Numerical Methods for Solving Ordinary Differential Equations and Differential/Algebraic Equations

Most systems of differential equations that arise in the sciences cannot be solved exactly. Powerful numerical techniques, coupled with fast computers, have considerably extended the class of systems that can be solved to a good approximation.

For further information contact
Alan C. Hindmarsh (415) 422-4276
or Linda R. Petzold (415) 423-6571.

One way to gain insight into a physical process is to construct a mathematical model and compute solutions to it. 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 the continuous spatial coordinates are also discretized in some way.

Problems of this sort arise in a wide variety of disciplines. Among the areas in the Laboratory’s programmatic applications that generate time-dependent differential-equation problems are chemical kinetics, laser kinetics, electronic networks, and various reaction-transport processes.

Whatever the process being modeled, however, only rarely are the equations involved simple enough to lend themselves to a purely analytic, or exact, solution. Instead, it is nearly always necessary to calculate a numerical, or approximate, solution. Because the techniques involved in numerical solutions usually entail a large number of repetitive steps, they often require the resources of large, fast computers. Rather than going through the time-consuming and expensive process of developing a numerical technique for each system of equations, numerical mathematicians attempt to develop methods that can be applied to broad classes of problems.

In recent years, it has been recognized that problems arising in various disciplines share many formal features. Applied mathematicians and computer scientists have exploited these features to devise powerful and general numerical techniques. The field known as numerical mathematics has evolved in response to the demand of scientists and engineers for efficient and accurate solutions to an enormous variety of problems. As a result, many of the differential-equation problems that arise within Laboratory 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 or engineer free to focus on the content of the model itself instead of the details of its numerical solution.

The effort represented by modern numerical methods and mathematical 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. This is especially true at LLNL, where so many disciplines are represented, and mutual feedback often occurs between the researchers who pose the problems and those responsible for developing solution methods and software. In dozens of instances, LLNL programmatic workers have saved at least 10 man-years of effort by applying existing mathematical software to their problems rather than generating solutions independently.

One example is the development of chemical-kinetics transport models of the stratosphere for studying the climatic impact of ozone depletion in the upper atmosphere. The resulting
models relied heavily on the use of a large ODE system solver that was written at LLNL but based on years of prior theoretical work on stiff-system and iterative linear-system methods.

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.

### Systems of Differential Equations

Mathematical models frequently take the form of a system of ODEs, which we write in general as

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

Here, $t$ is time and $y$ is a vector (or ordered set) of (unknown) dependent variables of interest. Equation 1 relates the time derivative of $y$ (its time rate of change) to $t$ and $y$. The initial-value problem for Equation 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 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

$$F(t,y,y') = 0.$$

where $y'$ denotes $dy/dt$. An important special case is the "semi-explicit" system

$$\frac{dy}{dt} = f(t,y,z),
0 = g(t,y,z).$$

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

### 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 $y_1$, $y_2$, ..., that approximate $y(t)$ at the times $t_1$, $t_2$, ..., 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 obtained from

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

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 $f(t_{n-1}, y_{n-1})$; that is, the slope of the solution through that point according to Equation 1. The method is completely explicit. That is, the new value is defined directly in terms of the known previous values. This leaves unspecified the choice of the step sizes $h_1, h_2, \ldots, h_n$, but we defer this question until later. Figure 1 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.

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 1, 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 Equation 1 from $t_0$ to a fixed final time $T$ with $N$ steps of equal size $h = (T - t_0)/N$, and that we make $N$ larger and larger, so that $h$ gets smaller and smaller. If we suppose also that we know the exact final answer, then we would find that the final error $y_N - y(T)$ behaves roughly like $Kh$ for some constant $K$. When this happens, we say that this error is "of order $h$," and we write

$$\text{error} = O(h).$$

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 reasonably good, much better error behavior can be achieved with other methods (for example, error of order $h^2$) 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 applicable to DAE systems, even in the special case of Equation 3. If we have values

![Figure 1](image_url)

**Figure 1.** 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.
y_{n-1} and z_{n-1} approximating y and z
at time t = t_{n-1}, we can apply the
Euler method in Equation 4 to the
ODE of Equation 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
\[ g(t_n, y_n, z_n) = 0, \] (6)
for z_n (given t_n and y_n), but this may
be either difficult, because of the
nonlinear way in which g might
depend on z, or even mathematically
impossible, because the dependence
of g on z is 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 Equation 6, yet the
DAE system, Equation 3, is well-
posed (it has a well-defined solution).

Stiff Systems

Another important issue in
matching solution methods to ODE
problems is stiffness. In the simplest
terms, the ODE system of Equation 1
is said to be stiff if it has a strongly
damped, or "superstable," mode. To
gain 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 2a. 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 2b. Here, the
curves merge rapidly to a set of
smoother curves, the deviation from
the smooth curve being strongly
damped as t increases.

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 an example. The simple
damped oscillator circuit in Figure 3,
with a capacitor, a resistor, and an
inductor, has an electric current I that
obeys the second-order ODE
\[ L \frac{d^2 I}{dt^2} + R \frac{dI}{dt} + \frac{I}{C} = 0. \] (7)

Figure 3. 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.

Figure 2. A system of ODEs is said to be "stiff" if its solutions are relatively insensitive to initial conditions. The family of curves shown in (a) represents the behavior of solutions to a nonstiff system for various initial conditions. The solutions remain separate over time. In contrast, solutions to the stiff system shown in (b) tend to merge quickly. Strongly damped physical systems are described by stiff systems of ODEs.
If we let $y$ be a vector with two components, $y^1 = I$ and $y^2 = dl/dt$ (we use superscripts to avoid confusion with the notation in Equation 4), then Equation 7 is equivalent to a system of the same form as Equation 1, namely

\[
dy^1/dt = y^2,
\]

\[
dy^2/dt = -(R/L)y^2 - y^1/LC.
\]

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

Figure 4 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 10,000.

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 $\tau_1$ 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 4.

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 5 shows a partial Euler-method solution of the problem of Equation 8, starting with values taken from the earlier one at $t = 10$ and using a constant step size $h = 0.2$ (colored curve), along with the true solution (black 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 over 10.

The circuit problem of Equation 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 of Equation 8 becomes more and more stiff (the stiffness ratio is roughly $R^2C/L$). In the limit $L = 0$, the system of Equation 8 (with the second equation first multiplied by $L$) reduces to the DAE system

\[
dy^1/dt = y^2,
\]

\[
0 = -Ry^2 - y^1/C.
\]

(9)

Here, no time derivative of $y^2$ appears, and the system has the general form of Equation 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 Equations 7 or 8 allow us to freely specify two initial conditions ($I$ and $dl/dt$), the system (Equation 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 Equation 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 Equation 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 \( y^2 = -y/v/RC \). In a more complicated problem, finding consistent initial conditions may be quite a challenge.

**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. In practice, the time steps often need to be so small that round-off errors and computation time become critical. 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 the cure for this problem is to use implicit 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 of Equation 1 is given by

\[
y_n = y_{n-1} + h_n f(t_n, y_n).
\]

It is instructive to compare this with the explicit Euler formula (Equation 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 (Equation 10). We can write this nonlinear system abstractly as

\[
F(u) = 0,
\]

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

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

Here, \( u \) 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 \( t_j \). In practice, the Jacobian matrix is not reevaluated at each iteration, and furthermore is often approximated by numerical

Figure 5. An explicit Euler-method solution to the system of ODEs describing the circuit shown in Figure 3. Initial values are the same as those illustrated in Figure 4 at \( t = 10 \), and the time step is constant. The oscillatory behavior of the numerical solution (colored curve) as contrasted with the true solution (black curve) indicates that the explicit Euler's 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. For some problems, time steps must be so small that round-off errors and computation time dominate the numerical solution.
difference quotients rather than evaluated exactly.

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,

\[ y' = -\alpha (y - t^2) + 2t, \quad (13) \]

\[ y(0) = 0, \]

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

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

Note that if the initial value is perturbed slightly away from \( y_0 = 0 \), for large \( \alpha \), the solution tends rapidly 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 6a, where it can be seen that the numerical solution exhibits the correct behavior. In contrast, the explicit Euler-method solution is shown in Figure 6b, where the instability is evident in the same way as in the circuit example (see Figure 5).

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 Equation 13, we obtain

\[ y_n = y_{n-1} - h\alpha (y_n - t_n^2) + 2t_n. \quad (14) \]

Here, we are dropping the subscript on \( h \). If we expand Equation 14 in a series about \( t_{n-1} \), the true solution \( y(t_n) \) satisfies

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

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

\[ e_n = e_{n-1} - h\alpha e_n + O(h^2). \quad (16) \]

Solving for \( e_n \), we see that

\[ |e_n| \leq |e_{n-1}| + O(h^2). \quad (17) \]

Thus the global error remains small for large values of \( \alpha \). In contrast, the global error for the explicit Euler method satisfies

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

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 systems, \( y' = \mathbf{f}(t,y) \), the negative of the eigenvalues of the matrix \( \mathbf{J} = \partial \mathbf{f}/\partial \mathbf{y} \) play the part of \( \alpha \). (The eigenvalues of an \( N \times N \) matrix \( \mathbf{J} \) are the complex numbers \( \lambda \) such that \( \mathbf{J}x = \lambda x \) for some nonzero vector \( x \).) For stiff systems, the eigenvalues of \( \mathbf{J} = \partial \mathbf{f}/\partial \mathbf{y} \) include at least one with a relatively large negative real part. In the circuit example (Equation 8), the eigenvalues of \( \mathbf{J} \) are approximately \(-0.0005\) and \(-20.0\). The great disparity between these two numbers is what makes the problem stiff.

When \( \lambda \) is viewed as an eigenvalue of \( \mathbf{J} \), 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 be used to solve the DAE system of Equation 2. By identifying \( \mathbf{f}(t_n, \mathbf{y}_n) = (y_n - y_{n-1})/h_n \) with \( y'(t_n) \) in Equation 10 we arrive at the equation

\[ \mathbf{F}(t_n, \mathbf{y}_n, \mathbf{y}_n - \mathbf{y}_{n-1}/h_n) = 0, \quad (19) \]

Figure 6. The implicit Euler method overcomes a weakness of the explicit Euler method inasmuch as it does not need to restrict the step size to provide stable solutions for stiff systems. The solution of the system of Equation 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).
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 Equation 13 as $a \to \infty$, the solution is $y_n = t_n^2$. Thus, the implicit Euler method is exact! Applied to the semi-explicit DAE system of Equation 3, the implicit Euler method yields the pair of equations

$$\frac{y_n - y_{n-1}}{h_n} = f(t_n, y_n, z_n),$$

$$0 = g(t_n, y_n, z_n),$$

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.

**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 Equation 1:

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

(20)

This method makes, on each step, 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

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

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 $t_{n-1} < \xi_n < t_n$.

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 7 illustrates this error.

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

$$u'(t) = f(t, u(t)),$$

$$u(t_{n-1}) = y_{n-1},$$

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

$$l_n = u_n - u(t_n).$$

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

$$l_n = \left(1 - h \frac{df}{dy}\right)^{-1} d_n + O(h^3).$$

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 $h \frac{df}{dy}$ is large, these two measures of the error are quite different.

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. 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 to compute the solution by two different methods, one of which is locally more accurate than the other. The difference between the local solutions computed by the two methods 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 \( h \to 0 \).

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

**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. 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.

**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 Equation 1. The general form of a \( k \)-step multistep method is

\[
\sum_{j=0}^{k} \alpha_j y_{n-j} = h \sum_{j=0}^{k} \beta_j f_{n-j},
\]

where \( \alpha_j \) and \( \beta_j \) are constants that depend on the order and on previous step sizes, and \( \alpha_0 \neq 0 \). The method is explicit if \( \beta_0 = 0 \) and implicit otherwise. Here, \( h = h_n \).

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

\[
y_n = y_{n-1} + h \sum_{j=0}^{k} \beta_j f_{n-j}.
\]

Equation 22 gives a method of order \( k+1 \), i.e., global error equals \( O(h^{k+1}) \).

The Adams methods are the best-known methods for solving general nonstiff systems. Several popular codes are based on these methods, which are stable up to order twelve for nonstiff problems. The nonlinear system in the implicit Adams method is usually solved by simple functional iteration, which converges for nonstiff systems. In functional iteration, given an approximate value \( y_{n(n_0)} \), we insert this as the \( y \) argument of \( f_n = f(t_n, y_n) \) of Equation 22 and let \( y_{n(n+1)} \) be the resulting right-hand side. There are no linear systems to be solved.

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

\[
y_n = \sum_{i=1}^{k} \alpha_i y_{n-i} + h \beta_0 y'_n.
\]

The method based on Equation 23 has order \( k \), i.e., global error equals \( O(h^k) \), and 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 most of the cost of obtaining the solution.
Several popular codes for solving ODE systems are based on this class of methods. The BDF methods are also very well suited for solving DAE systems. For the general form $F(t,y,y') = 0$, this means requiring $y_n$ to satisfy

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

Several popular DAE codes are based on these methods.

Multistep methods are the most complex, both to analyze and to implement. Their stability depends on the behavior of the solutions to the difference equation (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) 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, to which much of the usual methodology is inapplicable.

**One-Step Methods**

The second class of higher-order methods comprises 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 Equation 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. A single step with a Runge-Kutta method for $y' = f$ is defined by a set of equations of the form

$$ y_n = y_{n-1} + h \sum_{i=1}^{s} b_i k_i, $$

$$ k_i = f \left( t_{n-1} + c_i h, y_{n-1} + h \sum_{j=1}^{s} a_{ij} y_j \right), $$

$$ i = 1, 2, \ldots, s. \quad (24)$$

Such methods can be explicit 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 to obtain an efficient solution of the nonlinear system, which is in general larger than for multistep methods, and to obtain 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 they raise new questions of accuracy, described below.

**Computer Codes**

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 is 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 usually goes through several such feedback-revision cycles during its lifetime.

Various general-purpose packages have been written to solve systems of ODEs and or DAEs. Some of these packages are listed in Table 1. In most cases, the solvers were written at LLNL but based in part on prior work done elsewhere; see References 1-6 for details. The packages listed are all available from the National Energy Software Center at Argonne National Laboratory.

One software package that is the outcome of a lengthy evolutionary process is a Fortran program called LSODE. LSODE solves ODE initial-value problems that are given in the explicit form of Equation 1. It allows
a user to select between an Adams method (for nonstiff systems) and a BDF method (for stiff systems). When solving a stiff system, and therefore when dealing with the Jacobian matrix \( J \), 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 can let LSODE generate an approximation to \( J \) internally (at somewhat greater expense). Thus, in one sense, LSODE really includes at least eight different methods rather than one.

LSODE is a “standard choice” for ODE initial-value problems. Some applications, however, give rise to other problem forms that 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, we wrote a sparse variant of LSODE called LSODES. Another common situation is one where the problem changes with time from stiff to nonstiff and back again. For that case, we wrote 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. Finally, two other variants, LSODI and LSOIBT, are tailored for the case in which the ODE system is not given in the explicit form of Equation 1 but in an implicit form with a matrix \( A \) multiplying the time derivative. This system is written

\[
A(t,y) \frac{dy}{dt} = g(t,y). \tag{25}
\]

Even if \( A \) is invertible, so that one could write an equivalent system \( \frac{dy}{dt} = A^{-1} g(t,y) \), this is usually not an efficient way to solve the problem. Instead, one can efficiently treat Equation 25 directly by the same methods used in LSODE, slightly reformulated. LSODI does this under the assumption that the matrices involved (\( A \) 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.

The six ODE packages just described form a “systematized collection” of solvers called ODEPACK. 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.

When the matrix \( A \) of Equation 25 is singular (not invertible), the system is a DAE system. The ODEPACK

---

**Table 1.** Representative general-purpose packages for solving systems of ODEs and/or DAEs.

<table>
<thead>
<tr>
<th>Solver</th>
<th>Problem</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSODE</td>
<td>( y' = y(t,y) )</td>
<td>User specifies whether problem is stiff or nonstiff; allows dense or banded Jacobian matrix in stiff case.</td>
</tr>
<tr>
<td>LSODES</td>
<td>( y' = f(t,y) )</td>
<td>Like LSODE, but allows general sparse Jacobian.</td>
</tr>
<tr>
<td>LSODA</td>
<td>( y' = f(t,y) )</td>
<td>Automatically and dynamically determines whether problem is stiff and chooses appropriate method; allows dense or banded Jacobian matrix.</td>
</tr>
<tr>
<td>LSODAR</td>
<td>( y' = f(t,y) )</td>
<td>Same as LSODA but includes additional root-finding stopping criteria.</td>
</tr>
<tr>
<td>LSODI</td>
<td>( A(t,y)y' = g(t,y) )</td>
<td>Solves linearly implicit ODE system or DAE systems; allows dense or banded coupling.</td>
</tr>
<tr>
<td>LSOIBT</td>
<td>( A(t,y)y' = g(t,y) )</td>
<td>Same as LSODI but allows block-tridiagonal coupling.</td>
</tr>
<tr>
<td>DASSL</td>
<td>( F(t,y)y' = 0 )</td>
<td>Solves general DAE systems; allows dense or banded coupling.</td>
</tr>
<tr>
<td>LSODPK</td>
<td>( y' = f(t,y) )</td>
<td>Experimental code; uses preconditioned Krylov iterative methods for algebraic systems.</td>
</tr>
</tbody>
</table>
solving for Equation 25 have been used on such problems with some success. But they were not designed for DAE systems, and are less reliable for them than another package, called DASSL. 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. More is said of this below.

The ODEPACK solvers, DASSL, and other solvers developed for various kinds of ODE and DAE problems are widely used, both within and outside the Laboratory. Uses vary from one-shot simulations, where one wants a numerical solution to a single model as quickly as possible, to incorporation in an application code that exercises the ODE/DAE solver repeatedly in a production mode over a broad class of problems. The reliability, accuracy, and efficiency of the solver are important in both of these settings. A recent survey of stiff ODE solvers discusses various software, applications, examples, and related issues.3

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 Equation 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 problem.

For a given time step, we can write the nonlinear system as \( F(y) = 0 \), where \( F \) is closely related to the \( f \) in

\[
dy/dt = f(t,y). \quad \text{(The two differ by an added constant vector and a constant scalar factor.)} \]

By the well-known process of Newton’s method, we generate successive approximations to the desired solution vector \( y \) by substituting instead an approximate linear system. This reduces the problem to a sequence of large linear systems, which we write as

\[
Ax = b. \quad (26)
\]

Here \( b \) is a vector of residuals [the negative of \( F(y) \) for the current approximation to \( y \)], \( A \) is a matrix closely related to the Jacobian \( J \) of \( f \), 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 iterations. 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 \( Av \) for any given vector \( v \) 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 related to \( f \). This implies that \( |F(y + \epsilon v) - F(y)|/\epsilon \), where \( \epsilon \) is a sufficiently small constant, 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 matrix \( J \) (or \( A \)).

We have written an experimental variant of LSODE that comines the methods described above. When tested, it worked very 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 \( P x = b \) (solve for \( x \), given a vector \( b \)) reasonably efficiently. To solve \( A x = b \), we write an equivalent system, say \((A P^{-1}) \) \((P x) = b\), with a different matrix \( A' = A P^{-1} \) and a different solution vector \( x' = P x \), and apply the Krylov method to the problem \( A' x' = b \).

Each iteration requires the evaluation of a product \( A' v = A P^{-1} v \), but that is achieved by solving \( P w = v \) for \( w \) and then approximating \( A w \) as before. If the iteration converges to a vector \( x' \), then the vector we want is \( x = P^{-1} x' \), or the solution of \( P x = x' \).

Convergence is more likely to occur now, because \( A' \) is close 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, and one can do both with two preconditioners \( P_1 \) and \( P_2 \) whose product approximates \( A \).

We have written another experimental solver, called LSODPK, with a selection of preconditioned Krylov methods to solve the linear system problem.\(^6\) It 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 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 partial differential equation (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 a PDE system representing the combined kinetics and diffusion of a set of chemical species, the chemical-kinetics terms 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 one or the other of the preconditioners alone.

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. In the cases run so far, the size of the ODE system varied up to 38,745, and LSODPK has generated solutions with complete success.

The success of Krylov iteration methods combined with the ODE solver LSODE suggests that the same approach may work for DAE systems. To find out, we are developing an experimental variant of DASSL that combines one of the preconditioned Krylov methods with the BDF method as applied to DAE systems. This project combines two different numerical techniques, each of which is relatively new and whose scope of effectiveness is subject to various uncertainties. Thus, the combination is even more unpredictable, and is likely to provide some surprises.

**Differential/Algebraic Systems**

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

\[
F(t, y, y') = 0.
\]  

(27)

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, and 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 recent results here; they are summarized in greater detail in Reference 7.

In a sense, the more singular a DAE system is, the more difficult it is to solve numerically. The index of a 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, $y' = f(y, x)$, $0 = g(y, x)$] have index one if $g = 0$ can be solved for $x$ given $y$ (that is, if $\frac{dg}{dx}$ is nonsingular) and otherwise have an index higher than one. The index can also be defined for systems that are not expressed in the semiexplicit 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, $\lambda$ 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 $\lambda$ solve the DAE system

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

(28)

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. There is now a large body of research results dealing with the issues of how to solve these high-index systems numerically and with techniques that might be implemented symbolically for rewriting the high-index systems in a lower-index form.

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

$$
\begin{align*}
y &= g(t), \\
x &= y',
\end{align*}
$$

(29)

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 any initial value. 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 few years, we have developed a theoretical framework for understanding the convergence of backward-differentiation formulas (BDFs) applied to general index-one systems. The theory applies also to index-two and index-three systems that can be written in a semi-explicit triangular form that commonly occurs in applications.

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 difficulties even for index-one systems.

We have developed techniques for overcoming these difficulties and the even more severe difficulties with ill-conditioning, scaling, and error estimation that are inherent in the practical solution of high-index systems. These techniques are implemented in DASSL, which uses BDF methods for index-one DAEs. 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.

As we have seen, systems that exhibit frequent discontinuities in time are not well-suited to solution by multistep methods such as BDF. For example, when partial differential equations are discretized by the method of lines and coupled with certain adaptive moving-mesh methods, the result is a DAE system with a discontinuity every time the mesh is rezoned.
Discontinuities can also arise as a consequence of a change of state or material property. BDF methods make use of the values of the solution at several previous times to take the next time step, so they are not very efficient for solving these types of systems. On the other hand, because of their one-step nature, implicit Runge-Kutta methods are in principle ideally suited to this purpose. We found that the coefficients of a Runge-Kutta method must satisfy a set of order conditions, in addition to the usual set of order conditions, to attain the same order of accuracy for index-one DAE systems as they attain for nonstiff systems. For higher-index systems, the order conditions are even more stringent.

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 (Equation 28), the constraint and its first and second time derivatives must be satisfied at the initial time, leading to

\[ \begin{align*}
0 &= x^2 + y^2 - L^2, \\
0 &= xx' + yy', \\
0 &= \lambda L^2 - gy + (x')^2 + (y')^2.
\end{align*} \tag{30} \]

We have developed an algorithm for finding a consistent set of initial conditions for index-one nonlinear systems. This is the first general algorithm of which we are aware that holds promise for finding consistent initial conditions for real scientific problems.

We are writing a monograph on numerical methods for DAEs in collaboration with K. E. Brenan of Aerospace Corp. and S. L. Campbell of North Carolina State University. The monograph, to be completed this year, will be published by Elsevier North-Holland. This work will serve to make recent research in this area more widely accessible to a large audience of engineers, scientists and mathematicians.

Key Words: Adams methods; backward differentiation formula (BDF) method; computer code—DAEpack, LSODA, LSODAR, LSODE, LSODES, LSodi, LSOIT, ODEPACK; differential/algebraic equation (DAE); Euler method—explicit, implicit; numerical method; ordinary differential equation (ODE); Runge-Kutta method; stiff systems.

Notes and References