costs have ranged from 2% to 7% of the total, depending on the size and complexity of the problem, with lower relative costs for larger problems. Therefore, it should be activated only when there is reasonable expectation of modes in the user's system for which it is appropriate, together with poor performance at orders 3–5, for no apparent reason, with the option turned off.

Normally, CVODE takes steps until a user-defined output value $t = t_{\text{out}}$ is overtaken, and then it computes $y(t_{\text{out}})$ by interpolation. However, a "one-step" mode option is available, where control returns to the calling program after each step. There are also options to force CVODE not to integrate past a given stopping point $t = t_{\text{stop}}$.

Last, CVODE has been augmented to include a rootfinding feature, whereby the roots of a set of user-defined functions $g_i(t, y)$ can be found while integrating the initial value problem for $y(t)$. The algorithm checks for changes in sign in the g_i over each time step, and when a sign change is found, it homes in on the root(s) with a weighted secant iteration method [Hiebert and Shampine 1980]. (CVODE also checks for exact zeros of the g_i.) The iteration stops when the root is bracketed within a tolerance that is near the roundoff level of t.

2.2 KINSOL

KINSOL solves nonlinear algebraic systems in real space, which we write as

$$F(u) = 0, \quad F : \mathbf{R}^N \to \mathbf{R}^N, \tag{7}$$

given an initial guess u_0. It is a rewrite in C of the Fortran 77 code NKSOL of Brown and Saad [1990].

KINSOL employs the Inexact Newton method developed in Brown and Saad [1990], Brown [1987], and Dembo et al. [1982] and further described in Dennis and Schnabel [1996] and Kelley [1995], resulting in the following iteration:

INEXACT NEWTON ITERATION.

(1) Set $u_0 =$ an initial guess
(2) For $n = 0, 1, 2, \ldots$ until convergence do:
  (a) Approximately solve $J(u_n)\delta_n = -F(u_n)$
  (b) Set $u_{n+1} = u_n + \lambda\delta_n$, $\lambda \leq 1$
  (c) Test for convergence

Here, u_n is the nth iterate to u, and $J(u) = F'(u)$ is the system Jacobian. As this code module is anticipated for use on large systems, only iterative methods are provided to solve the system in step 2(a). These solutions are only approximate. At each stage in the iteration process, a scalar multiple of the approximate solution, δ_n, is added to u_n to produce a new iterate, u_{n+1}. A test for convergence is made before the iteration continues.

The linear iterative method currently implemented is one of the class of Krylov methods, GMRES [Brown and Hindmarsh 1989; Saad and Schultz 1986], provided through the SPGMR module common to all SUNDIALS codes. Use of SPGMR provides a linear solver which, by default, is applied in a matrix-free manner, with matrix-vector products Jv obtained by either finite difference quotients or a user-supplied routine. In the case where finite differences are used, the matrix-vector product $J(u)v$ is approximated by a quotient of the form given in (6), where $f(t, y) = F(y)$ for our nonlinear system, u is the current approximation to a root of (7), and σ is a scalar. The choice of σ is taken from Brown and Saad [1990] and is given by

$$\sigma = \frac{\max\{|u^T v|, \text{typ}u^T |v|\}}{\|v\|_2} \text{sign}(u^T v)\sqrt{U}, \tag{8}$$

where typ$u$ is a vector of typical values for the absolute values of the solution (and can be taken to be inverses of the scale factors given for u as described below), and U is unit roundoff. Convergence of the Newton method is maintained as long as the value of σ remains appropriately small, as shown in Brown [1987].

To the above methods are added scaling and preconditioning. Scaling is allowed for both the solution vector and the system function vector. For scaling to be used, the user should supply values D_u, which are diagonal elements of the scaling matrix such that $D_u u_n$ has all components roughly the same magnitude when u_n is close to a solution, and $D_F F$ has all components roughly the same magnitude when u_n is not too close to a solution. In the text below, we use the following scaled norms:

$$\|z\|_{D_u} = \|D_u z\|_2, \quad \|z\|_{D_F} = \|D_F z\|_2, \quad \text{and} \quad \|z\|_{D,\infty} = \|Dz\|_\infty, \tag{9}$$

where $\|\cdot\|_\infty$ is the max norm. When scaling values are provided for the solution vector, these values are automatically incorporated into the calculation of σ in (8). Additionally, right preconditioning is provided if the preconditioning setup

and solve routines are supplied by the user. In this case, GMRES is applied to the linear systems $(JP^{-1})(P\delta) = -F$.

Two methods of applying a computed step δ_n to the previously computed solution vector are implemented. The first and simplest is the Inexact Newton strategy, which applies step 2(b) as above with λ always set to 1. The other method is a global strategy, which attempts to use the direction implied by δ_n in the most efficient way for furthering convergence of the nonlinear problem. This technique is implemented in the second strategy, called *Linesearch*. This option employs both the α and β conditions of the Goldstein-Armijo linesearch given in Dennis and Schnabel [1996] for step 2(b), where λ is chosen to guarantee a sufficient decrease in F relative to the step length as well as a minimum step length relative to the initial rate of decrease of F. One property of the algorithm is that the full Newton step tends to be taken close to the solution. For more details, the reader is referred to Dennis and Schnabel [1996].

Stopping criteria for the Newton method can be required for either or both of the nonlinear residual and the step length. For the former, the Newton iteration must pass a stopping test

$$\|F(u_n)\|_{D_F,\infty} < \text{FTOL},$$

where FTOL is an input scalar tolerance with a default value of $U^{1/3}$. For the latter, the Newton method will terminate when the maximum scaled step is below a given tolerance

$$\|\delta_n\|_{D_u,\infty} < \text{STEPTOL},$$

where STEPTOL is an input scalar tolerance with a default value of $U^{2/3}$.

Three options for stopping criteria for the linear system solve are implemented, including the two algorithms of Eisenstat and Walker [1996]. The Krylov iteration must pass a stopping test

$$\|J\delta_n + F\|_{D_F} < (\eta_n + U)\|F\|_{D_F},$$

where η_n is one of the following:

—Eisenstat and Walker Choice 1

$$\eta_n = \frac{|\ \|F(u_n)\|_{D_F} - \|F(u_{n-1}) + J(u_{n-1})\delta_n\|_{D_F}\ |}{\|F(u_{n-1})\|_{D_F}};$$

—Eisenstat and Walker Choice 2

$$\eta_n = \gamma \left(\frac{\|F(u_n)\|_{D_F}}{\|F(u_{n-1})\|_{D_F}} \right)^{\alpha},$$

where default values of γ and α are 0.9 and 2, respectively;

—η_n = constant with 0.1 as the default.

The default is Eisenstat and Walker Choice 1. For both options 1 and 2, appropriate safeguards are incorporated to ensure that η does not decrease too fast [Eisenstat and Walker 1996].

As a user option, KINSOL permits the application of inequality constraints, $u^i > 0$ and $u^i < 0$, as well as $u^i \geq 0$ and $u^i \leq 0$, where u^i is the ith component

of u. Any such constraint, or no constraint, may be imposed on each component. KINSOL will reduce step lengths in order to ensure that no constraint is violated. Specifically, if a new Newton iterate will violate a constraint, the maximum (over all i) step length along the Newton direction that will satisfy all constraints is found and δ_n in Step 2(b) is scaled to take a step of that length.

2.3 IDA

The IDA code is a C implementation of a previous code, DASPK, a DAE system solver written in Fortran 77 by Petzold, Brown, and Hindmarsh [Brown et al. 1994; Brenan et al. 1996]. IDA solves the initial-value problem for a DAE system of the general form

$$F(t, y, \dot{y}) = 0, \quad y(t_0) = y_0, \ \dot{y}(t_0) = \dot{y}_0, \tag{10}$$

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

Prior to integrating a DAE initial-value problem, an important requirement is that the pair of vectors y_0 and $\dot{y}_0$ 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 *semiexplicit index-one systems* [Brenan et al. 1996], IDA provides a routine that computes consistent initial conditions from a user's initial guess [Brown et al. 1998]. For this, the user must identify subvectors of y (not necessarily contiguous), denoted y_d and y_a, which are its differential and algebraic parts, respectively, such that F depends on $\dot{y}_d$ but not on any components of $\dot{y}_a$. The assumption that the system is "index one" means that, for a given t and y_d, the system $F(t, y, \dot{y}) = 0$ defines y_a uniquely. In this case, a solver within IDA computes y_a and $\dot{y}_d$ at $t = t_0$, given y_d and an initial guess for y_a.

A second available option with this solver also computes all of $y(t_0)$ given $\dot{y}(t_0)$; this is intended mainly for quasi-steady-state problems, where $\dot{y}(t_0) = 0$ is given. In both cases, IDA solves the system $F(t_0, y_0, \dot{y}_0) = 0$ for the unknown components of y_0 and $\dot{y}_0$, using Newton iteration augmented with a line search global strategy. In doing this, it makes use of the existing machinery that is to be used for solving the linear systems during the integration, in combination with certain tricks involving the step size (which is set artificially for this calculation).

For problems that do not fall into either of these categories, the user is responsible for passing consistent values or risk failure in the numerical integration.

The integration method in IDA is variable-order, variable-coefficient BDF, in fixed-leading-coefficient form [Brenan et al. 1996]. The method order ranges from 1 to 5, with the BDF of order q given by the multistep formula

$$\sum_{i=0}^{q} \alpha_{n,i}\, y_{n-i} = h_n \dot{y}_n, \tag{11}$$

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 (11) to the DAE system (10) results in a nonlinear algebraic system to be solved at each step:

$$G(y_n) \equiv F\left(t_n, y_n, h_n^{-1} \sum_{i=0}^{q} \alpha_{n,i}\, y_{n-i}\right) = 0. \tag{12}$$

Regardless of the method options, the solution of the nonlinear system (12) is accomplished with some form of Newton iteration. This leads to a linear system for each Newton correction, of the form

$$J[y_{n,m+1} - y_{n,m}] = -G(y_{n,m}), \tag{13}$$

where $y_{n,m}$ is the mth approximation to y_n. Here J is some approximation to the system Jacobian

$$J = \frac{\partial G}{\partial y} = \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial \dot{y}}, \tag{14}$$

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

—direct dense solve (serial version only),

—direct banded solve (serial version only), or

—SPGMR = Scaled Preconditioned GMRES, with restarts allowed.

(By *serial version* we mean the IDA solver with the serial NVECTOR module attached.) For the SPGMR case, preconditioning is allowed only on the left,[1] so that GMRES is applied to systems $(P^{-1}J)\Delta y = -P^{-1}G$.

In the process of controlling the various errors, IDA uses the same weighted root-mean-square norm as CVODE, $\|\cdot\|_{\mathrm{WRMS}}$, for all error-like quantities. The weights used are based on the current solution and on input tolerances, as given by (5).

In the cases of a direct linear solver (dense or banded), the nonlinear iteration (13) is a Modified Newton iteration, in that the Jacobian J is fixed (and usually out of date), with a coefficient $\bar{\alpha}$ in place of α in J. When using SPGMR as the linear solver, the iteration is an Inexact Newton iteration, using the current Jacobian (through matrix-free products Jv), in which the linear residual $J\Delta y + G$ is nonzero but controlled. The Jacobian matrix J (direct cases) or preconditioner matrix P (SPGMR case) is updated when

—starting the problem,

—the value $\bar{\alpha}$ at the last update is such that $\alpha/\bar{\alpha} < 3/5$ or $\alpha/\bar{\alpha} > 5/3$, or

—a nonfatal 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

[1]Left preconditioning is required in order to make the norm of the (preconditioned) linear residual in the Newton iteration meaningful. Otherwise this WRMS-norm, $\|J\Delta y + G\|$, is meaningless in general, because it involves division by weights that correspond to y, not G. The appropriate scalings for the components of G, or even their physical units, need not agree with those of y.

storage costs on an update, Jacobian information is always reevaluated from scratch.

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

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

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 mth iteration is then

$$S\|\delta_m\| < 0.33, \tag{15}$$

where $S = R/(R-1)$ whenever $m > 1$ and $R \le 0.9$. The user has the option of changing the constant in the convergence test from its default value of 0.33. The quantity S is set to 20 initially and whenever J or P is updated, and it is reset to 100 on a step with $\alpha \neq \bar{\alpha}$. Note that at $m = 1$, the convergence test (15) 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 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, that is, $\|P^{-1}(Jx + G)\| < 0.05 \cdot 0.33$. The safety factor 0.05 can be changed by the user.

In the direct cases, the Jacobian J defined in (14) can be either supplied by the user or have IDA compute one internally by difference quotients. In the latter case, we use the approximation

$$\begin{aligned} J_{ij} &= [F^i(t, y + \sigma_j e_j, \dot{y} + \alpha\sigma_j e_j) - F^i(t, y, \dot{y})]/\sigma_j, \text{ with} \\ \sigma_j &= \sqrt{U}\max\left\{|y^j|, |h\dot{y}^j|, W_j\right\}\operatorname{sign}(h\dot{y}^j), \end{aligned}$$

where U is the unit roundoff, h is the current step size, and W_j is the error weight for y^j defined by (5). In the SPGMR 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 with CVODE).[2] (As an option, the user can specify a constant factor that is inserted into this expression for σ.)

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

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

$$\|\mathrm{LTE}\| \leq 1.$$

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

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

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

$$\max\{|C|, \bar{C}\}\|\Delta_n\| \leq 1. \tag{16}$$

A user option is available by which the algebraic components of the error vector are omitted from the test (16), if these have been so identified.

In IDA, the local error test is tightly coupled with the logic for selecting the step size and order. First, there is an initial phase that is treated specially; for the first few steps, the step size is doubled and the order raised (from its initial value of 1) on every step, until (a) the local error test (16) fails, (b) the order is reduced (by the rules given below), or (c) the order reaches 5 (the maximum). For step and order selection on the general step, IDA uses a different set of local error estimates, based on the asymptotic behavior of the local error in the case of fixed step sizes. At each of the orders q' equal to q, $q-1$ (if $q > 1$), $q-2$ (if $q > 2$), or $q+1$ (if $q < 5$), there are constants $C_{q'}$ such that the norm of the local truncation error at order q' satisfies

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

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

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

The step/order selection begins with a test for monotonicity that is made even *before* the local error test is performed. Namely, the order is reset to $q' = q-1$ if (a) $q = 2$ and $T(1) \leq T(2)/2$, or (b) $q > 2$ and $\max\{T(q-1), T(q-2)\} \leq T(q)$; otherwise $q' = q$. Next the local error test (16) is performed, and if it fails, the step is redone at order $q \leftarrow q'$ and a new step size h'. The latter is based on the h^{q+1} asymptotic behavior of ELTE_q, and, with safety factors, is given by

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

The value of η is adjusted so that $0.25 \leq \eta \leq 0.9$ before setting $h \leftarrow h' = \eta h$. If the local error test fails a second time, IDA uses $\eta = 0.25$, and on the third and

subsequent failures it uses $q = 1$ and $\eta = 0.25$. After 10 failures, IDA returns with a give-up message.

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

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

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

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

The value of η is adjusted such that (a) if $\eta > 2$, η is reset to 2; (b) if $\eta \leq 1$, η is restricted to $0.5 \leq \eta \leq 0.9$; and (c) if $1 < \eta < 2$, we use $\eta = 1$. Finally h is reset to $h' = \eta h$. Thus we do not increase the step size unless it can be doubled. See Brenan et al. [1996] for details.

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

Normally, IDA takes steps until a user-defined output value $t = t_{\mathrm{out}}$ is overtaken, and then computes $y(t_{\mathrm{out}})$ by interpolation. However, a "one-step" mode option is available, where control returns to the calling program after each step. There are also options to force IDA not to integrate past a given stopping point $t = t_{\mathrm{stop}}$.

3. PRECONDITIONING

All of the SUNDIALS solvers make repeated use of a Krylov method to solve linear systems of the form A (correction vector) $= -$(residual vector), where A is an appropriate Jacobian or Newton matrix. But simple (unpreconditioned) Krylov methods are rarely successful; it is generally necessary to precondition the system in order to obtain acceptable efficiency. A system $Ax = b$ can be preconditioned on the left, as $(P^{-1}A)x = P^{-1}b$; on the right, as $(AP^{-1})Px = b$; or on both sides, as $(P_L^{-1}AP_R^{-1})P_Rx = P_L^{-1}b$. 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. In order to improve the convergence of the Krylov iteration, the preconditioner matrix P, or the product P_LP_R in the last case, should in some sense approximate the

system matrix A. Yet at the same time, in order to be cost-effective, the matrix P, or matrices P_L and P_R, 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, but not always, as we show below.

The CVODE and CVODES solvers allow for preconditioning either side, or on both sides, although we know of no situation where preconditioning on both sides is clearly superior to preconditioning on one side only (with the product $P_L P_R$). In contrast, as noted in the previous section, KINSOL allows only right preconditioning, while IDA and IDAS allow only left preconditioning.

Typical preconditioners used with the solvers in SUNDIALS are based on approximations to the Jacobian matrices of the systems involved. Because the Krylov iteration occurs within a Newton iteration, and often also within a time integration, and 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.

We further exploit this nested iteration setting, and differences in the costs of the various preconditioner operations, by treating in two separate phases each preconditioner matrix P involved:

—a setup phase: evaluate and preprocess P (done infrequently), and

—a solve phase: solve systems $Px = b$ (done frequently).

Accordingly, the user of each solver must supply two separate routines for these operations. The setup of P is generally more expensive than the solve operation, and so it is done as infrequently as possible, with updates to P dictated primarily by convergence failures of the Newton iteration. The system solves $Px = b$ must of course be done at every Krylov iteration (once for each matrix in the case of two-sided preconditioning).

We provide help to SUNDIALS users with respect to preconditioning in two ways. First, for each solver, there is at least one example problem program which illustrates a preconditioner for reaction-diffusion systems, based on the concept of operator splitting. The example does not perform operator splitting (which generally requires giving up error control), but builds the preconditioner from one of two operators (reaction) in the problem. These examples are intended to serve as templates for possible user-defined preconditioners in similar applications. See Brown and Hindmarsh [1989] for an extensive study of preconditioners for reaction-transport systems.

Second, the SUNDIALS package includes some extra preconditioner modules, for optional use with the solvers. For parallel environments, each of the SUNDIALS solvers provides a preconditioner module which generates a band-block-diagonal (BBD) preconditioner. For serial environments, CVODE and CVODES also supply a band preconditioner module. These band and BBD preconditioners are described below. Full details on the usage of these optional

modules are given in the respective user guides—Hindmarsh and Serban [2004a, 2004b], Hindmarsh et al. [2004], and Hindmarsh and Serban [2004c].

In any case, for any given choice of the approximate Jacobian, it may be best to consider choices for the preconditioner linear solver that are more appropriate to the specific problem than those supplied with SUNDIALS.

3.1 Preconditioners for CVODE

Assuming that the CVODE user has chosen one of the stiff system options, recall from (3) that the Newton matrix for the nonlinear iteration has the form $I - \gamma J$, where J is the ODE system Jacobian $J = \partial f/\partial y$. Therefore, a typical choice for the preconditioner matrix P is

$$P = I - \gamma \tilde{J}, \text{ with } \tilde{J} \approx J.$$

As noted above, the approximation may be a crude one.

The setup phase for P is generally performed only once every several time steps, in an attempt to minimize costs. In addition to evaluating P, it may involve preprocessing operations, such as LU decomposition, suitable for later use in the solve phase. Within the setup routine, the user can save and reuse the relevant parts of the approximate Jacobian $\tilde{J}$, as directed by CVODE (in its call to the user routine), so as to further reduce costs when the scalar γ has changed since the last setup call. This option requires the user to manage the storage of the saved data involved. But this tradeoff of storage for potential savings in computation may be beneficial if the cost of evaluating $\tilde{J}$ is significant in comparison with the other operations performed on P.

For serial environments, CVODE supplies a preconditioner called CVBANDPRE, whose use is optional. This preconditioner computes and solves a banded approximation P to the Newton matrix, computed with difference quotient approximations. The user supplies a pair of lower and upper half-bandwidths—`ml`, `mu`—that define the shape of the approximate Jacobian $\tilde{J}$; its full bandwidth is `ml+mu+1`. $\tilde{J}$ is computed using difference quotients, with `ml+mu+1` evaluations of f. The true Jacobian need not be banded, or its true bandwidth may be larger, as long as $\tilde{J}$ approximates J sufficiently well.

Extending this idea to the parallel setting, CVODE also includes a module, called CVBBDPRE, that generates a band-block-diagonal preconditioner. CVBBDPRE is designed for PDE-based problems and uses the idea of domain decomposition, as follows. Suppose that a time-dependent PDE system, with the spatial operators suitably discretized, yields the ODE system $\dot{y} = f(t, y)$. Now consider a decomposition of the (discretized) spatial domain into M non-overlapping subdomains. This decomposition induces a block form $y = (y_1, \ldots, y_M)$ for the vector y, and similarly for f. We will use this distribution for the solution with CVODE on M processors.

The mth block of f, $f_m(t, y)$, depends on both y_m and ghost cell data from other blocks $y_{m'}$, typically in a local manner, according to the discretized spatial operators. However, when we build the preconditioner P, we will ignore that coupling and include only the diagonal blocks $\partial f_m/\partial y_m$. In addition, it may be cost-effective to exclude from P some parts of the function f. Thus, for the

computation of these blocks, we replace f by a function $g \approx f$ (and $g = f$ is certainly allowed). For example, g may be chosen to have a smaller set of ghost cell data than f. In the CVBBDPRE module, the matrix blocks $\partial g_m/\partial y_m$ are approximated by band matrices J_m, again exploiting the local spatial coupling, and on processor m these matrices are computed by a difference quotient scheme. Then the complete preconditioner is given by

$$P = \text{diag}[P_1, \ldots, P_M], \quad P_m = I_m - \gamma J_m.$$

Linear systems $Px = b$ are then solved by banded LU and backsolve operations on each processor. The setup phase consists of the evaluation and banded LU decomposition of P_m, and the solve phase consists of a banded backsolve operation.

In order to minimize costs in the difference quotient scheme, the function g is supplied by the user in the form of two routines. One routine, called once per P evaluation, performs interprocessor communication of data needed to evaluate the g_m. The other routine evaluates g_m on processor m, assuming that the communication routine has already been called. The banded structure of the problem is exploited in two different ways. First, the user supplies a pair of half-bandwidths, `ml` and `mu`, that defines the shape of the matrix J_m. But the user also supplies a second pair of half-bandwidths, `mldq` and `mudq`, for use in the difference quotient scheme, in which J_m is computed by way of `mldq+mudq+2` evaluations of g_m. The values `ml` and `mu` may be smaller than `mldq` and `mudq`, giving a tradeoff between lower matrix costs and slower convergence. Thus, for example, a matrix based on five-point coupling in two dimensions (2D) (`mldq = mudq =` mesh dimension) might be well approximated by a tridiagonal matrix (`ml = mu = 1`). In any case, for the sake of efficiency, both pairs of half-bandwidths may be less than the true values for $\partial g_m/\partial y_m$, and both pairs may depend on m.

3.2 Preconditioners for KINSOL and IDA

The KINSOL package includes a module, called KINBBDPRE, that provides a band-block-diagonal preconditioner for use in parallel environments, analogous to that of the CVODE module CVBBDPRE. Here the problem to be solved is $F(u) = 0$, and the preconditioner is constructed by way of a function $g \approx F$. Namely, it is defined as

$$P = \text{diag}[P_1, \ldots, P_M], \quad P_m \approx \partial g_m/\partial u_m,$$

in terms of the blocks of g and u on processor m. Again, P_m is banded and is computed using difference quotients, with user-supplied half-bandwidths for both the difference quotient scheme and the retained band matrix.

Likewise, the IDA package, in the parallel setting, includes a band-block-diagonal preconditioner module, called IDABBDPRE. For the problem $F(t, y, \dot{y}) = 0$, the preconditioner is defined by way of a function $G \approx F$. Specifically, the preconditioner is

$$P = \text{diag}[P_1, \ldots, P_M], \quad P_m \approx \partial G_m/\partial y_m + \alpha \partial G_m/\partial \dot{y}_m.$$

Each block P_m is banded, computed using difference quotients, with user-supplied half-bandwidths for the difference quotient scheme and the retained matrix.

4. SENSITIVITY ANALYSIS

Many times, models depend on parameters, either through their defining function—$f(t, y)$ for the ODE in (1), $F(t, y, \dot{y})$ for the DAE (10), and $F(u)$ for nonlinear systems (7)—or through initial conditions in the case of ODEs and DAEs. In addition to the solution y or u, we often want to quantify how the solution (or some other output functional that depends on the solution) is influenced by changes in these model parameters.

Depending on the number of model parameters and the number of functional outputs, one of two sensitivity methods is more appropriate. The *forward sensitivity* method is mostly suitable when we need the gradients of many outputs (for example the entire solution vector) with respect to relatively few parameters. In this approach, the model is differentiated with respect to each parameter in turn to yield an additional system of the same size as the original one, the result of which is the solution sensitivity. The gradient of any output function depending on the solution can then be directly obtained from these sensitivities by applying the chain rule of differentiation. The *adjoint sensitivity* method is more practical than the forward approach when the number of parameters is large and when we need the gradients of only few output functionals. In this approach, the solution sensitivities need not be computed explicitly. Instead, for each output functional of interest, we form and solve an additional system, adjoint to the original one, the solution of which can then be used to evaluate the gradient of the output functional with respect to any set of model parameters.

For each of the basic solvers described in Section 2, extensions that are sensitivity-enabled are already available (CVODES), or under development (IDAS), or under consideration depending on the need (KINSOLS). The various algorithmic features of CVODES and IDAS are documented in Cao et al. [2003]. A detailed description of the CVODES software package is presented in Serban and Hindmarsh [2005], while full usage description is given in Hindmarsh and Serban [2004b].

4.1 CVODES

CVODES is an extension of CVODE that, besides solving ODE initial-value problems of the form (1), also provides forward and adjoint sensitivity analysis capabilities. Here, we assume that the system depends on a vector of parameters, $p = [p_1, \ldots, p_{N_p}]$,

$$\dot{y} = f(t, y, p), \quad y(t_0, p) = y_0(p), \tag{17}$$

including the case where the initial value vector y_0 depends on p, and we consider a scalar output functional of the form $g(t, y, p)$. In addition to y as a function of t, we want the total derivative $dg/dp = (\partial g/\partial y)s + \partial g/\partial p$, where $s = dy/dp \in R^{N \times N_p}$ is the so-called *sensitivity matrix*. Each column $s_i = dy/dp_i$

of s satisfies the sensitivity ODE

$$\dot{s}_i = Js_i + \frac{\partial f}{\partial p_i}, \quad s_i(t_0) = \frac{dy_0}{dp_i}, \tag{18}$$

where J is the system Jacobian defined in (3).

4.1.1 *Forward Sensitivity.* CVODES can be used to integrate an extended system $Y = [y, s_1, \ldots, s_{N_s}]$ forward in time, where $[s_1, \ldots, s_{N_s}]$ are a subset of the columns of s. CVODES provides the following three choices for the sequence in which the states and sensitivity variables are advanced in time at each step.

—*Simultaneous Corrector*: the nonlinear system (2) is solved simultaneously for the states and all sensitivity variables [Maly and Petzold 1996], using a coefficient matrix for the Newton update, which is simply the block-diagonal portion of the Newton matrix.

—*Staggered Corrector* 1: the correction stages for the sensitivity variables take place after the states have been corrected and have passed the error test. To prevent frequent Jacobian updates, the linear sensitivity systems are solved with a modified Newton iteration [Feehery et al. 1997].

—*Staggered Corrector* 2: a variant of the previous one, in which the error test for the sensitivity variables is also staggered, one sensitivity system at a time.

The matrices in the *staggered corrector* methods and all of the diagonal blocks in the *simultaneous corrector* method are identical to the matrix M in (3), and therefore the linear systems corresponding to the sensitivity equations are solved using the same preconditioner and/or linear system solver that were specified for the original ODE problem. The sensitivity variables may be suppressed from the step size control algorithm, but they are always included in the nonlinear system convergence test.

The right-hand side of the sensitivity equations may be supplied by a user routine, or approximated by difference quotients at the user's option. In the latter case, CVODES offers both forward and central finite difference approximations. We use increments that take into account several problem-related features, namely, the relative ODE error tolerance RTOL, the machine unit roundoff U, scale factors $\bar{p}$ for the problem parameters p, and the weighted root-mean-square norm of the sensitivity vector s_i. Using central finite differences as an example, the two terms Js_i and $\partial f/\partial p_i$ in the right-hand side of (18) can be evaluated separately:

$$\begin{aligned} Js_i &\approx \frac{f(t, y + \sigma_y s_i, p) - f(t, y - \sigma_y s_i, p)}{2\sigma_y}, \\ \partial f/\partial p_i &\approx \frac{f(t, y, p + \sigma_i e_i) - f(t, y, p - \sigma_i e_i)}{2\sigma_i}, \\ \sigma_i = |\bar{p}_i|\sqrt{\max(\text{RTOL}, U)}, &\quad \sigma_y = \frac{1}{\max(1/\sigma_i, \|s_i\|_{\text{WRMS}}/|\bar{p}_i|)}; \end{aligned} \tag{19}$$

or simultaneously:

$$Js_i + \partial f/\partial p_i \approx \frac{f(t, y + \sigma s_i, p + \sigma e_i) - f(t, y - \sigma s_i, p - \sigma e_i)}{2\sigma}, \qquad (20)$$
$$\sigma = \min(\sigma_i, \sigma_y),$$

or adaptively switching between (19)+(19′) and (20), depending on the relative size of the estimated finite difference increments σ_i and σ_y.

4.1.2 *Adjoint Sensitivity.* CVODES can also be used to carry out adjoint sensitivity analysis, in which the original system for y is integrated forward, an adjoint system is then integrated backward, and finally the desired sensitivities are obtained from the backward solution. To be specific about how the adjoint approach works, we consider the following situation. We assume as before that f and/or y_0 involves the parameter vector p and that there is a functional $g(t, y, p)$ for which we desire the total derivative $(dg/dp)|_{t=t_f}$ at the final time t_f. We first integrate the original problem (17) forward from t_0 to t_f. The next step in the procedure is to integrate from t_f to t_0 the adjoint system

$$\dot{\lambda} = -J^T\lambda, \quad \lambda(t_f) = \left(\frac{\partial g}{\partial y}\right)^T\Bigg|_{t=t_f}. \qquad (21)$$

When this backward integration is complete, then the desired sensitivity array is given by

$$\left.\frac{dg}{dp}\right|_{t=t_f} = \lambda^T(t_0)\frac{dy_0}{dp} + \int_{t_0}^{t_f}\lambda^T\frac{\partial f}{\partial p}dt + \left.\frac{\partial g}{\partial p}\right|_{t=t_f}. \qquad (22)$$

Other situations, with different forms for the desired sensitivity information, are covered by different adjoint systems [Cao et al. 2003].

For the efficient evaluation of integrals such as the one in (22), CVODES allows for special treatment of quadrature equations by excluding them from the nonlinear system solution, while allowing for inclusion or exclusion of the corresponding variables from the step size control algorithm.

During the backward integration, we regenerate $y(t)$ values, as needed, in evaluating the right-hand side of the adjoint system. CVODES settles for a compromise between storage space and execution time by implementing a checkpoint scheme combined with piecewise cubic Hermite interpolation: at the cost of, at most, one additional forward integration, this approach offers the best possible estimate of memory requirements for adjoint sensitivity analysis. Finally, we note that the adjoint sensitivity module in CVODES provides the infrastructure to integrate backward in time any ODE terminal-value problem dependent on the solution of the IVP (17), not just adjoint systems such as (21). In particular, for ODE systems arising from semidiscretization of time-dependent PDEs, this feature allows for integration of either the discretized adjoint PDE system or the adjoint of the discretized PDE.

4.2 IDAS

IDAS, an extension to IDA with sensitivity analysis capabilities, is currently under development and will be soon released as part of SUNDIALS.

Forward sensitivity analysis for systems of DAEs system is similar to that for ODEs. Writing the system as $F(t, y, \dot{y}, p) = 0$ and defining $s = dy/dp$ as before, we obtain DAEs for the individual sensitivity vectors,

$$\frac{\partial F}{\partial y}s_i + \frac{\partial F}{\partial \dot{y}}\dot{s}_i + \frac{\partial F}{\partial p_i} = 0, \quad s_i(t_0) = dy_0/dp_i,\ \dot{s}_i(t_0) = d\dot{y}_0/dp_i. \tag{23}$$

IDAS implements the same three options for correction of the sensitivity variables as CVODES. For the simultaneous corrector approach, the coefficient matrix for the Newton update of the extended system (10)+(23) is again approximated by its diagonal blocks, each of them identical to the matrix J of (14). For the generation of the residuals of the sensitivity equations, IDAS provides several difference quotient approximations equivalent to those described in Section 4.1.

The use of adjoint DAE systems for *adjoint sensitivity* analysis is also similar to the ODE case. As an example, if λ satisfies

$$\frac{d}{dt}\left[\left(\frac{\partial F}{\partial \dot{y}}\right)^T \lambda\right] - \left(\frac{\partial F}{\partial y}\right)^T \lambda = -\left(\frac{\partial g}{\partial y}\right)^T, \tag{24}$$

with appropriate conditions at t_f, then the total derivative of $G(p) = \int_{t_0}^{t_f} g(t, y, p)dt$ is obtained as

$$\frac{dG}{dp} = \int_{t_0}^{t_f}\left(\frac{\partial g}{\partial p} - \lambda^T\frac{\partial F}{\partial p}\right)dt - \left(\lambda^T\frac{\partial F}{\partial \dot{y}}s\right)\Big|_{t_0}^{t_f}.$$

However, unlike the ODE case, homogeneous final conditions for the adjoint variables may not always be enough (such is the case for Hessenberg index-2 DAEs). Moreover, for implicit ODEs and index-1 DAEs the adjoint system may not be stable to integration from the right, even if the original system (10) is stable from the left. To circumvent this problem for such systems, IDAS integrates backward in time the so-called *augmented adjoint DAE system* defined as

$$\begin{aligned} &\dot{\bar{\lambda}} - \left(\frac{\partial F}{\partial y}\right)^T \lambda = -\left(\frac{\partial g}{\partial y}\right)^T, \\ &\bar{\lambda} - \left(\frac{\partial F}{\partial \dot{y}}\right)^T \lambda = 0, \end{aligned} \tag{25}$$

which can be shown to preserve stability [Cao et al. 2003].

IDAS employs a combination of checkpointing with piecewise cubic Hermite interpolation for generation of the solution $y(t)$ needed during the backward integration phase in (24) or (25). As in CVODES, for efficiency, pure quadrature equations are treated separately in that their correction phase does not include a nonlinear system solution. At the user's discretion, the quadrature variables can be included or excluded from the step size control algorithm.

4.3 KINSOLS

In the case of a nonlinear algebraic system, the sensitivity equations are considerably simpler. If the system is written $F(u, p) = 0$ and we define $s = du/dp$,

then for the individual sensitivity vectors s_i,

$$Js_i = -\frac{\partial F}{\partial p_i}, \tag{26}$$

where $J = \partial F/\partial u$.

Forward sensitivity analysis for nonlinear systems thus reduces to solving a number of linear systems equal to the number of model parameters. The Jacobian-vector product and right-hand side of (26) can be provided by the user or evaluated with directional derivatives. In the latter case we approximate Js_i with the formulas presented in Section 2.2 and $\partial F/\partial p_i$ with

$$\frac{\partial F}{\partial p_i} \approx \frac{F(u, p + \sigma_i e_i) - F(u, p)}{\sigma_i},$$

where $\sigma_i = |\bar{p}_i|\sqrt{U}$.

When the dimension N_p of the problem parameters p is large, the *adjoint sensitivity* is again a much more efficient method for computing the total derivative of some functional $g(u, p)$. If λ is the solution of the adjoint system

$$J^T\lambda = \left(\frac{\partial g}{\partial u}\right)^T,$$

then the desired gradient becomes $dg/dp = -\lambda^T(\partial F/\partial p) + (\partial g/\partial p)$.

5. CODE ORGANIZATION

The writing of CVODE from the Fortran 77 solvers VODE and VODPK initiated a complete redesign and reorganization of the existing LLNL solver coding. The features of the design of CVODE include the following:

—memory allocation is heavily used;
—the linear solver modules are separate from the core integrator, so that the latter is independent of the method for solving linear systems;
—each linear solver module contains a generic solver, which is independent of the ODE context, together with an interface to the CVODE core integrator module;
—the vector operations (linear sums, dot products, norms, etc.) on N-vectors are isolated in a separate NVECTOR module.

The process of modularization has continued with the development of CVODE, KINSOL, and IDA. The SUNDIALS distribution now contains a number of common modules in a shared directory. Additionally, compilation of SUNDIALS is now independent of any prior specification of a particular NVECTOR implementation, facilitating the use of binary libraries. The current NVECTOR design also allows the use of multiple implementations within the same code, as may be required to meet user needs.

Figure 1 shows the overall structure of SUNDIALS, with the various separate modules. The evolution of SUNDIALS has been directed toward keeping the entire set of solvers in mind. Thus, CVODE, KINSOL, and IDA share

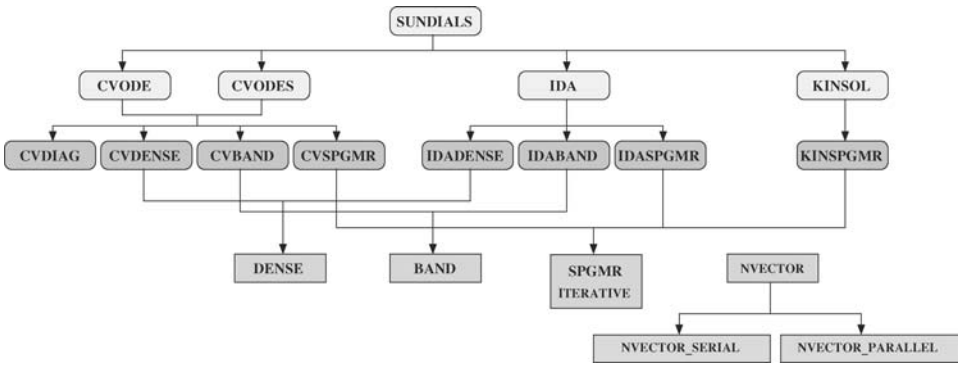

Fig. 1. Overall structure of the SUNDIALS package.

much in their organization and have a number of common modules. The separation of the linear solvers from the core integrators allows for easy addition of linear solvers not currently included in SUNDIALS. At the bottom level is the NVECTOR module, providing key vector operations such as creation, duplication, destruction, summation, and dot products on potentially distributed data vectors. Serial and parallel NVECTOR implementations are included with SUNDIALS, but a user can substitute his/her own implementation as useful. Two small modules defining several data types and elementary mathematical operations are also included.

A number of necessary and optional user-supplied routines for the solvers in SUNDIALS are not shown in Figure 1. The user must provide a routine for the evaluation of f (CVODE) or F (KINSOL and IDA). The user-provided routines may include, depending on the options chosen, routines for Jacobian evaluation (direct cases) or Jacobian-vector products (Krylov case), and routines for the setup and solution of Krylov preconditioners.

5.1 Shared Modules—Linear Solvers

As can be seen in Figure 1, three linear solver packages are currently included with SUNDIALS: a direct dense matrix solver (DENSE); a direct band solver (BAND); and an iterative Krylov solver (SPGMR). These are stand-alone packages in their own right.

The shared linear solvers are accessed from SUNDIALS via solver-specific wrappers. Thus, SPGMR is accessed via CVSPGMR, IDASPGMR, and KINSPGMR, for CVODE (and CVODES), IDA, and KINSOL, respectively. For the DENSE solver, the wrappers are CVDENSE and IDADENSE for CVODE/CVODES and IDA, respectively. Similar wrappers for BAND are CVBAND and IDABAND.

Within each solver, each linear solver module consists primarily of the user-callable function that specifies that linear solver, and four or five wrapper routines. The wrappers conform to a fixed set of specifications, enabling the central solver routines to be independent of the linear system method used. Specifically, for each such module, there are four wrappers—for initialization, Jacobian/precondioner matrix setup, linear system solution, and memory

freeing. The IDA modules have a fifth wrapper, for linear solver performance monitoring. These wrapper specifications are fully described in the user documentation for each solver. By following those, and using any of the existing modules as a model, the user can add a linear solver module to the package, if appropriate.

5.2 Shared Modules—NVECTOR

A generic NVECTOR implementation is used within SUNDIALS to operate on vectors. This generic implementation defines an NVECTOR structure which consists of an implementation-specific content and a set of abstract vector operations. The NVECTOR module also provides a set of wrappers for accessing the actual vector operations of the implementation under which an NVECTOR was created. Because details of vector operations are thus encapsulated within each specific NVECTOR implementation, the solvers in SUNDIALS are now independent of a specific implementation. This allows the solvers to be precompiled as binary libraries and allows more than one NVECTOR implementation to be used within a single program.

A particular NVECTOR implementation, such as the serial and parallel implementations included with SUNDIALS or a user-provided implementation, must provide certain functionalities. At a minimum, each implementation must provide functions to create a new vector, a function to destroy such vectors, and the definitions of vector operations required by the SUNDIALS solvers, including, for example, duplication, summation, element-by-element inversion, and dot product.

If neither the serial nor parallel NVECTOR implementation provided within SUNDIALS is suitable, the user can provide one or more NVECTOR implementations. For example, it might (and has been) more practical to substitute a more complex data structure in a parallel implementation.

For complete details, see the user documentation for any of the solvers in SUNDIALS [Hindmarsh and Serban 2004a, 2004b, 2004c; Hindmarsh et al. 2004].

5.3 User Interface Design

When CVODE was initially developed from VODE and VODPK, its user interface was completely redesigned, and the same design principles were adopted when the other solvers were added to the suite. Further changes in the interface design were made more recently. Unlike the typical Fortran solver, where the interface consists of one callable routine with many arguments, the user interface to each of the SUNDIALS solvers involves many callable routines, each with only a few arguments. The various routines specify the various aspects of the problem to be solved and of the solution method to be used, or retrieve information about the solution. For each solver, there are separate (required) calls that initialize and allocate memory for the problem solver and for the linear system solver it will use. Then there are optional calls to specify various optional inputs (ranging from scalars like maximum method order to the user-supplied Jacobian routine), and optional calls to obtain optional outputs (mostly

performance statistics). Finally, there is a call to restart the solver, when a new problem of the same size is to be solved, but possibly with different initial conditions or a different right-hand side function. The interface design is intended to be fairly simple for the casual user, but also suitably rich and flexible for the more expert user.

6. USAGE

The new design and organization of SUNDIALS as described in Section 5 makes the codes flexible and easy to use. This versatility is due primarily to the control that the user has over the modules that comprise SUNDIALS: the specification of vectors; the linear solver and preconditioner methods; the basic solvers; and sensitivity analysis. Default routines are provided for computing Jacobian-vector approximations, or the right-hand side of the forward sensitivity systems, for example. But for these routines and other basic operations, SUNDIALS allows the user to provide their own variants that may be better suited to their problem-solving needs. Additionally, SUNDIALS provides the user with a fine level of control over various algorithmic parameters, heuristic values, and data structure pointers contained within the codes. Finally, SUNDIALS provides optional routines for extracting the solution, solver statistics, and other useful information from the codes.

A general approach for using SUNDIALS is given below. The outline conveys the basic elements of what is needed to properly specify and solve a problem, the order in which certain tasks must be done, the opportunities for providing user-supplied routines or input values, and so on. Complete details and additional examples are in the documentation that accompanies each solver in SUNDIALS.

(1) SUNDIALS contains header files that define various constants, enumerations, macros, data types, and function prototypes. At a minimum, the user must include header files that declare the SUNDIALS data types for real, integer, and Boolean variables; the NVECTOR implementation to be used; and the solver functions needed to set up and initialize the problem, compute, and extract the solution. Typically, additional header files will be specified to declare the preconditioning and/or linear solver methods to be used.

(2) The user must provide a function for evaluating the equations to be solved. Optionally, a user-defined data structure can be created and passed to this function.

(3) To completely specify the problem, the user must provide whatever initial guesses and/or initial values are needed, specify solution error tolerances, and so on.

(4) The next step is to call a routine for initializing a block of memory that will be used in solving the problem. The memory block is created with certain default values for the solver, such as the use of standard output for writing warning and error messages, or NULL as a default value for the pointer to the user-specified data structure to be passed in evaluating the user's function.

Table I. Optional Inputs for the Basic Solvers in SUNDIALS (The value of unit roundoff for the machine is denoted by U, and *est.* indicates that a quantity is automatically estimated by the code)

Optional Input	CVODE	IDA	KINSOL
Pointer to the user-defined data	NULL	NULL	NULL
Pointer to an error file	NULL	NULL	NULL
Maximum order for BDF method	5	5	—
Maximum order for Adams method	12	—	—
Maximum number of internal steps before t_{out}	500	500	—
Maximum number of warnings for $h < U$	10	—	—
Flag to activate stability limit detection	FALSE	—	—
Initial step size	est.	est.	—
Minimum absolute step size	0.0	—	—
Maximum absolute step size	∞	∞	—
Value of t_{stop}	—	∞	—
Maximum number of Newton iterations	3	4	200
Maximum number of convergence failures	10	10	—
Maximum number of error test failures	7	10	—
Coefficient in the nonlinear convergence test	0.1	0.33	—
Flag to exclude algebraic variables from error test	—	FALSE	—
Differential-algebraic identification vector	—	NULL	—
Vector with additional constraints	—	NULL	NULL
Flag to skip initial linear solver setup call	—	—	FALSE
Maximum number of prec. solves without setup	—	—	10
Flag for selection of η computation	—	—	choice 1
Constant η value	—	—	0.1
Parameters α and γ in η choice 2	—	—	2.0,0.9
Flag to control minimum value for ϵ	—	—	FALSE
Maximum length of Newton step	—	—	est.
Relative error in computing $F(u)$	—	—	U
Stopping tolerance on residual	—	—	$U^{1/3}$
Stopping tolerance on max. scaled step	—	—	$U^{2/3}$

(5) At this stage, the default values in the solver memory block can be changed if so desired. Choices and default values are given in Table I for each of the basic solvers and are discussed further below.

(6) After checking the initialized memory block for errors in the default or optional input values, the user now calls the appropriate routine to perform any required memory allocation.

(7) Typically, preconditioning and/or linear solver methods are needed for solving the linear systems that may arise. These methods can now be attached to the block of memory allocated for the solver. Likewise, if rootfinding is to be done (by CVODE or CVODES) along with the integration, then the user specifications for that task are also attached at this point.

(8) The appropriate routine is called to solve the problem according to the tolerances and other settings that have been specified.

(9) To extract the solution, solver statistics, and other information, optional output extraction routines can be called. A listing of the optional outputs for the basic solvers is given in Table II.

Table II. Optional Outputs for the Basic Solvers in SUNDIALS

Optional Output	CVODE	IDA	KINSOL
Size of workspace allocated by the solver	✓	✓	✓
Cumulative number of internal steps taken	✓	✓	—
Number of calls to the user's function	✓	✓	✓
Number of calls to the linear solver's setup routine	✓	✓	—
Number of local error test failures that have occurred	✓	✓	—
Number of nonlinear solver iterations	✓	✓	✓
Number of nonlinear convergence failures	✓	✓	—
Order used during the last step	✓	✓	—
Order to be attempted on the next step	✓	✓	—
Order reductions due to stability limit detection	✓	—	—
Actual initial step size used	✓	✓	—
Step size used for the last step	✓	✓	—
Step size to be attempted on the next step	✓	✓	—
Current internal time reached by the solver	✓	✓	—
Vector containing the error weights for state variables	✓	✓	—
Vector containing the estimated local errors	✓	—	—
Number of backtrack operations during linesearch	—	✓	✓
Number of times the β condition could not be met	—	—	✓
Scaled norm at a given iteration	—	—	✓
Last step length in the global strategy routine	—	—	✓
Information on roots found	✓	—	—

(10) To end the process, the user must make the appropriate calls to free memory that was allocated in the previous steps. Otherwise, if applicable, a reinitialization routine can be called for solving additional problems.

In order to carry out sensitivity analysis, the above outline needs to be modified at several steps. For forward sensitivities, step (1) requires that the appropriate header file for forward sensitivity analysis be used in place of the header file for the basic solver. At step (2), the user must create an array of real parameters upon which the solution depends and attach a pointer to this array to the user-defined data structure that is passed to the user's function. Also, the user must specify the number of sensitivities to be computed and provide an array that indicates which solution sensitivities are to be computed. Step (6) requires that the user call the memory allocation routine for the forward sensitivity version of the basic solver. As the solution and forward sensitivities are computed, these results and various solver statistics can be extracted as part of steps (8)–(9). Finally, memory space that has been allocated previously must be freed at step (10). For complete details on performing forward or adjoint sensitivity analysis for CVODES, the reader is referred to Serban and Hindmarsh [2005].

If using the parallel NVECTOR module in SUNDIALS, the MPI header file must be specified in step (1) so that in step (3) the MPI communicator can be initialized, the set of active processors can be established, and the global and local vector lengths can be set. In step (10), memory allocated for MPI must be freed.

6.1 Optional Inputs and Outputs

Within SUNDIALS, an attempt is made to set reasonable defaults for the various methods, heuristic parameters, and pointers used in the codes. A key feature

of SUNDIALS is that it provides a collection of optional input and output routines so that default settings can be changed, or various solver statistics and other information can be extracted. These "set" and "get" routines are available for each of the solvers, as noted, as well as for the linear solver and preconditioning methods that support them.

—*Basic solvers*. Table I lists the various optional inputs that the user can set to control the basic solvers within SUNDIALS. Under each solver column we give the default value for the respective input. Inputs marked with a "—" are not applicable to that particular solver. Table II lists the various optional outputs that the user can get to monitor solver performance. Optional outputs available for a solver are marked with a "✓" and those not available are marked by a "—".

—*Sensitivity analysis.* Each sensitivity solver (CVODES and IDAS) offers the complete list of optional "set" and "get" routines as the corresponding basic solver (CVODE and IDA, respectively). In addition, the user has control over various inputs that affect sensitivity calculations. The following are examples of options that can be set by the user with the default given in parentheses: a user-supplied routine to compute sensitivity ODEs or DAE sensitivity residuals (CVODES or IDAS difference quotient approximation); a pointer to user data that will be passed to this user-supplied ODE or DAE sensitivity routine (NULL); a pointer to the sensitivity relative error tolerance scalar (same value as for state variables); and a Boolean flag indicating whether the sensitivity variables are included in the error control mechanism (FALSE). For more options and details, see Serban and Hindmarsh [2005] and Hindmarsh and Serban [2004b].

—*Linear solvers and preconditioners.* For any of the linear solvers, the user can set optional inputs so that a user-supplied routine providing Jacobian-related information is used instead of the default difference quotient routine. Also, a pointer can be set so that user data is passed each time this user-supplied routine is called. In addition, for the SPGMR case, the following can be optionally changed from their default values (provided in parentheses): a classical Gram-Schmidt orthogonalization (modified Gram-Schmidt), the factor by which the tolerance on the nonlinear iteration is multiplied to get a tolerance on the linear iteration (0.05); the preconditioner setup routine (NULL); the preconditioner solver routine (NULL); and a pointer to the user preconditioner data (NULL).

The optional outputs for any of the linear solvers are the amount of integer and real workspace used; the number of calls made to the user-supplied Jacobian evaluation routine; and the number of calls to the user's function within the default difference quotient routine. In addition, for the SPGMR case the user can obtain the number of preconditioner evaluations, the number of calls made to the preconditioner solve routine, the number of linear iterations, and the number of linear convergence failures.

For the band-block-diagonal preconditioner, the optional outputs are the amount of integer workspace used; the amount of real workspace used; and the number of calls to the local function that approximates the user's function.

6.2 Fortran Usage

Some support is available for using Fortran 77 and Fortran 90 applications with SUNDIALS. In particular, a Fortran/C interface package is provided with CVODE and KINSOL. Each package is a collection of C header files and functions that provide interfaces from user Fortran routines to solver C routines, and the reverse. These enable the user to write a main program and all user-supplied routines in Fortran, and then use either CVODE or KINSOL to solve the problem. This mixed-language capability entails some compromises in portability, such as requiring fixed names for the user-supplied routines, but the restrictions are minor. For complete details, see the CVODE and KINSOL user documentation ([Hindmarsh and Serban 2004a; Hindmarsh et al. 2004]).

7. AVAILABILITY

SUNDIALS and each of its individual solvers have been released under BSD open-source licenses. Sources for the entire suite or separately for each of CVODE, CVODES, KINSOL, and IDA are available from the LLNL/CASC web site at

`www.llnl.gov/CASC/sundials`,

or from the DOE ACTS software collection at

`acts.nersc.gov/sundials/main.html`.

Both serial and parallel example applications utilizing the solvers are contained in these sources.

8. CONCLUSIONS

The time integrators and nonlinear solvers within SUNDIALS have been developed to take advantage of the long history of research and development of such codes at LLNL. The codes feature state-of-the-art technology for BDF time integration as well as for inexact Newton-Krylov methods. The design philosophy of providing clear interfaces to the user and allowing the user to supply their own data structures makes the solvers reasonably easy to add into existing simulation codes. As a result, these solvers have been used in numerous applications.

In particular, CVODE has been used to solve three-dimensional radiation diffusion problems on up to 5,800 processors of the ASCI Red machine and verifying the scalability of a fully implicit approach for these problems [Brown and Woodward 2001]. The same code using a preliminary sensitivity version of CVODE was further used to examine behaviors of solution sensitivities to parameters that characterize material opacities for these diffusion problems Lee et al. [2000, 2003]. CVODE is also being used in a three-dimensional tokamak turbulence model within LLNL's Magnetic Fusion Energy Division to solve fusion energy simulation problems with approximately 1.1 million unknowns on 60 processors [Rognlien et al. 2002]. KINSOL is being applied within LLNL to solve a nonlinear Richards' equation model for pressures in variably saturated porous media flows. Fully scalable solution performance of this code has been

obtained on up to 225 processors of ASCI Blue [Jones and Woodward 2001; Woodward 1998]. The same code using a preliminary sensitivity version was used to quantify uncertainty due to variations in relative permeability input parameters within these groundwater problems [Woodward et al. 2002]. IDA has been used in a cloud and aerosol microphysics model at LLNL to study cloud formation processes and to study model parameter sensitivity. CVODE, CVODES, KINSOL, and IDA, with multigrid preconditioners, are being used to solve 3D neutral particle transport problems within LLNL.

Although the SUNDIALS codes have proven to be versatile and robust, further development of the suite is underway. In particular, a sensitivity version of IDA, called IDAS, is currently under development. This code will have forward and adjoint sensitivity capabilities similar to CVODES. Further nonlinear solver capabilities are being considered for extensions to KINSOL, including a trust region globalization method as well as other strategies for choosing finite differencing parameters. In addition, a Picard iteration package and a BiCGStab Krylov solver module are also planned for addition to SUNDIALS.

ACKNOWLEDGMENTS

The authors wish to acknowledge the contributions of Scott Cohen in the development of CVODE, Allan Taylor in the development of KINSOL and IDA, and Homer Walker for implementation of the Eisenstat and Walker forcing term options in KINSOL.

REFERENCES

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

Brown, P. N. 1987. A local convergence theory for combined inexact-Newton/finite difference projection methods. *SIAM J. Numer. Anal. 24*, 2, 407–434.

Brown, P. N., Byrne, G. D., and Hindmarsh, A. C. 1989. VODE, a variable-coefficient ODE solver. *SIAM J. Sci. Stat. Comput. 10*, 1038–1051.

Brown, P. N. and Hindmarsh, A. C. 1989. Reduced storage matrix methods in stiff ODE systems. *J. Appl. Math. Comp. 31*, 49–91.

Brown, P. N., Hindmarsh, A. C., and Petzold, L. R. 1994. Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM J. Sci. Comput. 15*, 1467–1488.

Brown, P. N., Hindmarsh, A. C., and Petzold, L. R. 1998. Consistent initial condition calculation for differential-algebraic systems. *SIAM J. Sci. Comput. 19*, 1495–1512.

Brown, P. N. and Saad, Y. 1990. Hybrid Krylov methods for nonlinear systems of equations. *SIAM J. Sci. Stat. Comput. 11*, 450–481.

Brown, P. N. and Woodward, C. S. 2001. Preconditioning strategies for fully implicit radiation diffusion with material-energy transfer. *SIAM J. Sci. Comput. 23*, 2, 499–516.

Byrne, G. D. 1992. Pragmatic experiments with Krylov methods in the stiff ODE setting. In *Computational Ordinary Differential Equations*, J. Cash and I. Gladwell, Eds. Oxford University Press, Oxford, U.K., 323–356.

Byrne, G. D. and Hindmarsh, A. C. 1975. A polyalgorithm for the numerical solution of ordinary differential equations. *ACM Trans. Math. Softw. 1*, 71–96.

Byrne, G. D. and Hindmarsh, A. C. 1998. User documentation for PVODE, an ODE solver for parallel computers. Tech. rep. UCRL-ID-130884. Lawrence Livermore National Laboratory, Livermore, CA.

Byrne, G. D. and Hindmarsh, A. C. 1999. PVODE, an ODE solver for parallel computers. *Intl. J. High Perf. Comput. Apps. 13*, 4, 254–365.

CAO, Y., LI, S., PETZOLD, L. R., AND SERBAN, R. 2003. Adjoint sensitivity analysis for differential-algebraic equations: The adjoint DAE system and its numerical solution. *SIAM J. Sci. Comput. 24*, 3, 1076–1089.

COHEN, S. D. AND HINDMARSH, A. C. 1994. CVODE user guide. Tech. rep. UCRL-MA-118618. Lawrence Livermore National Laboratory, Livermore, CA.

COHEN, S. D. AND HINDMARSH, A. C. 1996. CVODE, a stiff/nonstiff ODE solver in C. *Comput. Phys. 10*, 2, 138–143.

COLLIER, A. M., HINDMARSH, A. C., SERBAN, R., AND WOODWARD, C. S. 2004. User documentation for KINSOL v2.2.1. LLNL Tech. rep. UCRL-SM-208116. Lawrence Livermore National Laboratory, Livermore, CA.

CURTIS, A. R., POWELL, M. J. D., AND REID, J. K. 1974. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Applic. 13*, 117–119.

DEMBO, R. S., EISENSTAT, S. C., AND STEIHAUG, T. 1982. Inexact Newton methods. *SIAM J. Numer. Anal. 19*, 400–408.

DENNIS, J. E. AND SCHNABEL, R. B. 1996. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, Press, Philadelphia, PA.

EISENSTAT, S. C. AND WALKER, H. F. 1996. Choosing the forcing terms in an inexact Newton method. *SIAM J. Sci. Comput. 17*, 16–32.

FEEHERY, W. F., TOLSMA, J. E., AND BARTON, P. I. 1997. Efficient sensitivity analysis of large-scale differential-algebraic systems. *Appl. Num. Math. 25*, 1, 41–54.

HAIRER, E. AND WANNER, G. 1991. *Solving Ordinary Differential Equations II, Stiff and Differential-Algebraic Problems*. Springer-Verlag, Berlin, Germany.

HIEBERT, K. L. AND SHAMPINE, L. F. 1980. Implicitly defined output points for solutions of ODEs. Tech. rep. SAND80-0180. Sandia National Laboratories, Albuquerque, NM.

HINDMARSH, A. C. 1992. Detecting stability barriers in BDF solvers. In *Computational Ordinary Differential Equations*, J. Cash and I. Gladwell, Eds. Oxford University Press, Oxford, U.K., 87–96.

HINDMARSH, A. C. 1995. Avoiding BDF stability barriers in the MOL solution of advection-dominated problems. *Appl. Num. Math. 17*, 311–318.

HINDMARSH, A. C. 2000. The PVODE and IDA algorithms. Tech. rep. UCRL-ID-141558. Lawrence Livermore National Laboratory, Livermore, CA.

HINDMARSH, A. C. AND SERBAN, R. 2004a. User documentation for CVODE v2.2.1. LLNL Tech. rep. UCRL-SM-208108. Lawrence Livermore National Laboratory, Livermore, CA.

HINDMARSH, A. C. AND SERBAN, R. 2004b. User documentation for CVODES v2.2.1. LLNL Tech. rep. UCRL-SM-208111. Lawrence Livermore National Laboratory, Livermore, CA.

HINDMARSH, A. C. AND SERBAN, R. 2004c. User documentation for IDA v2.2.1. LLNL Tech. rep. UCRL-SM-208112. Lawrence Livermore National Laboratory, Livermore, CA.

JACKSON, K. R. AND SACKS-DAVIS, R. 1980. An alternative implementation of variable step-size multistep formulas for stiff ODEs. *ACM Trans. Math. Softw. 6*, 295–318.

JONES, J. E. AND WOODWARD, C. S. 2001. Newton-Krylov-multigrid solvers for large-scale, highly heterogeneous, variably saturated flow problems. *Adv. Water Resourc. 24*, 763–774.

KELLEY, C. T. 1995. *Iterative Methods for Solving Linear and Nonlinear Equations*. SIAM Press, Philadelphia, PA.

LEE, S. L., HINDMARSH, A. C., AND BROWN, P. N. 2000. User documentation for SensPVODE, a variant of PVODE for sensitivity analysis. Tech. Rep. UCRL-MA-140211. Lawrence Livermore National Laboratory, Livermore, CA.

LEE, S. L., WOODWARD, C. S., AND GRAZIANI, F. 2003. Analyzing radiation diffusion using time-dependent sensitivity-based techniques. *J. Comp. Phys. 192*, 1, 211–230.

MALY, T. AND PETZOLD, L. R. 1996. Numerical methods and software for sensitivity analysis of differential-algebraic systems. *Appl. Num. Math. 20*, 57–79.

RADHAKRISHNAN, K. AND HINDMARSH, A. C. 1993. Description and use of LSODE, the Livermore solver for ordinary differential equations. Tech. rep. UCRL-ID-113855. Lawrence Livermore National Laboratory, Livermore, CA.

ROGNLIEN, T. D., XU, X. Q., AND HINDMARSH, A. C. 2002. Application of parallel implicit methods to edge-plasma numerical simulations. *J. Comp. Phys. 175*, 249–268.

SAAD, Y. AND SCHULTZ, M. H. 1986. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comp. 7*, 856–869.

SERBAN, R. AND HINDMARSH, A. C. 2005. CVODES, the Sensitivity-enabled ODE solver in SUNDIALS. In *Proceedings of the 2005 ASME International Design Engineering Technical Conference* (Long Beach, CA, Sep. 24–28). Also published as 2005 LLNL Tech. rep. UCRL-PROC-21030, Lawrence Livermore National Laboratory, Livermore, CA.

WOODWARD, C. S. 1998. A Newton-Krylov-multigrid solver for variably saturated flow problems. In *Proceedings of the Twelfth International Conference on Computational Methods in Water Resources*, vol. 2. Computational Mechanics Publications, Southampton, U.K., 609–616.

WOODWARD, C. S., GRANT, K. E., AND MAXWELL, R. 2002. Applications of sensitivity analysis to uncertainty quantification for variably saturated flow. In *Computational Methods in Water Resources*, S. M. Hassanizadeh, R. J. Schotting, W. G. Gray, and G. F. Pinder, Eds. Elsevier, Amsterdam, The Netherlands, 73–80.

Received September 2003; revised September 2004; accepted October 2004