Algorithms and Software for Ordinary Differential Equations and Differential/Algebraic Equations

Alan C. Hindmarsh
Lawrence Livermore National Laboratory

Linda R. Petzold
University of Minnesota

Numerical Mathematics Group
Center for Computational Sciences & Engineering

UCRL-JC-116619
April 1994

Algorithms and Software for Ordinary Differential Equations and Differential/Algebraic Equations *

Alan C. Hindmarsh Linda R. Petzold

April 19, 1994

1 Introduction

Gaining insight into a physical process is frequently accomplished by constructing a mathematical model and computing solutions to it. Very often such a model takes the form of a system of differential equations that govern the behavior of the relevant physical variables as a function of time. If these variables are also functions of space, then in the computation the continuous spatial coordinates must also be discretized in some way. Problems of this sort arise in a wide variety of disciplines. Among the scientific areas that generate time-dependent differential-equation problems are chemical kinetics, laser kinetics, mechanical systems, molecular dynamics, power systems, neuronal modeling, electronic networks, computational fluid dynamics, and various reaction-transport processes.

It is rare that a realistic mathematical model is amenable to a purely analytic solution. We must usually generate a computational model from the

*Alan C. Hindmarsh is a mathematician in the Center for Computational Sciences and Engineering at Lawrence Livermore National Laboratory, Livermore, California 94550. Linda R. Petzold is a professor in the Department of Computer Science, University of Minnesota, Minneapolis, MN 55455. Hindmarsh and Petzold have been responsible for developing numerous software packages for the solution of ordinary differential equations and differential-algebraic equations. The work of the first author was performed under the auspices of the U.S Department of Energy by the Lawrence Livermore National Laboratory under contract W-7405-Eng-48.

mathematical one. While avoiding issues of analytic solvability, this introduces a variety of other difficult issues that couple the features of the original model and the computing environment. On recognizing that problems arising in various disciplines share many formal mathematical properties, the field of numerical mathematics seeks to devise powerful and general techniques for the transformation of a mathematical to a computational model, and for the efficient numerical solution of the latter. As a result, many of the differential equation problems that arise in applications are now routinely solved by the use of general-purpose mathematical software packages. The availability of such software has the additional advantage of leaving the scientist free to focus on the content of the model itself instead of the details of its numerical solution.

The effort represented by modern numerical algorithms and software goes far beyond what could be justified within any one discipline or application that benefits from it. A typical ordinary differential equation (ODE) solver available today might well represent several man-years of work just in the development and testing of the computer code, excluding many previous man-years of theoretical investigations into error estimation, numerical stability, and efficiency. The effort leading to a production code is highly cost-effective because it benefits a broad spectrum of users.

In what follows, we identify some of the more important issues in solving problems involving differential equations, show how these ideas lead to various kinds of solution methods, and outline the current state of research work in these areas.

2 Systems of Differential Equations

Mathematical models frequently take the form of a system of ODEs, and for the moment we will suppose that these can be written in the concise and explicit general form

$$\frac{dy}{dt} = f(t, y). \tag{1}$$

Here, t is time and y is a vector of dependent variables of interest (the "state variables"). To save writing, we will often denote dy/dt by y'. The *initial value problem* for (1) is to find the solution $y(t)$ that satisfies a given *initial condition* $y(t_0) = y_0$.

In many instances, the model also involves state variables whose time derivatives do not appear in the equations. Then the set of equations is known as a *differential-algebraic equation* (DAE) system. The most general DAE system is written as

$$F(t, y, y') = 0, \tag{2}$$

where F is some function. An important special case is the "semi-explicit" system

$$\begin{aligned} \frac{dy}{dt} &= f(t, y, z) \\ 0 &= g(t, y, z), \end{aligned} \tag{3}$$

where z is another vector of dependent variables. Here, z is coupled to the ODE for y, but dz/dt does not appear.

The independent variable t need not actually be time, of course. ODE and DAE initial value problems arise in applications in which the independent variable is a spatial coordinate or some other variable, and everything we say applies equally well to such problems. Yet in practice the majority of these problems actually do involve time, and the nomenclature in the literature on numerical ODE and DAE methods reflects that fact, in terms such as "time step" and "time integration."

3 Example: An Ozone Model

In order to give an idea of the kinds of problems for which ODE and DAE solvers can be effectively applied, we give here an example problem derived from a time-dependent system of PDEs. The problem comes from atmospheric modeling, namely the production and transport of ozone in the stratosphere. However, it has been considerably simplified in order to make it presentable in full in a limited space such as this.

The model is a system of two coupled PDEs in time and two space dimensions. The dependent variables represent, respectively, the concentrations of the species O_1 (singlet oxygen) and O_3 (ozone) in moles/cm^3. Molecular oxygen O_2 is of course also present, but is assumed to have a constant concentration of $3.7 \cdot 10^{16}$ here. The kinetic interaction between the three species is governed by the so-called Chapman mechanism, which includes the

destruction of ozone by sunlight:

$$\begin{aligned} O + O_2 &\longrightarrow O_3, & k_1 \\ O + O_3 &\longrightarrow 2O_2, & k_2 \\ O_2 &\longrightarrow 2O, & k_3(t) \\ O_3 &\longrightarrow O + O_2, & k_4(t). \end{aligned}$$

Thus the chemistry is diurnal, having a reaction rate constant that varies with the time of day. In addition, vertical and horizontal diffusion is assumed, with the vertical diffusivity increasing with altitude. Specifically, the PDE system is:

$$\frac{\partial c^i}{\partial t} = K_h \frac{\partial^2 c^i}{\partial x^2} + \frac{\partial}{\partial z}\left[K_v(z)\frac{\partial c^i}{\partial z}\right] + R^i(c^1, c^2, t) \quad (i = 1, 2),$$
$$K_h = 4 \times 10^{-6} \quad \text{and} \quad K_v(z) = 10^{-8} e^{z/5}.$$

The spatial domain is the rectangle $0 \le x \le 20$, $30 \le z \le 50$ km (x = latitude, z = altitude), and the time interval is $0 \le t \le 432,000$ seconds (5 days). The reaction terms are given by:

$$R^1(c^1, c^2, t) = -k_1 c^1 - k_2 c^1 c^2 + k_3(t) \cdot 7.4 \cdot 10^{16} + k_4(t) c^2,$$
$$R^2(c^1, c^2, t) = k_1 c^1 - k_2 c^1 c^2 - k_4(t) c^2,$$
$$k_1 = 6.031, \quad k_2 = 4.66 \cdot 10^{-16},$$
$$k_3(t) = \begin{cases} \exp[-22.62/\sin(\omega t)], & \text{for } \sin(\omega t) > 0, \\ 0, & \text{for } \sin(\omega t) \le 0, \end{cases}$$
$$k_4(t) = \begin{cases} \exp[-7.601/\sin(\omega t)], & \text{for } \sin(\omega t) > 0, \\ 0, & \text{for } \sin(\omega t) \le 0, \end{cases}$$
$$\omega = \pi/43,200 \ .$$

As boundary conditions, we pose homogeneous Neumann boundary conditions (zero gradients) on all boundaries. To complete the problem description, we pose initial profiles for c^i at $t = 0$ which are consistent with the boundary conditions.

The process of generating an ODE system from this PDE problem is referred to as semi-discretization, and also as the Method of Lines. We discretize the spatial region, with (in this case) a uniform mesh of size $M \times M$.

At each mesh point (x_j, z_k) we have approximate values $c^i_{j,k}$ for the two concentrations. Central differencing gives discrete approximations to the spatial derivatives in the PDEs. The result is an ODE system in the vector

$$y = (c^1_{1,1}, c^2_{1,1}, c^1_{2,1}, c^2_{2,1}, \ldots, c^1_{M,M}, c^2_{M,M})^T$$

of length $2M^2$. The ordering is first by species index, then by j, then by k. Initial conditions for the ODE system would simply be the discrete values of the given initial profiles for the $c^i(x, z)$.

The ODE initial value problem obtained in this example has some interesting features. First, the size of the problem can be arbitrarily large, depending on M. Second, the rate constants k_i span a considerable range of values, and as a result the ODE system has the property of "stiffness," which will be discussed in detail shortly. Thirdly, the diurnal variation of the last two rate constants causes a corresponding wide diurnal variation in one of the solution components (oxygen singlet). The diurnal variation of the rate constant k_3 over a five-day period is shown in Figure 1. The combination of these properties makes the problem particularly challenging for an ODE solver.

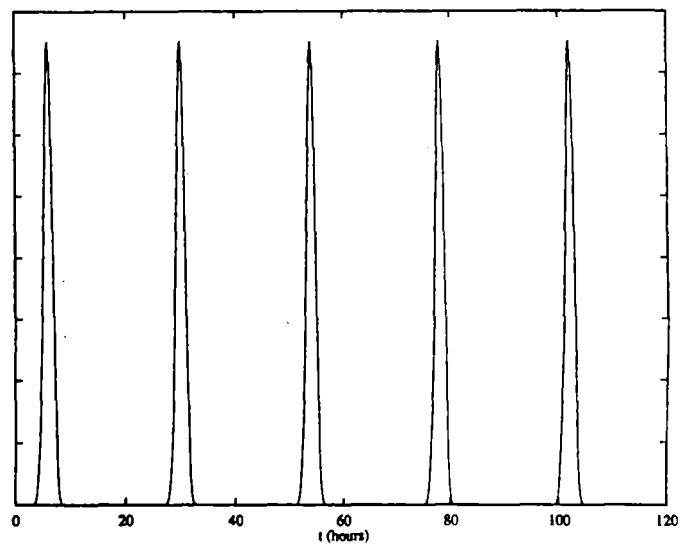

Figure 1: The diurnal kinetic rate constant $k_3(t)$ is shown over a five-day period. It peaks at noon of each day and is zero during the night time. The variation in time of the concentration of O_1 follows the same diurnal pattern very closely.

4 The Euler Method

To illustrate the variety of issues associated with ODE and DAE problems, we examine some typical numerical methods. The oldest and simplest of all methods for solving ODEs was devised by the mathematician Leonhard Euler in the eighteenth century. It consists of computing discrete vectors $y_1, y_2, \ldots$ that approximate $y(t)$ at the times $t_1, t_2, \ldots$, starting from the initial condition $y(t_0) = y_0$. If y_{n-1} has been computed for some $n \geq 1$, then y_n is defined as

$$y_n = y_{n-1} + h_n f(t_{n-1}, y_{n-1}), \tag{4}$$

where $h_n = t_n - t_{n-1}$ is the size of the time step. In other words, the next solution point is computed at time $t_n = t_{n-1} + h_n$ by moving from the point (t_{n-1}, y_{n-1}) on a line at a constant slope of $f(t_{n-1}, y_{n-1})$, the slope of the solution through that point according to (1). The method is completely *explicit*: The new value is defined directly in terms of the known previous values. This leaves unspecified the choice of the step sizes $h_1, h_2, \ldots$ but we defer this question until later. Figure 2 shows this Euler-method solution (connected dots) for a single ODE, along with the true solution (solid curve). In this case, the step sizes h_n are all equal.

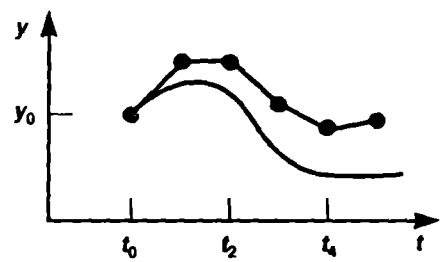

Figure 2: The Euler method is the oldest and simplest technique for solving ODEs. Shown here are the numerical solution computed by the Euler method (connected dots) and the true solution (solid curve). Since the computed solution can quickly drift away from the true solution unless the step sizes are quite small, it is no longer the method of choice.

Although the Euler method is natural and easy to apply, it is rarely the

method of choice, for reasons that will become clear later on. As suggested by Figure 2, the numerical solution can easily drift away from the true solution unless the step sizes are kept quite small. Suppose we use the Euler method to solve (1) from t_0 to a fixed final time T with N steps of equal size $h = (T - t_0)/N$, and that we let $N \to \infty$, so that $h \to 0$. If we suppose also that we know the exact final answer $y(T)$, then we would find that the error in the final computed value y_N behaves as

$$y_N - y(T) = O(h). \tag{5}$$

In fact, this general behavior of the error can be deduced by a careful analysis. We will see later that, even on problems in which the Euler solution appears to be reasonably accurate, much better error behavior can be achieved with other methods (for example, error $= O(h^2)$) at very little additional cost. For reference, the dominant cost in this Euler solution is N evaluations of f (one evaluation per step).

The Euler method is not directly applicable to DAE systems, even in the special semi-explicit case of (3). If we have values y_{n-1} and z_{n-1} approximating y and z at time $t = t_{n-1}$, we can apply the Euler method in (4) to the ODE of (3) to advance y to y_n, but there is no easy way to advance z. We might pose the problem of solving the algebraic equation

$$g(t_n, y_n, z_n) = 0 \tag{6}$$

for z_n (given t_n and y_n), but this may be either difficult, because of the nonlinear way in which g depends on z, or even mathematically impossible, because the dependence of g on z may be singular (unsolvable). In an extreme case (which occurs in equations describing incompressible hydrodynamics), $g = g(t, y)$ does not depend on z at all, and there is no hope of solving (6), yet the DAE system (3), is well-posed (it has a well-defined solution).

5 Stiff Systems

Another important issue in matching solution methods to ODE problems is stiffness. In the simplest terms, the ODE system of (1) is said to be stiff if it has a strongly damped, or "superstable" mode. To get a feeling for this concept, consider the solutions $y(t)$ of an ODE system starting from various

initial conditions. For a typical nonstiff system, if we plot a given component of the vector y versus t, we might get a family of curves such as those shown in Figure 3(a). The curves show a stable tendency to merge as t increases, but not very rapidly. When such a family of curves is plotted for a typical stiff system, the result might be as shown in Figure 3(b). Here, the curves merge rapidly to a set of smoother curves, the deviation from the smooth curve being strongly damped as t increases.

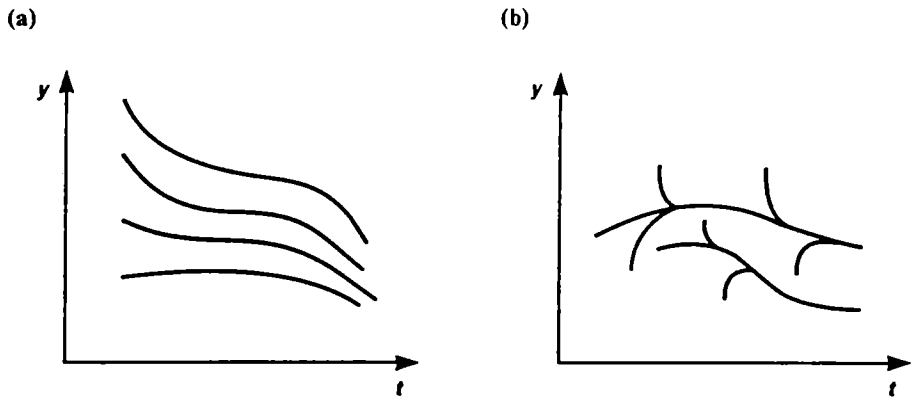

Figure 3: A system of ODEs is said to be "stiff" if its solutions show strongly damped behavior as a function of the initial conditions. The family of curves shown in (a) represents the behavior of solutions to a nonstiff system for various initial conditions. In contrast, solutions to the stiff system shown in (b) tend to merge quickly.

Stiffness in a system of ODEs corresponds to a strongly stable behavior of the physical system being modeled. At any given time, the system is in a sort of equilibrium (though not necessarily a static one). Accordingly, if some state variable is perturbed slightly, the system responds rapidly to restore itself to equilibrium. Typically, the true solution $y(t)$ of the corresponding ODE system shows no such rapid variation, except possibly at the very beginning of the time interval. However, the potential for rapid response is present in the ODEs at all times, and becomes real if one poses an initial value problem by perturbing y at some point out of equilibrium. The system is said to have at least two time scales (or time constants); by a "time scale," we mean the rough value of the spacing of t values needed to resolve a solution curve accurately. There is a long time scale present in the solution of interest, and there is a short time scale given by the damping time (or

time constant) of any of the perturbed solutions. The more different these two time scales are, the stiffer the system is; the ratio of the longest to the shortest time constant in a stiff system is called the "stiffness ratio" of the system.

Stiffness is perhaps best understood by means of a small example. The simple damped oscillator circuit in Figure 4, with a capacitor, a resistor, and an inductor, has an electric current I that obeys the second-order ODE

$$L\frac{d^2I}{dt^2} + R\frac{dI}{dt} + \frac{I}{C} = 0. \tag{7}$$

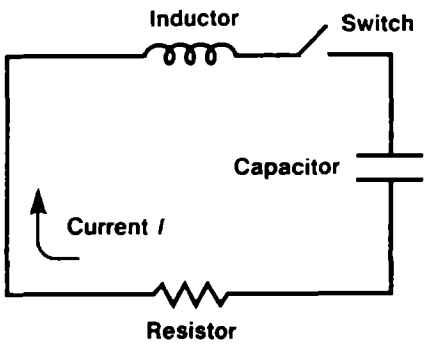

Figure 4: A simple electrical circuit illustrates the behavior of a typical stiff system. In this case, the capacitor damps perturbations to the system caused by a change in current.

If we let y be a vector with two components, $y^1 = I$ and $y^2 = dI/dt$ (we use superscripts to avoid confusion with the notation in (4)), then (7) is equivalent to a system of the same form as (1), namely

$$\begin{aligned} \frac{dy^1}{dt} &= y^2 \\ \frac{dy^2}{dt} &= -(R/L)\,y^2 - y^1/LC \end{aligned} \tag{8}$$

Consider parameter values such that (in suitable dimensionless units) $R/L = 20$ and $LC = 100$, and initial conditions at time $t = 0$ in which $I = 0$

and $dI/dt = 10$ (as if a voltage were applied to the circuit and then switched off). In the notation of (1) and (4), $t_0 = 0$ and $y_0 = \begin{pmatrix} 0 \\ 10 \end{pmatrix}$.

Figure 5 is a plot of the solution (solid line), where the time axis is logarithmic for convenience. Notice that the solution varies on a time scale of less than 0.1 at early times, then becomes smooth and varies on a time scale of around 1000. The system has two different time scales and a stiffness ratio of around 10000. In fact, a precise analytic solution is easily derived. It consists of a linear combination of simple exponential functions $\exp(-t/\tau_1)$ and $\exp(-t/\tau_2)$, where (very nearly) $\tau_1 = 0.05$ and $\tau_2 = 2000$. The short time constant τ_1 is present in the system even when the solution has a much longer time scale, as can be seen by posing an initial value problem with a perturbed initial y at (say) $t = 10$. Such a perturbed solution is shown as the dashed line in Figure 5.

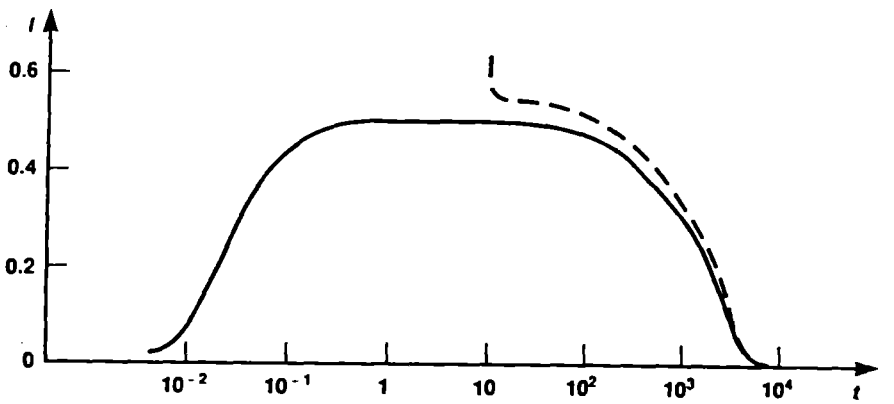

Figure 5: The solid curve is a plot of the solution to the ODE describing the circuit shown in Figure 4. Note that the time axis is logarithmic. The plateau of the curve separates two regions where the solution varies on two different time scales. If the initial value of the y variable is perturbed at $t = 10$ (dashed curve), the shorter time scale predominates for a while, and then the solution displays the same long time scale as the unperturbed solution.

The smallest time scale in a stiff system manifests itself in another way when we try to carry out a numerical solution of the system. Solution by an explicit method like the Euler method either will produce completely inaccurate answers or will require very small time step sizes (comparable with the smallest time constant present in the system) to get accurate answers.

Figure 6 shows a partial solution by the Euler method of the problem of (8), starting with values taken from the earlier one at $t = 10$ and using a constant step size $h = 0.2$ (broken line), along with the true solution (flat curve). After a while, the successive values of $y^1 = I$ oscillate roughly like $(-3)^n$. We say the numerical method is *unstable* when this happens. To get a reasonably accurate and stable Euler method solution of this problem, we must use values of h well below 0.05. Yet this part of the true solution is very well resolved on a time scale of more than 10.

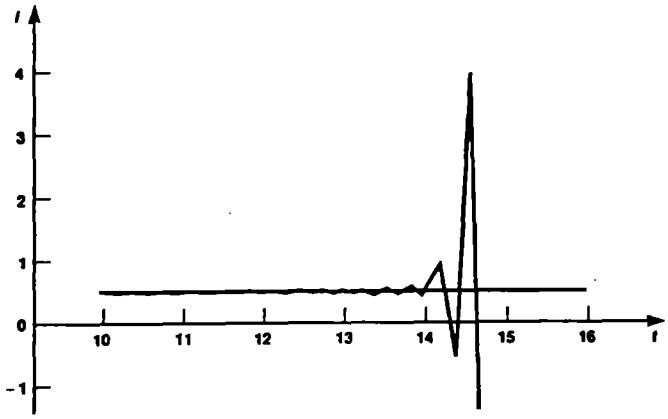

Figure 6: An explicit Euler method solution to the system of ODEs describing the circuit shown in Figure 4. Initial values are the same as those illustrated in Figure 5 at $t = 10$, and the time step is constant. The oscillatory behavior of the numerical solution (broken line) as contrasted with the true solution (flat curve) indicates that the explicit Euler method introduces an instability in this application. An accurate and stable solution by Euler's method would require a step size smaller than the shortest time scale of the problem.

The circuit problem of (8) also provides an example of a DAE system, albeit a very simple one. If we fix R and C but make the inductance L smaller and smaller, the ODE system (8) becomes more and more stiff (the stiffness ratio is roughly R^2C/L). In the limit $L = 0$, (8) (with the second equation first multiplied by L) reduces to the DAE system

$$\begin{aligned} \frac{dy^1}{dt} &= y^2 \\ 0 &= -Ry^2 - y^1/C \end{aligned} \tag{9}$$

Here, no time derivative of y^2 appears, and the system has the general form (3) (with $y = y^1$ and $z = y^2$). The limit process has changed the mathematical properties of the system in a fundamental way: although (7) or (8) allow us to freely specify two initial conditions (I and dI/dt) the system (9) allows only one, since y^1 and y^2 are algebraically related. This example trivially enables us to eliminate y^2, leaving a single first-order ODE, which is the limit of (7) as L approaches zero. But for a complicated DAE problem, this elimination may be either impossible or highly impractical. So if we continue to approach (9) as a system in two dependent variables, we now find that the initial vector $y(0)$ is not arbitrary, as it was in the ODE case. Accordingly, we have to set $y(0)$ in a manner that is consistent with the equations. In this simple example, that means that $y^2 = -y^1/RC$. In a more complicated problem, finding consistent initial conditions may be quite a challenge.

6 The Implicit Euler Method

As we have seen, the explicit Euler method is unstable when applied to a stiff system of ODEs unless the step size is constrained to be smaller than the shortest time scale of the system. This constraint on the step size can be a very severe limitation in some applications, forcing the method to take time steps that are intolerably small before acceptable accuracy is obtained. For some problems, the explicit Euler time steps must be so small (in inverse proportion to the stiffness) that roundoff errors degrade the numerical solution significantly, and the computation cost is prohibitive. It is natural to ask whether there are other methods that can solve stiff systems using time steps that are not limited by stability but only by the need to resolve the solution curve. It is now widely recognized that in general the answer requires the use of implicit methods, and in particular methods that are designed to have good stability properties for stiff systems. The simplest of these methods is the implicit Euler method.

The implicit Euler method for the ODE (1) is given by

$$y_n = y_{n-1} + h_n f(t_n, y_n). \tag{10}$$

In contrast to the explicit Euler formula (4), this method is called *implicit* because y_n is not defined directly in terms of past values of the solution.

Instead, it is defined implicitly as the solution of the nonlinear system of equations (10). We can write this nonlinear system abstractly as

$$F(u) = 0, \tag{11}$$

where $u = y_n$ and $F(u) = u - y_{n-1} - h_n f(t_n, u)$. The nonlinear system of (11) is typically solved by Newton iteration,

$$\left(\frac{\partial F}{\partial u}\right)[u^{(m+1)} - u^{(m)}] = -F(u^{(m)}). \tag{12}$$

Here, if N is the size of the ODE system, u and F are vectors of length N, and the Jacobian matrix $\partial F/\partial u$ is an $N \times N$ matrix of partial derivatives of F evaluated at $u^{(m)}$. Thus, there is a linear system to be solved at each iteration. Newton's method converges in one iteration for linear systems, and the convergence is quite rapid for general nonlinear systems, given a good initial guess. For the initial guess, we can use an explicit formula such as the explicit Euler method or, more commonly, a polynomial that coincides with recent past solution values, evaluated at t_n. In practice, the Jacobian matrix is *not* reevaluated at each iteration, and furthermore is often approximated by numerical difference quotients rather than evaluated exactly. This use of an approximate Jacobian that is fixed throughout the iteration sequence in (12) is called *Modified Newton* iteration.

To gain a better understanding of why the implicit Euler method does not need to restrict the step size to maintain stability for stiff systems, let us consider a very simple example,

$$y' = -\alpha(y - t^2) + 2t \quad , \quad y(0) = 0, \tag{13}$$

on the interval $0 \leq t \leq 1$. Here, α is a positive parameter. When α is very large, the system is stiff. The general solution to (13) is given by

$$y(t) = t^2 + y_0 e^{-\alpha t}.$$

This equation shows clearly that if α is large and the initial value is perturbed slightly away from $y_0 = 0$, the solution tends rapidly back to the curve $y = t^2$. This behavior is characteristic of stiff systems. A sketch of the solution by the implicit Euler method for a slightly perturbed initial value is given in Figure 7(a), where it can be seen that the numerical solution exhibits the

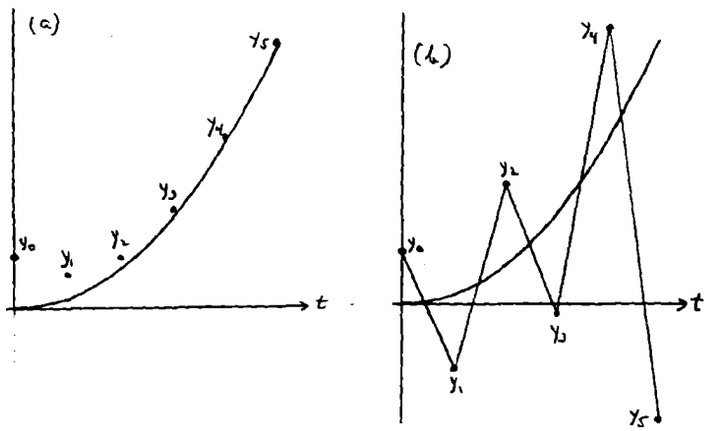

Figure 7: The implicit Euler method overcomes a weakness of the explicit Euler method in that it does not need to restrict the step size to provide stable solutions for stiff systems. The solution of the system of (13) for a slightly perturbed initial value, shown in (a), was generated by the implicit Euler method. It is well-behaved in the sense that the y values merge rapidly with the unperturbed solution curve. In contrast, the explicit Euler method applied to the same system produces the erratic oscillatory behavior shown in (b).

correct behavior. In contrast, the explicit Euler method solution is shown in Figure 7(b), where the instability is evident in the same way as in the circuit example (see Figure 6).

To see why the implicit Euler method gives such a good result for this problem, we can examine the error propagation properties of this method in more detail. When the implicit Euler method is applied to (13), we obtain

$$y_n = y_{n-1} - h\alpha(y_n - t_n^2) + 2ht_n. \tag{14}$$

(Here we are dropping the subscript on h.) If we expand the true solution $y(t)$ in a series about t_{n-1}, we find that

$$y(t_n) = y(t_{n-1}) - h\alpha[y(t_n) - t_n^2] + 2ht_n + O(h^2). \tag{15}$$

Subtracting (15) from (14) and defining the global error $e_n = y_n - y(t_n)$, we obtain

$$e_n = e_{n-1} - h\alpha e_n + O(h^2). \tag{16}$$

Solving for e_n, we see that

$$|e_n| \leq \frac{|e_{n-1}|}{|1 + h\alpha|} + O(h^2). \tag{17}$$

Thus the global error remains small even for large values of α. In contrast, the global error for the explicit Euler method satisfies

$$|e_n| \leq |1 - h\alpha||e_{n-1}| + O(h^2). \tag{18}$$

Here the error will grow exponentially unless $|1 - h\alpha| < 1$. Thus the step size must be constrained to satisfy $h \leq 2/\alpha$.

For general ODE systems $y' = f(t, y)$, the negative of the eigenvalues of the matrix $J = \partial f/\partial y$ play the role of α. For stiff systems, the eigenvalues of $J = \partial f/\partial y$ include at least one with a relatively large negative real part. In the circuit example (8), the eigenvalues of J are approximately -0.0005 and -20.0. The great disparity between these two numbers is what makes the problem stiff. When λ is viewed as an eigenvalue of J, the set of complex numbers $h\lambda$ satisfying $|1 + h\lambda| < 1$ is called the *region of absolute stability* for the explicit Euler method. The corresponding region for the implicit Euler method is given by $1/|1 - h\lambda| < 1$, and is much larger, indicating much greater stability for the implicit method.

The implicit Euler method can also be used to solve the DAE system (2). By identifying $f(t_n, y_n) = (y_n - y_{n-1})/h_n$ in (10) with $y'(t_n)$ in $F(t, y, y') = 0$, we arrive at

$$F\left(t_n, y_n, \frac{y_n - y_{n-1}}{h_n}\right) = 0, \tag{19}$$

which implicitly defines y_n on each time step. It is interesting to note that when the implicit Euler method is applied to the very simple DAE system

$$y(t) - t^2 = 0$$

which is the limit of (13) as $\alpha \to \infty$, the solution is $y_n = t_n^2$. Thus, the implicit Euler method is exact for this problem! More generally, when applied to the semi-explicit DAE system of (3), the implicit Euler method yields the pair of equations

$$\begin{aligned} \frac{y_n - y_{n-1}}{h_n} &= f(t_n, y_n, z_n), \\ 0 &= g(t_n, y_n, z_n), \end{aligned}$$

for the new values y_n and z_n. That is, we replace the ODE by the implicit Euler equation and force the algebraic equation $g = 0$ to hold at the same time. It turns out that the implicit Euler method, as well as some higher-order generalizations of this method, have several properties that make them quite attractive for the solution of DAE systems.

7 Errors and Error Estimates

In the previous section, we derived recurrence relations for the global errors of the implicit and explicit Euler methods applied to a specific stiff ODE. We saw that although the errors remain small for the implicit Euler method, errors for the explicit Euler method can propagate in a disastrous way. It is important in using these methods to have a basic understanding of the various types of errors that are associated with a computation. Modern computer codes attempt to adjust the step size to control the size of some of these errors but not others.

For simplicity, we return to the implicit Euler method applied to the ODE system of (1)

$$y_n = y_{n-1} + h_n f(t_n, y_n). \tag{20}$$

On each step, this method makes an error that results from the approximation of the differential equation by the difference equation. One measure of this error is the amount by which the true solution to the ODE fails to satisfy the difference equation defined by the method. This is known as the *local truncation error* or *local discretization error.* For the implicit Euler method, the local truncation error is given by

$$d_n = y(t_{n-1}) + hf(t_n, y(t_n)) - y(t_n),$$

which, after expanding in a series about t_{n-1}, we can simplify to

$$d_n = -\frac{h^2}{2} y''(\xi_n)$$

for some ξ_n in $t_{n-1} < \xi_n < t_n$.

There is another measure of the error at each time step that lends itself to a more graphical interpretation. The *local error* is the amount by which the numerical solution after one step differs from the value of the true solution to the ODE that passes through the previous numerical solution y_{n-1}. Figure 8 illustrates this error.

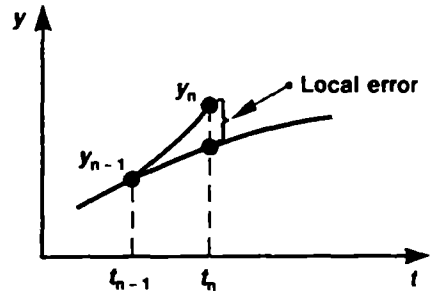

Figure 8: Local error is the difference between the value y_n of the numerical solution to an ODE at a time t_n and the value of the true solution that passes through the numerical solution at the last time step y_{n-1}.

As an example, we shall determine the local error of the implicit Euler method. Let $u(t)$ be the analytic solution to the initial value problem

$$\begin{aligned} u'(t) &= f(t, u(t)) \\ u(t_{n-1}) &= y_{n-1}, \end{aligned}$$

where y_{n-1} is the value of the numerical solution at t_{n-1}. Applying one step of the method, we obtain

$$u_n = y_{n-1} + h_n f(t_n, u_n).$$

The local error is given by

$$\ell_n = u_n - u(t_n).$$

From $d_n = y_{n-1} + hf(t_n, u(t_n)) - u(t_n)$, we find that

$$\ell_n = \left(I - h\frac{\partial f}{\partial y}\right)^{-1} d_n + O(h^3).$$

If the implicit Euler method is applied to nonstiff systems, the local error and local truncation error are nearly the same, whereas for stiff systems, where $h\partial f/\partial y$ is large, these two measures of the error are quite different. However, both are $O(h^2)$ in the limit of small h.

There is yet another measure of the error that is, in a sense, the most relevant for the user of ODE and DAE codes. This is global error, which we touched on briefly above. The global error is the difference between the numerical solution and the true solution to the initial value problem. In the case of either of the two Euler methods, the fact that $d_n = O(h^2)$ can be used to prove that the global errors $y_n - y(t_n)$ are $O(h)$.

One might ask, why bother with the local error and the local truncation errors? The reason is that most ODE codes do not attempt to estimate or control the global error because it is very expensive to do so. Instead, they typically estimate either the local error or the local truncation error, and attempt to control the step size so that a norm of this error is smaller than a user-selected error tolerance. The global error is the result of the propagation of local errors over many time steps. Its eventual size depends not only on the size of the local errors, but on the stability of the method and of the differential equation as well. Local error control in a code can be viewed as a knob that can be turned to try to adjust the step sizes and hence the global error. It is not a guarantee of a small global error.

Finally, we have touched on the notion of an *error estimate*. This is the difference approximation that a code makes to estimate the dominant term of the local truncation error or the local error. For the implicit Euler

method, the local truncation error depends on the local value of y''. This second derivative can be approximated by the difference in y over the past three points: t_{n-2}, t_{n-1}, and t_n. This type of difference approximation of the leading term of the local truncation error is often used in codes based on multistep methods (described below) because the solutions at these points are readily available.

Another type of error estimate is one obtained by computing the solution by two different methods, one of which is locally more accurate than the other. The difference between the locally computed solutions is an approximation to the error of the less accurate method. This type of error estimate is often used in codes based on Runge-Kutta methods, which do not keep past solution values.

Finally, another way to obtain an error estimate is to compute the solution with two different step sizes and to compute the estimate on the basis of its known asymptotic behavior as $h \rightarrow 0$. This type of error estimate is often used in codes based on extrapolation methods.

All of these error estimates are valid in various somewhat idealized situations. It is important to understand, however, that nearly all codes estimate the local error or the local truncation error, and not the global error.

8 Higher-Order Methods

Because of their simplicity, we have been using the explicit and implicit Euler methods to illustrate some basic concepts. Both have first-order accuracy: the global errors are $O(h)$ for a maximum step size of h. In most problems, however, computational efficiency can be considerably increased by using higher-order methods that are generalizations of these simple methods. The importance of higher-order methods is that they are often able to achieve the same level of accuracy as lower-order methods but with many fewer steps and, hence, with much more efficiency. The higher-order methods fall primarily into two classes, multistep methods and one-step methods.

8.1 Multistep Methods

Multistep methods make use of several past values of y and/or f to achieve a higher order of accuracy for the ODE of (1). The general form of a k-step

multistep method is

$$\sum_{j=0}^{k} \alpha_j y_{n-j} = h \sum_{j=0}^{k} \beta_j f_{n-j}, \tag{21}$$

where α_j and β_j are constants that depend on the order, and possibly on previous step sizes, and $\alpha_0 \neq 0$. The quantities y_{n-j} and f_{n-j} represent values of y and y' respectively at the points t_{n-j}. The method is explicit if $\beta_0 = 0$ and implicit otherwise. Here, $h = h_n = t_n - t_{n-1}$.

Several important classes of multistep methods have proven very efficient and robust for solving various types of ODE systems. Adams methods make use of past values of f, and are written

$$y_n = y_{n-1} + h \sum_{j=0}^{k} \beta_j f_{n-j}. \tag{22}$$

Equation (22) gives a method of order $k+1$; i.e., global errors are $O(h^{k+1})$.

The Adams methods are the best known multistep methods for solving general *nonstiff* systems. Several popular codes are based on these methods, which are stable up to order twelve for nonstiff problems. Each step requires the solution of a nonlinear system,

$$y_n = a_n + h\beta_0 f(t_n, y_n)$$

(a_n = past history terms). But rather than use the modified Newton procedure described earlier, this system is nearly always solved by simple *functional iteration.* Here, from a predicted value $y_{n(0)}$, one simply iterates on the function in (22) whose fixed point is sought:

$$y_{n(m+1)} = a_n + h\beta_0 f(t_n, y_{n(m+1)}).$$

This converges reasonably well for nonstiff systems, and has the advantage that no linear systems have to be solved. For this reason, most people refer to the Adams/functional-iteration combination as an explicit method, even though the underlying formula is implicit.

The most effective multistep methods for solving *stiff* systems are the *backward differentiation formulas* (BDFs). BDF methods make use of past values of y to advance the solution, according to the formula

$$y_n = \sum_{i=1}^{k} \alpha_i y_{n-i} + h\beta_0 f_n. \tag{23}$$

The reason for the name is that, on identifying f_n with y'_n, (23) is a formula for approximating $y'(t_n)$ (differentiation) in terms of current and past (backward) values of y_j.

The BDF method based on (23) has order k; i.e., global errors are $O(h^k)$, and it is stable up to order six. The nonlinear system at each time step is almost always solved by some form of Newton iteration, which usually accounts for much, if not most, of the total cost of obtaining the solution. Each Newton iteration involves the solving of an $N \times N$ linear system

$$A\Delta y = \text{residual vector}$$

for a correction to y_n, in which the coefficient matrix is

$$A = I - h\beta_0 J\ , \quad J = \partial f/\partial y, \tag{24}$$

and J is evaluated at some nearby value of t and y. (I denotes the $N \times N$ identity matrix.) Functional iteration is ruled out in the stiff case, because stiffness produces a large value for the Lipschitz constant L of f with respect to y (the maximum of the norm of $\partial f/\partial y$), and convergence of functional iteration requires $hL < 1$; thus step sizes h are severely restricted, just as they are for a nonstiff method such as explicit Euler.

Many widely used codes for solving ODE systems are based on this class of methods. A representative code is the solver LSODE [1]. In addition, BDF methods are very well suited for solving DAE systems. For the general form $F(t, y, y') = 0$, this means requiring y_n to satisfy

$$F\left(t_n, y_n, \frac{y_n - \sum_{i=1}^{k} \alpha_i y_{n-i}}{h\beta_0}\right) = 0. \tag{25}$$

Several popular DAE codes are based on these methods, the most well-known being the solver DASSL [2]. (More will be said about software in a later section.)

The BDF methods were originally proposed and used in *fixed-step* form, where the coefficients α_i and β_0 in (23) depend only on the order k. In the implementations (e.g. LSODE), the step sizes h are actually allowed to change periodically in accordance with a test on the estimated local error, and the steps following a step size change use interpolated values for the y_{n-i} at the new step size. However, many problems demand frequent changes of

step size, and for them the fixed-step BDF methods can lose efficiency or even reliability. In fact, diurnal chemical kinetics problems, such as in the ozone model given above, first demonstrated the need for variable-step forms for BDF methods. Two different variable-coefficient forms of BDF methods have been developed. In both, the method coefficients are recomputed at every step as a function of the actual step sizes $h_n, h_{n-1}, \ldots$ used over the last k steps. But in one version, the so-called *fixed-leading-coefficient* version of BDFs, the value of β_0 does not vary; it depends only on k. This has important consequences for the Newton iteration used, which will be discussed later. The ODE solver VODE [3] and the DAE solver DASSL are representative of codes that use this form of the BDF methods.

Multistep methods are more complex than one-step methods, both to analyze and to implement. Their stability depends on the behavior of the solutions to the difference equation (21). This equation has several fundamental solutions. Coefficients in this method must be chosen so that the extraneous solutions to the difference equation (that is, solutions that do not approximate the solution of the ODE or DAE) do not grow. A robust and efficient implementation of a code based on multistep methods is far from straightforward. Issues that must be dealt with include deriving stable variants of the formulas that are applicable for variable step sizes, estimating errors and changing the step size and order of the method as the problem changes, obtaining suitable starting values, deciding when to terminate the nonlinear iteration, and determining appropriate starting step sizes. These issues are even more complicated for DAE systems, for which much of the ODE methodology is inapplicable.

8.2 One-Step Methods

The second class of higher-order methods is that of one-step methods. Unlike the multistep methods, these methods do not make use of past values of y or f (for the ODE system of (1)) to achieve a higher order. Instead, they depend on evaluations of the differential equation at judiciously chosen locations within the current time step. Such methods are known as Runge-Kutta methods, or extrapolation methods, which are actually a special case of general Runge-Kutta methods. Runge-Kutta methods were discussed in considerable detail by John Butcher in a previous PNA column [4], but we will give a short description here for the sake of completeness. A single

step with a Runge-Kutta method for the ODE $y' = f$ is defined by a set of equations of the form

$$\begin{aligned} y_n &= y_{n-1} + h\sum_{i=1}^{s} b_i k_i, \\ k_i &= f\left(t_{n-1} + c_i h, y_{n-1} + h\sum_{j=1}^{s} a_{ij} k_j\right), \\ & \quad i = 1, 2, \ldots, s. \end{aligned} \tag{26}$$

This defines an $s-$stage Runge-Kutta method. Such a method can be either explicit ($a_{ij} = 0$ for $j \geq i$) or implicit, and some implicit choices are useful for stiff problems. One-step methods offer advantages over multistep methods for some problems. For problems with frequent discontinuities, they are easier to restart at a high order. For stiff systems with highly oscillatory modes, one-step methods are stable with a higher order of accuracy than multistep methods.

A difficulty in implementing one-step methods is finding an efficient solution of the nonlinear system, which is in general larger than for multistep methods. Another is obtaining the solution at points between time steps. This latter task is easily accomplished with multistep methods via a polynomial that passes through past values of y or f. For most problems, it is quite difficult to write a one-step code that is competitive with the best multistep codes. Implicit Runge-Kutta methods are potentially useful for some DAE systems also, but there is in general an additional set of order conditions which the method coefficients must satisfy to achieve a given order [5].

9 Large Stiff Systems

ODE systems that are both stiff and large (in number of ODEs) are especially challenging, even if given in the explicit form of (1). As indicated above, an implicit method then leads to a nonlinear algebraic system that must be solved at every time step. The size and complexity of such systems may make conventional treatments prohibitive in computational cost or memory storage, or both. Considerable research is currently devoted to this class of problems.

For a given time step, we can write the nonlinear system as $F(y) = 0$, where F is related to the function f in $dy/dt = f(t,y)$ by the equation $F(y) = y - a_n - h\beta_0 f(t_n, y)$. By the well-known process of Newton's method, we generate successive approximations to the desired solution vector y by adding corrections that are defined by an approximate linear system. This reduces the problem to a sequence of large linear systems, which we write simply as

$$Ax = b. \tag{27}$$

Here b is a vector of residuals (the negative of $F(y)$ for the current approximation to y), A is a matrix related to the Jacobian J of f, namely $A = I - h\beta_0 J$, and x is the unknown vector of corrections to y.

Instead of relegating this problem to a standard linear-system solution algorithm, an approach that can be much more effective is the use of *iterative methods*. One starts with a guess x_0 (we use $x_0 = 0$), and corrects it successively to get iterates $x_1, x_2, \ldots$. Many iterative methods for linear systems are known, but some are much more appealing than others in the setting of large stiff systems. Such methods are known as Krylov subspace iteration methods. Their crucial property is that at each iteration they require only the value of the matrix-vector product Av for a given vector v. That is, if m iterations have been done, so that one has $x_0, x_1, \ldots, x_m$ (or some equivalent set of vectors), a vector v is generated as a linear combination of these vectors, and the next iterate x_{m+1} is a linear combination of Av and the older vectors. Many methods of this type (such as conjugate gradient iteration, for example) are known to work well when A has certain special properties (such as symmetry), but only a few are good candidates when no such assumptions about A are made. These are the most useful choices, because no special properties can be assumed about the function f from which A is obtained.

Given a suitable Krylov method, it can be exploited to best advantage by finding an efficient way to calculate products Av that does not entail calculating the matrix A itself. To do this, we note that A is just the matrix of partial derivatives of $F(y)$, just as J is that of f. This implies that for a suitably small constant ϵ, $[F(y + \epsilon v) - F(y)]/\epsilon$ is a good approximation to Av. The value of $F(y)$ is already available, and the value of $F(y + \epsilon v)$ is easily expressed in terms of $f(t, y + \epsilon v)$. Thus the Krylov iteration proceeds by making one evaluation of f and some simple vector operations at each

iteration until convergence of the iterates is achieved to within a suitable tolerance. When Newton's method and Krylov iteration are combined with, for example, a BDF method for the ODE system, the result is a *matrix-free* method for stiff systems. In contrast to traditional stiff-system methods, such a method involves no explicit construction or storage of the matrices J or A.

Working from the solver LSODE, which uses BDF methods for stiff ODE systems, we wrote another solver that combines the Krylov methods described above with BDF integration [6, 7]. When tested, it worked well on many of the test problems but failed badly on many others. The reason is that Krylov methods are just not powerful enough, by themselves, to handle with acceptable efficiency the wide variety of matrices A that can occur. However, they can be assisted greatly by a technique known as *preconditioning*.

Suppose we can find a matrix P (the preconditioner matrix) that resembles A to some extent but is much easier to construct and operate with. In particular, suppose that we can solve linear systems $Px = b$ reasonably efficiently. To solve $Ax = b$, we write an equivalent system, say $(AP^{-1})(Px) = b$, with a different matrix $A' = AP^{-1}$ and a different solution vector $x' = Px$, and apply the Krylov method to the problem $A'x' = b$. Each iteration requires the evaluation of a product $A'v = AP^{-1}v$, but that is achieved by solving $Pw = v$ for w and then approximating Aw as before. If the iteration converges to a vector x', then the vector we want is $x = P^{-1}x'$, or the solution of $Px = x'$. Convergence is more likely to occur now, because A' is closer to the identity matrix, depending on how close P is to A. This arrangement is called preconditioning on the right (since P^{-1} multiplies A on the right), but one can just as easily precondition on the left, by writing $(P^{-1}A)x = P^{-1}b$. In fact, one can precondition on both sides, with two preconditioners P_1 and P_2 whose product approximates A.

To incorporate the idea of preconditioning, we wrote another LSODE variant, called LSODPK, containing a selection of preconditioned Krylov methods to solve the linear system problem [8]. The Krylov methods available are Preconditioned Conjugate Gradient iteration (PCG), the Arnoldi method, and the Generalized Minimal Residual method (GMRES) [9]. LSODPK works well on many test problems that could not be handled without preconditioning. Because the choice of preconditioner can best be made by exploiting the structure of the problem, the user of LSODPK must supply the preconditioner. That is, in terms of the ODE system itself, the user must

identify the most important contributions to the Jacobian matrix J (that is, to the stiffness of the ODE system), find a way to represent and operate with these contributions in an economical manner, and then use them to build one or two preconditioner matrices P_1 and P_2. For a complicated problem, the user's job may seem to require as much effort as constructing a complete solution method for the problem from scratch. But it does not, because it focuses on the linear system aspect of the solution only, while the solver takes care of accounting for the errors associated with the choice of preconditioners, for the nonlinear iteration surrounding the linear system, and for the accuracy of the time-stepping procedure.

Although the construction of good preconditioners depends heavily on the nature of the problem, considerable experience has been built up with respect to certain classes of problems. For ODE systems that arise from the spatial discretization of time-dependent systems of PDEs, two natural choices are typically available. First, the terms in the PDEs that reflect how the different PDE components are coupled to each other at each spatial point give rise to one type of preconditioner, which we call the interaction preconditioner. Second, the terms that reflect how each PDE component is transported in space can be used to construct another type of preconditioner, which we call the transport preconditioner. For example, in the ozone model given at the beginning, the chemical kinetics terms R_i lead to an interaction preconditioner and the diffusion terms lead to a transport preconditioner. If both contributions are important, then either they can be regarded as the two preconditioners P_1 and P_2 needed by LSODPK or their product can be used as a single preconditioner on either side.

One particular problem solved by this approach is a system of PDEs on a two-dimensional spatial grid with a discretized frequency variable that represents a laser oscillator model. We developed a pair of preconditioners, first by considering the interaction and transport contributions separately but later with a modification motivated by the Jacobian structure whereby some interaction coefficients were moved to the transport preconditioner. The size of the ODE system varied up to 38,745, and LSODPK generated solutions with complete success.

After seeing how successful the combination of Krylov and BDF methods was with the LSODPK solver, we generated a similar combination with the VODE solver, called VODPK [10]. In this case, the Krylov method chosen is the GMRES method. In using VODPK on a large stiff system, the power

and generality of variable-coefficient BDF methods, Newton iteration, and GMRES iteration is combined with a user-supplied preconditioner (or preconditioner pair) that incorporates problem-specific informtion where it is most needed.

10 Differential-Algebraic Systems

Many physical phenomena are most naturally described by a system of differential/algebraic equations of the form

$$F(t, y, y') = 0. \tag{28}$$

This type of system occurs frequently as an initial value problem in modeling electrical networks, the flow of incompressible fluids, mechanical systems subject to constraints, robotics, distillation processes, power systems, trajectories, control systems, and in many other applications. Differential/algebraic systems are different from ODE systems in that, while they include ODE systems as a special case, they also include problems that are quite different from ODEs. Some of these systems can cause severe difficulties for numerical methods. Consequently, the numerical solution of these systems is a very active area of research. We outline some of the key ideas here; they are described in greater detail in [11].

In a sense, the more singular a DAE system is, the more difficult it is to solve numerically. The *index* of a DAE system is a measure of its degree of singularity. Roughly speaking, ODE systems $y' = f(t, y)$ have *index zero.* Differential equations coupled with algebraic constraints (that is, $y' = f(y, z)$, $0 = g(y, z)$) have *index one* if $g = 0$ can be solved for z given y (that is, if $\partial g/\partial z$ is nonsingular) and otherwise have an index higher than one. The index can also be defined for systems that are not expressed in the semi-explicit form of differential equations coupled with algebraic constraints. Additional difficulties can arise for these systems because the singularity may be moving from one part of the system to another.

A simple example of a higher-index system is given by the equations describing the motion of a pendulum in Cartesian coordinates. Let L denote the length of the bar, λ the force on the bar (suitably normalized), and x and y the coordinates of the infinitesimal ball of mass one located at the free

end of the bar. Then x, y, and λ solve the DAE system

$$\begin{aligned} x'' &= \lambda x, \\ y'' &= \lambda y - g, \\ 0 &= x^2 + y^2 - L^2, \end{aligned} \tag{29}$$

where g is the gravitational constant.

The index of this system is three. While this simple system can be easily rewritten as a standard ODE system by converting to radial coordinates, this is often not practical for the much larger systems that are automatically generated by simulation packages designed to model complicated physical networks.

An even simpler example of a higher-index system, which illustrates some of the ways in which these singular systems are quite different from ODEs, is given by

$$\begin{aligned} y &= g(t), \\ x &= y'. \end{aligned} \tag{30}$$

The index of this system is two. While it looks superficially similar to an ODE system, there are important differences. The solution is less continuous than the input function $g(t)$. There is no family of solutions corresponding to an arbitrary choice of initial values. Rather, the initial values (in fact, all values) are completely determined in terms of the function g and its derivative. Finally, it is clear that there is an implied differentiation to obtain x. Since numerical differentiation is notoriously ill-conditioned (sensitive to small errors), difficulties for numerical ODE methods can be expected when there is a higher-index subsystem present in the system.

Over the past decade, a theoretical framework has been developed for understanding the order, stability and convergence of linear multistep and Runge-Kutta methods applied to general index-one and to index-two and index-three systems that can be written in a semi-explicit triangular form that commonly occurs in applications. Not all ODE methods are appropriate for DAEs; the theory shows which methods are stable and accurate. Often for DAEs there is also a choice of formulations of the equations. Different formulations may have the same exact solution but differ considerably in their properties for numerical solution. Recent work has focused on finding

appropriate formulations for classes of problems in applications which are advantageous for stability and accuracy of the numerical solution [12].

The development of codes for DAEs is not a straightforward task because of difficulties in the computation arising from the singular part of the system and the coupling to the differential part, which do not occur for ODE systems. In particular, starting, error estimation, and solving the nonlinear system all present potential difficulties even for index-one systems, and especially for higher-index systems. We have developed a Fortran package called DASSL [2], which uses fixed-leading-coefficient BDF methods for index-one DAEs. Complete details of the algorithm are available in the book [11]. DASSL has been used successfully for solving a wide range of problems at various universities, laboratories, and in industry, both in the U.S. and in several foreign countries. With some modification as described in [11], DASSL can also be used to solve index-two systems. Codes for DAEs based on Runge-Kutta methods have also been developed; see for example [5]. These methods are particularly effective for problems with frequent discontinuities.

In contrast to the situation for ODEs, initial conditions for DAEs must be *consistent*, in the sense that they must satisfy the constraints of the system and possibly also some of the derivatives of the constraints. For example, for the pendulum problem (29), the constraint and its first and second time derivatives must be satisfied at the initial time, leading to

$$\begin{aligned} 0 &= x^2 + y^2 - L^2, \\ 0 &= xx' + yy', \\ 0 &= \lambda L^2 - gy + (x')^2 + (y')^2. \end{aligned} \tag{31}$$

Currently, the user computes these consistent initial conditions, using his knowledge of the problem and a nonlinear system solver. We are working on a software package to be used in combination with DASSL or its extensions which would make this task more routine for many index-one systems. Methods for finding consistent initial conditions for higher-index systems are described for example in [13].

The success of Krylov iteration methods combined with the ODE solvers LSODE and VODE has inspired the same approach for DAE systems. Accordingly, we developed a variant of DASSL, called DASPK, that combines the preconditioned GMRES Krylov iterative method with the BDF methods of DASSL, as applied to DAE systems [14].

11 Software Packages

Even the best numerical method is unlikely to find wide acceptance until it is embodied in a computer code that is made available for general use. In that spirit, much of our work on methods for ODE and DAE systems has been accompanied by the development of software packages. It is important to understand that this process is not simply a direct translation of a set of formulas into a suitable programming language. Initially, it entails a multitude of decisions on representing and manipulating the relevant data most efficiently and on carrying out all of the numerical processes that together constitute a complete algorithm. The resulting computer code is tested on a wide variety of problems to see that it performs as expected. Then, at some point, it is given to users, along with suitable documentation, so that it can be tried out on realistic problems. All of these phases generate feedback that may result in revision or rewriting of parts of the code. A code often goes through several such feedback-revision cycles during its lifetime.

Various general-purpose packages have been written by the authors of this paper to solve systems of ODEs and or DAEs. These packages are listed in Table 1. Details of the algorithms are available in the various references. Nearly all of the packages listed are available from the Energy Science and Technology Software Center in Oak Ridge. The survey paper [15] on stiff ODE solvers discusses various software, applications, examples, and related issues. The book [11] discusses DAE issues, applications, and software.

A great deal of useful software for solving ODEs and a wide variety of other numerical and non-numerical problems is available freely on the Internet via Netlib [16]. This includes most of the codes listed here. One can obtain an index of Netlib ODE software by

```
mail netlib@ornl.gov
Subject: send index from ode
```

The netlib system will then mail back an index of ODE solvers and descriptions. To obtain one of these solvers (for example, to obtain DDASSL—double precision DASSL), send the following message

```
mail netlib@ornl.gov
send ddassl from ode
```

Solver	Problem	Comments
LSODE	$y' = f(t,y)$	User specifies stiff or nonstiff method; allows dense or banded Jacobian matrix in stiff case.
VODE	$y' = f(t,y)$	Like LSODE, but with variable-coefficient methods internally.
LSODES	$y' = f(t,y)$	LSODE variant for general sparse Jacobian.
LSODA	$y' = f(t,y)$	Automatically, dynamically determines where problem is stiff, and chooses appropriate method; allows dense or banded Jacobian matrix.
LSODAR	$y' = f(t,y)$	Same as LSODA but includes additional root-finding stopping criteria.
LSODI	$M(t,y)y' = g(t,y)$	Solves linearly implicit ODE or DAE system; allows dense or banded coupling.
LSOIBT	$M(t,y)y' = g(t,y)$	Same as LSODI but allows block-tridiagonal coupling.
DASSL	$F(t,y,y') = 0$	Solves index-one DAE systems; allows dense or banded coupling.
DASRT	$F(t,y,y') = 0$	Same as DASSL but with additional root-finding stopping criteria.
LSODPK	$y' = f(t,y)$	LSODE variant; has preconditioned Krylov iterative methods for linear systems.
LSODKR	$y' = f(t,y)$	Like LSODPK, but with root-finding and automatic Newton/functional iteration switching.
VODPK	$y' = f(t,y)$	VODE variant; has preconditioned Krylov iterative methods for linear systems.
DASPK	$F(t,y,y') = 0$	DASSL variant; allows selection of direct methods or preconditioned Krylov iterative methods for linear systems.
CVODE	$y' = f(t,y)$	Rewrite of VODE and VODPK in C.

Table 1: General-purpose multistep packages available from the authors for solving systems of ODEs and/or DAEs

On many X-window systems, an interactive version of netlib called Xnetlib is available.

One software package that is the outcome of a lengthy evolutionary process is a Fortran solver called LSODE [1, 17]. (LSODE was written in 1979, but the comprehensive documentation [1] was only recently completed.) LSODE solves ODE initial value problems that are given in the explicit form of (1). It allows a user to select between an Adams method (for nonstiff systems) and a BDF method (for stiff systems), using the fixed-step-interpolatory form for both of these methods. When solving a stiff system, and therefore when dealing with the Jacobian matrix J in (24), LSODE assumes that the matrix is either full (dense), or banded (has nonzero elements located near its main diagonal). The user can either supply J with coding of his own or let LSODE generate an approximation to J internally. Jacobians generated internally are computed as finite difference quotient approximations. In the dense case, this uses N extra f evaluations, and in the banded case with bandwidth M it uses M extra f evaluations.

More recently, a variable-coefficient solver called VODE [3] was written. VODE looks nearly identical to LSODE as far as its usage is concerned, but the internal algorithm is considerably different. VODE uses the fixed-leading-coefficient form of variable-step BDF methods, and the fully variable-coefficient form of the Adams methods. In addition, it includes a feature not in LSODE that can drastically decrease the number of evaluations of the Jacobian J. VODE normally saves a separate copy of J, and when the modified Newton iteration fails to converge, and the apparent reason is the change in the coefficient $h\beta_0$ in the Newton matrix of (24), that matrix is updated without a re-evaluation of J.

VODE and LSODE are "standard choices" for ODE initial value problems. Some applications, however, give rise to other problem forms that VODE and LSODE cannot handle. For example, a large stiff system may have a Jacobian that is sparse (most elements are nonzero) but not tightly banded. For that case, there is a sparse variant of LSODE called LSODES. It uses parts of the Yale Sparse Matrix Package to solve the linear systems, and it includes an algorithm to generate difference quotient Jacobian approximations with a reduced number of f evaluations.

Another common situation is one where the problem changes with time from stiff to nonstiff and back again. For that case, there is another variant, called LSODA; this code switches automatically between stiff and nonstiff

methods in a dynamic manner. Yet another variant, LSODAR, addresses the case where the ODE solution is to be stopped at a root of some other function (or set of functions) of y, as when a particle trajectory is stopped at the boundary of a geometrical region. Another way of dealing with the change between stiff and nonstiff is to switch dynamically between Newton iteration and functional iteration while using the BDF integration method. This kind of switch has been used in another LSODE variant, LSODKR.

Two other variants of LSODE, called LSODI and LSOIBT, are tailored for the case in which the ODE system is not given in the explicit form of (1) but in an implicit form with a matrix M multiplying the time derivative. This system is written

$$M(t,y)\frac{dy}{dt} = g(t,y). \tag{32}$$

For example, if a PDE problem is treated by the Finite Element Method for the spatial discretization, then M is the mass matrix. Even if M is invertible, so that one could write an equivalent system $dy/dt = M^{-1}g(t,y)$, this is usually not an efficient way to solve the problem. Instead, one can efficiently treat (32) directly by the same methods used in LSODE, slightly reformulated. LSODI does this under the assumption that the matrices involved (M and the various Jacobian matrices) are either full or banded. LSOIBT treats the same problem form, but assumes that the matrices involved are "block-tridiagonal," meaning that the nonzero elements occur in blocks lying on and beside the main diagonal, a common occurrence in semi-discrete forms of PDE problems.

The LSODE solver, together with the variants of it just described, form a "systematized collection" of solvers called ODEPACK [17]. Their outward appearance (the user interface) is standardized by the use of identical names and meanings for features that are common to two or more of the codes. They are also standardized internally by, among other things, the use of shared Fortran subroutines for various subordinate tasks.

Large stiff ODE systems are often beyond the reach of the solvers in ODEPACK, and require iterative methods for the linear systems involved. For this case, there are two variants of LSODE, called LSODPK [8] and LSODKR, and a variant of VODE, called VODPK [10]. All three use Krylov subspace methods with user-supplied preconditioning. In addition, LSODKR includes root-finding (as in LSODAR).

Several solvers have been written for DAE problems. In the linearly

implicit case (32), with M singular, LSODI and LSOIBT have been used with some success. But they were not designed for DAE systems, and are less reliable for them than the DASSL package [11]. DASSL, which also uses a BDF method, treats the linear systems as full or banded, but in various details it addresses the issues of DAE problems directly. A variant of DASSL with a root-finding ability added, called DASRT, is also available.

For large DAE systems, where iterative methods are more suitable that direct methods for the linear systems, we have written a variant of DASSL called DASPK [14], which includes the GMRES Krylov method with user-supplied preconditioning as an option. DASPK actually includes the direct methods of DASSL as well. For use on massively parallel machines, two modified versions of DASPK have been written—one using Fortran 90 (with data-parallelism), and one using message-passing [18]. Codes for computing consistent initial conditions for index-one DAEs and for computing the sensitivity of solutions to DAEs and large-scale DAEs with respect to given parameters, are currently in progress.

In recent years, there has been a trend to away from writing software in Fortran and toward writing in the C language. In response, we have been working on a rewrite in C of the VODE and VODPK solvers (combined), called CVODE. CVODE is composed of a central integrator module that has no knowledge of the nature of the linear system solver (direct or iterative, full or banded, etc.) and a set of linear solve modules that the user selects from prior to starting the integration. An additional motivation for this C rewrite of VODE/VODPK is our plan to extend this package to a parallel version of the solver for distributed-memory MIMD machines.

References

[1] Krishnan Radhakrishnan and Alan C. Hindmarsh, *Discription and Use of LSODE, the Livermore Solver for Ordinary Differential Equations*, NASA Reference Publication 1327, and LLNL Report UCRL-ID-113855, March 1994.

[2] L. R. Petzold, *A Description of DASSL: A Differential/Algebraic System Solver*, in *Scientific Computing*, R. S. Stepleman et al., Eds. (North-Holland, Amsterdam, 1983), pp. 65-68.

[3] P. N. Brown, G. D. Byrne, and A. C. Hindmarsh, *VODE , a Variable-Coefficient ODE Solver*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 1038-1051.

[4] J. C. Butcher, *Runge-Kutta Methods in Modern Computation*, this column, 1994.

[5] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, Springer, 1991.

[6] P. N. Brown and A. C. Hindmarsh, *Matrix-Free Methods for Stiff Systems of ODEs*, SIAM J. Num. Anal., 23 (1986), pp. 610-638.

[7] P. N. Brown, *A Local Convergence Theory for Combined Inexact-Newton/ Finite-Difference Projection Methods*, SIAM J. Num. Anal., 24 (1987), pp. 407-434.

[8] P. N. Brown and A. C. Hindmarsh, *Reduced Storage Matrix Methods in Stiff ODE Systems*, J. Appl. Math & Comp., 31, (1989), 40-91.

[9] Y. Saad and M. H. Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comp., 7 (1986), 856-869.

[10] George D. Byrne, *Pragmatic Experiments with Krylov Methods in the Stiff ODE Setting*, in *Computational Ordinary Differential Equations*, J. R. Cash and I. Gladwell (Eds.), Oxford University Press, Oxford, 1992, pp. 323-356.

[11] K. Brenan, S. Campbell and L. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, Elsevier, New York, 1989.

[12] U. Ascher and L. R. Petzold, *Stability of Computational Methods for Constrained Dynamics Systems*, SIAM J. Sci. Comput., 14 (1993), pp. 95-120.

[13] B. J. Leimkuhler, L. R. Petzold, and C. W. Gear, *Approximation Methods for the Consistent Initialization of Differential-Algebraic Equations*, SIAM J. on Numer. Anal. 28 (1991), pp. 205-226.

[14] P. N. Brown, A. C. Hindmarsh, L. R. Petzold, *Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems*, SIAM J. Sci. Comp., 15 (1994), to appear.

[15] G. D. Byrne and A. C. Hindmarsh, *Stiff ODE Solvers: A Review of Current and Coming Attractions*, J. Comput. Phys., 70 (1987), pp. 1-62.

[16] J. Dongarra and E. Grosse, *Distribution of Mathematical Software via Electronic Mail*, Comm. ACM, 30 (1987), pp. 403-407.

[17] A. C. Hindmarsh, *ODEPACK, A Systematized Collection of ODE Solvers*, in *Scientific Computing*, R. S. Stepleman et al., Eds. (North-Holland, Amsterdam, 1983), p. 55-64.

[18] R. S. Maier, L. R. Petzold and W. Rath, *Solving Large-Scale Differential-Algebraic Equations via DASPK on the CM5*, submitted to Concurrency: Practice and Experience, 1994.