A review of recent developments in solving ODEs — Source link

Gopal Gupta, Ron Sacks-Davis, Peter E. Tescher

Institutions: Monash University, Clayton campus, Melbourne Institute of Technology

Published on: 01 Mar 1985 - ACM Computing Surveys (ACM)

A Review of Recent Developments in Solving ODEs

GOPAL K. GUPTA

Department of Computer Science, Monash University, Clayton, 3168, Australia

RON SACKS-DAVIS

Department of Computing, Royal Melbourne Institute of Technology, Melbourne, 3001, Australia

PETER E. TISCHER

Department of Computer Science, Monash University, Clayton, 3168, Australia

Mathematical models when simulating the behavior of physical, chemical, and biological systems often include one or more ordinary differential equations (ODEs). To study the system behavior predicted by a model, these equations are usually solved numerically.

Although many of the current methods for solving ODEs were developed around the turn of the century, the past 15 years or so has been a period of intensive research. The emphasis of this survey is on the methods and techniques used in software for solving ODEs.

ODEs can be classified as stiff or nonstiff, and may be stiff for some parts of an interval and nonstiff for others. We discuss stiff equations, why they are difficult to solve, and methods and software for solving both nonstiff and stiff equations. We conclude this review by looking at techniques for dealing with special problems that may arise in some ODEs, for example, discontinuities.

Although important theoretical developments have also taken place, we report only those developments which have directly affected the software and provide a review of this research. We present the basic concepts involved but assume that the reader has some background in numerical computing, such as a first course in numerical methods.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; G.1.7 [**Numerical Analysis**]: Ordinary Differential Equations

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Mathematical software, performance evaluation

INTRODUCTION

When simulating the behavior of physical, chemical and biological systems, mathematical models often include one or more ordinary differential equations (ODEs). To study the system behavior predicted by a model, the equations

CONTENTS

representing the model must be solved. Almost always numerical techniques must be used to obtain approximate solutions to the ODEs since analytical techniques available are not powerful enough to solve any ODEs except the simplest. For example, in the well-known example of simulating the population growth of rabbits and foxes, one of the models leads to the following two equations:

$$\frac{dr}{dt} = 2r - \alpha rf, \qquad r(0) = r_0,$$

$$\frac{df}{dt} = -f + \alpha rf, \qquad f(0) = f_0.$$

The variables r and f represent populations of rabbits and foxes, respectively, and t is the time. α, r_0, and f_0 are given. The derivatives therefore define the rate of population change. α is a scalar that determines how strong the interaction between rabbits and foxes is. If it is 0, the foxes die of starvation and their numbers decrease exponentially (f_0e^{-t}), while the number of rabbits grows exponentially (r_0e^{2t}) with time. The model is not particularly complex, and no time delays have been included. The growth rates at any time t depend only on the number of foxes and rabbits at that time.

It is clear in the above model that the population growth of rabbits and foxes depends on the initial numbers of rabbits and foxes and that the above equations have a family (infinite number) of solutions. Problems like this in which initial values are specified at a particular value of t (t_0) are called *initial value problems* (IVPs). There is another class of problems called *boundary value problems* (BVPs) in which conditions are given at both endpoints rather than just at the initial point t_0. In the above example of rabbits and foxes, for example, if the initial number of rabbits is given and we want to estimate the number of foxes needed to limit the rabbit population to a given number at a specified time in the future, then the problem becomes a boundary value problem. We do not discuss BVPs in this review.

In the next section we present some basic concepts that are required to describe numerical techniques for solving ODEs. We then discuss what the ODE solvers try to do. In Section 3 we discuss how ODE software has been evaluated in the last decade, and the problems associated with testing and comparing codes. In Section 4 we briefly discuss recent developments in solving nonstiff differential equations, giving some details of the available codes. In Section 5 we discuss some of the codes available for solving stiff equations, and some recent developments that have not yet been implemented in production codes. We conclude the review by looking at some special difficulties that may arise when solving ODEs.

We do not attempt to cover all the numerous methods suggested in the literature during the past decade.

1. BASICS

We write a set of ODEs as

$$y' = \frac{dy}{dx} = f(x, y), \qquad y(x_0) = y_0 \qquad (y \text{ a vector}). \tag{1.1}$$

We shall consider the solution of sets of first-order differential equations only. Users interested in solving higher order ODEs can reduce their problem to a set of first-order equations. This is a trivial task (see Dahlquist and Björck [1974, pp. 330–331]). Although higher order equations can sometimes be solved more efficiently directly, very little software is available for doing this.

Most numerical methods for solving a set of ODEs compute a sequence of approximate solutions $y_i \approx y(x_i)$, $i = 1, 2, \ldots, f$, where x_f is the final value of x. Since the method is computing an approximation at each step of the sequence, errors are compounded at every step. To discuss the error behavior of numerical methods, we first define local error and global error.

1.1 Local and Global Error

Let (1.1) be the differential equation being solved, and assume that we have computed solution values $y_1, y_2, \ldots, y_n$ at points $x_1, x_2, \ldots, x_n$, respectively. To compute y_{n+1}, the numerical method attempts to solve the following problem:

$$y' = f(x, y), \qquad y(x_n) = y_n.$$

The numerical method is therefore trying to approximate the curve on which (x_n, y_n) lies, not the curve on which (x_0, y_0) lies. Let the solution curve on which (x_n, y_n) lies be $u_n(x)$ and the original solution curve be $y(x)$. The local error in the computed solution y_{n+1} is now given by

$$l_{n+1} = u_n(x_{n+1}) - y_{n+1}.$$

This is the error made by the numerical method in one step. On the other hand, the global error at any point is the total error in the computed solution at that point. It shows how far the computed solution is from the original solution curve.

We define the global error at x_{n+1} as

$$\begin{aligned} g_{n+1} &= y(x_{n+1}) - y_{n+1} \\ &= y(x_{n+1}) - u_n(x_{n+1}) + l_{n+1}. \end{aligned}$$

Thus the global error has two components: one due to local error at the present step and the other due to local errors at previous steps. The relationship between local errors and global error is a complex one, depending on the problem being solved and the method being used.

1.1.1 The Euler Method

Closely related to the concept of local error is the concept of local truncation (or discretization) error. To explain this concept, we introduce the following simple method, called the Euler method, for computing a numerical solution of equation (1.1):

$$y_{n+1} = y_n + hf(x_n, y_n). \tag{1.2}$$

The local truncation error is defined as the difference between the left- and right-hand sides of (1.2) when the true solution values are substituted for the computed values. Therefore the local truncation error for the Euler method is given by

$$d_{n+1} = y(x_{n+1}) - y(x_n) - hf(x_n, y(x_n)).$$

An expression for d_{n+1} may now be obtained by replacing $y(x_{n+1})$ in the above equation by the following Taylor series expansion:

$$y(x_{n+1}) = y(x_n) + hy'(x_n) + \frac{h^2}{2!}y''(x_n) + \cdots .$$

Since $y'(x) = f(x, y(x))$, we obtain the following expression for d_{n+1}:

$$d_{n+1} = \frac{h^2}{2!}y''(x_n) + \cdots .$$

Local error and local truncation error are both indicators of the accuracy of the formula. They are the same for some formulas and are asymptotically (as $h \longrightarrow 0$) equivalent. It should be pointed out that subtle differences exist in various local error definitions found in the literature; however, in this review we ignore differences between the two for all formulas.

Euler's method is referred to as a *first-order* method since the local error is $O(h^2)$. In general, a method is said to be pth order if the local error is $O(h^{p+1})$. If the local error is $O(h^{p+1})$, then the global error is in general $O(h^p)$ for a stable method.

The Euler method is an *explicit* method since it defines y_{n+1} explicitly. A simple *implicit* method called the backward Euler method is defined as follows:

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1}).$$

To compute the solution using an implicit method, it is necessary to solve a set of nonlinear algebraic equations at each step. For the backward Euler method, we need to solve

$$y_{n+1} - y_n - hf(x_{n+1}, y_{n+1}) = 0.$$

One simple technique for solving the above equations is the following iterative scheme:

$$y_{n+1}^{(m+1)} = y_n + hf(x_{n+1}, y_{n+1}^{(m)}).$$

These iterations are called the *simple iterations*. A starting approximation $y_{n+1}^{(0)}$ is needed to begin the iterations.

Since implicit methods involve the additional cost of solving the algebraic equations, an explicit method is often preferred. However, implicit methods are necessary for some types of problems and usually have better stability properties.

1.2 Stability

To study stability of a formula, it is often useful to analyze its performance on the following test problem:

$$y' = \lambda y, \qquad y(x_0) = y_0. \tag{1.3}$$

The analytical solution of the above equation is $y(x) = y_0 e^{\lambda(x-x_0)}$ so that for real λ, the solution grows exponentially if $\lambda > 0$, but decays exponentially for $\lambda < 0$. This simple test problem has traditionally been used for stability analysis, since we can easily obtain analytic expressions describing the solution produced by the numerical method. Studying the behavior of a numerical method in solving this problem is also useful in predicting its behavior in solving other problems, since we may approximate the equation $y' = f(x, y)$ by

$$y' = \frac{\partial f}{\partial y}(y - y_0) + \frac{\partial f}{\partial x}(x - x_0) + f(x_0, y_0).$$

Over a small interval $(x_0, x_0 + h)$, we may approximate $\partial f/\partial y$ by λ if we are dealing with only one equation. We may then write the above equation as

$$y' = \lambda(y - y_0) + F(x_0, y_0). \tag{1.4}$$

The term $F(x_0, y_0)$ rarely affects stability. Thus the test problem (1.3) serves as a good model for studying the general case $y' = f(x, y)$ on a small interval $[x_0, x_0 + h]$. If we are dealing with a set of equations, $\partial f/\partial y$ is a matrix called the *Jacobian* of f, and then we may write

$$y' = J(y - y_0) + F(x_0, y_0).$$

Here J is the Jacobian of f. The above system of equations may be transformed to a set of equations like (1.4) using a principal axis transformation. The parameter λ is then an eigenvalue of J, possibly complex.

We now study the behavior of the solution computed by the Euler method for the test problem (1.3). We obtain the following difference equation after substituting λy_n for $f(x_n, y_n)$ in (1.2):

$$y_{n+1} = y_n + h\lambda y_n.$$

The ratio of the computed solutions at x_{n+1} and x_n is given by

$$\frac{y_{n+1}}{y_n} = 1 + h\lambda.$$

We compare this with the ratio of the true solutions at x_{n+1} and x_n, which is

$$\frac{y(x_{n+1})}{y(x_n)} = \frac{e^{\lambda x_{n+1}}}{e^{\lambda x_n}} = e^{h\lambda}.$$

Suppose that λ is real. Then $1 + h\lambda$ is a reasonable approximation to $e^{h\lambda}$ except if $h\lambda < -2$. For large negative $h\lambda$, $e^{h\lambda}$ is much smaller than 1, while $1 + h\lambda$ is (in magnitude) greater than 1.

This means that the numerical solution is growing while the true solution is decaying, and therefore we say that the numerical solution produced by Euler's method for $h\lambda < -2$ is unstable.

Another way of looking at stability is to look at the propagation of local errors. For example, for Euler's method the following expression may be obtained:

$$g_{n+1} = \left(1 + h\frac{\partial f}{\partial y}\right) g_n,$$

where $g_n = y(x_n) - y_n$. If $\partial f/\partial y = \lambda$, we say that the errors are growing if $|1 + h\lambda| > 1$. Relative to the solution, the errors grow if $h\lambda < -2$ and, for such step sizes, the formula is called unstable.

We now define some stability concepts. A formula is called *stable* if the computed solution for $h\lambda = 0$ does not grow. A formula is called *A-stable* if it produces decaying solutions for all $h\lambda$ values with $\text{Re}(h\lambda) < 0$. A formula is called $A(\alpha)$-stable if it produces decaying solutions for all $h\lambda$ values such that $\arg(-\lambda) < \alpha$, $h\lambda \neq 0$. The backward Euler method is an example of an A-stable method.

1.3 Stiffness

We now briefly discuss stiffness and the difficulties involved in solving stiff equations. As discussed earlier, a set of ODEs $y' = f(x, y)$ may be approximated by $y' = J(y - y_0) + C$ over a small interval. If J is diagonalizable, these equations may be transformed into $z' = D(z - z_0) + C^*$, where D is a diagonal matrix with eigenvalues λ_i of J (possibly complex) on its diagonal and z is related to y by a principal axis transformation.

Now assume that some eigenvalues λ_i are negative and quite large in magnitude in comparison with the others. This implies that some components of the solution will decay very quickly and, for all practical purposes, may become zero. For components that are insignificant, we are usually interested only in their size, and do not need to compute them accurately.

Suppose that we use an explicit method like Euler's method to solve such a problem, where, for example, J has an eigenvalue $\lambda = -10^6$. In order that the solution component corresponding to this eigenvalue not grow, the step size h must be restricted so that $h < 2 \times 10^{-6}$. It is possible that accurate approximations to the other solution components might be obtained with step sizes much greater than 2×10^{-6}. In this case, it is the stability requirements rather than the accuracy requirements that are limiting the step size.

The same situation can occur for complex eigenvalues with negative real parts. The problem is that Euler's method is not A-stable or even $A(\alpha)$-stable for any $\alpha < \pi/2$. The same is true for all explicit methods like Euler's method: No such explicit method can be $A(\alpha)$-stable. We are therefore forced to use implicit methods, like the backward Euler method, to solve stiff systems. These methods require more work per step than explicit methods, however, since a system of nonlinear algebraic equations must be solved at each step.

Stiffness is a difficult concept to define. To some extent, the definition of an equation's stiffness depends on the method being used to solve it. For example, consider the restriction on the step size $h < 2 \times 10^{-6}$ introduced above in connection with Euler's method. The factor 10^{-6} came from one of the eigenvalues of J (i.e., from the problem being solved), but the factor 2 is due to the method. Stiff equations can also be defined solely in terms of the eigenvalues of the Jacobian of the equations; they are characterized by the Jacobian having widely separated eigenvalues and having some eigenvalues with negative real parts of large modulus. For a detailed discussion of stiffness we refer the reader to Shampine [1982d], Lambert [1980], and Shampine and Gear [1979].

Equations that are not stiff are called nonstiff. Solutions to nonstiff equations are easier to obtain, and codes based on classical methods such as Adams formulas or Runge–Kutta formulas (RKFs) may be used. If these methods are used for solving stiff equations, they become very inefficient, since the step size is controlled by stability rather than accuracy requirements. For some problems, the classical methods may require several orders of magnitude more time than special methods designed for solving stiff equations. On the other hand, in those regions of problems in which accuracy is the dominant constraint on the step size, the classical methods are more efficient than stiff methods.

Shampine and Gear [1979] discuss ways of determining whether equations are stiff. Very stable systems are likely to be stiff, for example, as are equations in which some variables change on time scales very different from others. However, it is not always possible to recognize a stiff system of equations since a system may be stiff in one interval and nonstiff in another. Shampine [1983] discusses measuring the stiffness of a given problem.

The problems involved in solving stiff equations were probably first identified by Curtis and Hirschfelder [1952] although Dahlquist [1963] was instrumental in bringing the problem to the attention of the numerical computing community. The importance of stiff equations is discussed by Shampine and Gear [1979] and Bjurel et al. [1970], who present a comprehensive survey of application areas in which stiff equations arise. A more recent survey is presented by Aiken [1982].

2. WHAT THE CODES TRY TO DO

Anyone interested in solving a (set of) differential equations will usually want to compute the solution of (1.1) from an initial x value x_0 to some final value x_f. One may be interested in the solution only at the final point x_f or want to compute the solution at several intermediate points. In some cases the concern may be to find the x value at which a component of y achieves a particular value. Interest may lie not only in obtaining a solution, but also in obtaining the solution to a specified accuracy. In addition to the complexities introduced by these considerations, the differential equations being solved may be stiff without one even being aware of it.

Most software available for solving ODEs may be divided in two classes—those for solving stiff equations and those for solving nonstiff equations. Some codes are capable of solving both stiff and nonstiff equations efficiently, but with few exceptions, the codes require the user to choose the option appropriate to his or her problem. The reason for having separate codes for stiff and nonstiff equations is that the two problem types require very different kinds of methods. However, there are good reasons for combining the two algorithms into one code; the user can then solve either type of problem and switch between methods, either manually or automatically. Codes that detect stiffness and automatically choose the appropriate method are called *type insensitive.* We discuss such codes in Section 5.

Shampine and Watts [1984] discuss several software issues related to ODE solvers: First, it is important to recognize how the codes attempt to control the error in the computed solution. As noted previously, all ODE solvers compute the numerical solution using step-by-step methods.

The code must control the error at each step so that it never exceeds a user-specified tolerance, and at the same time, for reasons of efficiency, the code must take as large a step size as possible. Most modern codes vary the size of the step whenever possible, and base the choice of the new step size on the estimate of the error made in the previous step. When that error estimate is far smaller than the specified tolerance, an increase in the step size is usually made.

Most codes attempt to control only the local error made at each step and by controlling this error, expect to control the global error. Usually an ODE solver attempts to keep its estimate of the local error at each step, l_n, below $K\epsilon$, where ϵ is the user-specified accuracy requirement and K is some safety factor, typically 0.8. This is referred to as error per step control. However, a case can be made that a code taking larger step sizes should be able to allow larger errors at each step and still meet the user's requirements. Therefore it has sometimes been suggested that the local error should be controlled such that $l_n < K^*h\epsilon$. This is called local error per unit step control. Although error control per unit step seems desirable, it can lead to difficulties when the solution is changing rapidly and a low-order formula is being used.

It is also possible to control the global error and attempt to meet the user requirement more closely. However, global error control costs more, since the differential equations must be integrated several times. A less costly approach is to estimate the global error in the computed solution, rather than to control it. Techniques of global error estimation are discussed by Stetter [1974, 1978, 1979a], Shampine and Watts [1976], Zadunaisky [1976], Dew and West [1979], Dahlquist [1981], and Shampine [1982e].

Although controlling and estimating error in computed solutions is the most important issue in designing ODE codes, there are many other important considerations in making the codes robust as well as easy to use. For example, most codes now compute a starting step size to relieve the user from having to make this choice [Watts 1983]. Many codes warn the user if he or she asks for too much accuracy or too much output, or if the integration is taking too long.

Another particularly important issue for stiff ODE software is how to solve the set of nonlinear algebraic equations that must be solved at each step when using an implicit method. A code usually provides several options, so that the structure of the Jacobian can be exploited whenever possible. We discuss this in more detail in Section 5.

If ODE solvers are to be used as black boxes, it is desirable that all ODE solvers have a very similar appearance to the user. A standard user interface will also make it easier for the user to change over to new software when better software becomes available. Hull and Enright [1974] made an early attempt to define a standard user interface for ODE solvers, and other attempts have been made since to design a standard user interface acceptable to ODE software designers as well as the user community. Although a standard interface has not evolved, these attempts have led to similar interfaces for most ODE solvers. ODEPACK interface is discussed by Hindmarsh [1983], while the user interface of the DEPAC collection of ODE solvers at the Sandia Laboratories is discussed by Shampine and Watts [1980]. Gladwell [1979] discusses the initial value programs in the Numerical Algorithms Group (NAG) library.

3. EVALUATION OF ODE SOFTWARE

As we shall see in the next two sections, there have been many methods proposed for solving ODEs. Although a small subset of the proposed methods has been implemented in software that is robust, efficient, and portable enough to be made available to the general mathematical software community, there is enough software for solving ODEs to make it almost impossible to select the "best" code without extensive testing. It is therefore essential to systematically evaluate and compare software so that only the best codes are made available to the user community.

This can be a difficult task. Each code usually consists of a set of formulas and techniques to make the code efficient, reliable, and user friendly, and it is difficult to find an appropriate measure of efficiency for ODE solvers. To the user, a code is more efficient if it solves his or her problem in less CPU time. However, this criterion is both machine and problem dependent, and therefore not adequate for the evaluation of a code that will be used by many users. A code will be used to solve small and simple problems as well as large and complex ones. How do we measure its efficiency?

The best-known work in evaluating nonstiff ODE software is that of Hull et al. [1972]. Enright and Hull [1976a], Krogh [1973], and Shampine et al. [1975] have also tested nonstiff ODE solvers. Stiff ODE solvers have been tested by Ehle [1972], Enright et al. [1975], Enright and Hull [1976b], and Gladwell et al. [1979] and for a special class of problems by Gaffney [1984]. The evaluation of codes has relied very heavily on using a battery of test problems.

Two packages, DETEST, described by Hall et al. [1973], and STIFF DETEST, described by Bedet et al. [1975], have been produced at the University of Toronto and have had a great influence on the development of ODE software. Each original testing package has a collection of 25 test problems divided into five classes. Another class of problems was later added to each package, making a set of 30 test problems in each. To test an ODE solver, the appropriate package uses the code to solve the set of test problems at various accuracy requirements. Statistics are collected on CPU time, overhead time, number of function evaluations, number of steps, and, for stiff equations, the number of Jacobian evaluations and the number of *LU* decompositions. In addition, the packages monitor how well the code being tested controls the local and the global errors. A summary of statistics is produced for the code being tested.

This approach, of course, has its weaknesses. For example, the test problems have to be small systems so that testing is not too expensive. The relative efficiency of ODE solvers is not important for small problems, since a relatively less efficient ODE solver will still produce a solution at an acceptable cost. While the relative efficiency of ODE solvers is of great importance for large systems, extrapolating the results of solving small systems to large systems is not always possible. Also, the presentation and interpretation of results obtained from battery testing can be difficult, since one code may do better on one set of problems and another may solve some other set more efficiently. In addition, if a code fails to solve one or more problems in the set, comparison of codes becomes very difficult. Recently Enright [1982] has discussed some of the problems in testing stiff ODE solvers, and Shampine [1981a] has critically looked at the 30 test problems used in STIFF DETEST and made suggestions to improve them.

In earlier testing, it was assumed that the set of formulas being used was the most important consideration in determining the performance of a code. This was the approach used by Hull et al. [1972], who numerically tested codes for nonstiff equations after modifying them so that each code attempted to do the same task. However, numerical testing by Shampine et al. [1975] has shown that the performance of a code can be seriously affected by other implementation considerations as well. As we have noted, testing and comparing codes is a complex problem, since codes differ not only in the task that they are performing but also in robustness, flexibility, ease of use, etc., and using test batteries to compare codes appears to be adequate only if the codes are similar.

4. METHODS FOR SOLVING NONSTIFF EQUATIONS

Most widely used codes for solving nonstiff ODEs are based on the Runge–Kutta, Adams, or extrapolation methods. We briefly discuss these three classes of methods and the codes based on them.

4.1 Runge–Kutta Methods

The most widely used one-step methods are the Runge–Kutta formulas (RKFs). The RKF computes y_{n+1}, the numerical approximation at $x_n + h$, using the formula

$$y_{n+1} = y_n + h \sum_{i=1}^{r} b_i f(x_n + c_i h, Y_i),$$

where the internal approximations $Y_1, Y_2, \ldots, Y_r$ are given by

$$Y_i = y_n + h \sum_{j=1}^{r} a_{ij} f(x_n + c_j h, Y_j), \qquad i = 1, 2, \ldots, r.$$

Such a formula is called an r-stage RKF. For each internal approximation Y_i at least one function evaluation is required. Therefore an r-stage RKF requires at least r function evaluations per step.

The coefficients of a RKF are usually represented in the following tabular form:

$$\begin{array}{c|cccc}
c_1 & a_{11} & a_{12} & \cdots & a_{1r} \\
c_2 & a_{21} & a_{22} & \cdots & a_{2r} \\
\vdots & \vdots & \vdots & & \vdots \\
c_r & a_{r1} & a_{r2} & \cdots & a_{rr} \\
\hline
 & b_1 & b_2 & \cdots & b_r
\end{array}$$

The RKF is explicit if $a_{ij} = 0$ for $j \geq i$ and otherwise implicit. Explicit RKFs have been extensively used for solving nonstiff equations because they are easy to program. Because RKFs are one-step methods, there is no need for a special starting procedure or for the generation of past values when the step size is changed. Since the formulas are explicit, there are none of the complications involved in solving implicit equations at each step. The main drawback with explicit RKFs is that they require at least k function evaluations per step for a formula of order k. This is in contrast to Adams methods (discussed in the next section), which usually need no more than two function evaluations per step irrespective of the order. Thus explicit RKFs are not expected to be competitive with Adams methods when function evaluations are relatively expensive.

Most recent codes use a pair of RKFs for error estimation as suggested by Fehlberg [1969]. Two numerical solutions are computed by using two formulas of different orders. The error in the solution computed by the lower order formula is estimated by the difference in the two numerical solutions. This approach to error estimation is known as using an RK–Fehlberg pair. The higher order formula is constructed so that all the internal approximations that are required for the first formula are also used by the second formula. The higher order formula generally requires at least one additional internal approximation and therefore at least one more function evaluation. Several sets of RK–Fehlberg pairs have been derived by Verner [1978].

Since we assume that the higher order solution computed for error estimation is more accurate than the lower order solution, we can argue that the higher order solution is the one that should be used as the approximation at the current step. The process of correcting the numerical solution by adding an estimate of the error in that solution is known as local extrapolation. Accepting the higher order solution is local extrapolation, and this device is frequently employed when using RK–Fehlberg pairs. In addition to obtaining a more accurate solution, it turns out that the stability properties improve slightly for explicit RKFs as the order increases.

For a code to be efficient over a wide range of accuracy requirements, it is important that it be able to vary the order of the formulas being used. Such variable-order codes must estimate the error that would have resulted in using formulas of different orders. Most codes that implement RKFs are fixed-order codes. It is possible to design variable order RF–Fehlberg codes since embedded RKFs have been derived by Bettis [1978] and Verner [1979]. Bettis derives a set of embedded formulas from order 1 to 6, while Verner derives five sets of embedded formulas, which include two sets of formulas up to order 5, one set up to order 6, another up to order 7, and the last one has embedded formulas up to order 8.

The difficulty with using such embedded formulas is that a greater number of function evaluations are required, as is illustrated by the last set in Verner [1979]. It is a set of formulas up to order 8, and the formula pair with orders 7 and 8 requires 13 function evaluations. This is the number of function evaluations used by other RK–Fehlberg pairs of the same order. However, the embedded 6, 7 pair requires 12 function evaluations, the 5, 6 pair requires 12 evaluations, and the 4, 5 pair requires 9. RK–Fehlberg pairs for these orders requiring only 10, 8, and 6 evaluations, respectively, have been derived by Verner [1978]. Thus the embedded formula pairs for a given order require a greater number of stages in order that the code can consider a possible increase in order. A variable-order RK–Fehlberg code using a formula pair with orders 7 and 8 has been designed by Shampine et al. [1980]. The formula pair with orders 7 and 8 has two embedded formula pairs with orders 3 and 4: One of these embedded formulas requires 4 function evaluations and the other requires 13.

Enright and Hull [1976a] tested a number of fixed-order codes based on RK–Fehlberg pairs and concluded that these methods were best when derivative evaluations were relatively inexpensive. Shampine et al. [1975] considered only two RKF codes, both of which used the same RK–Fehlberg pair. The code RKF4, by Hull and Enright [1974], accepted the fourth-order solution, while the code RKF45, by Shampine and Watts [1977, 1979], in effect performed local extrapolation by accepting the fifth-order solution. Numerical testing showed that the code which performed local extrapolation was more efficient.

Gupta [1980] has shown that high-order RK–Fehlberg pairs are quite efficient in many application environments. He concludes that if the cost of a function evaluation is less than the equivalent of 5 square root or exponential evaluations per equation, then a suitable RK–Fehlberg code is likely to be more efficient than an Adams code. It is not until the cost of function evaluations is more than the equivalent of about 50 square root or exponential evaluations per equation that Adams codes are clearly more efficient. The costs for many problems lie between these extremes, and for these problems the relative efficiency of a RK–Fehlberg code to an Adams code is difficult to predict. In addition to the cost of each equation, the number of equations being solved, the accuracy requirements, the compiler, and the machine being used will also affect performance, and considerations such as the required frequency of output must also be considered before deciding which code will be more efficient.

Currently, two of the most widely used RK–Fehlberg codes are RKF45 of Shampine and Watts [1977] and DVERK of Hull et al. [1976]. DVERK is based on a pair of formulas of orders 5 and 6 derived by Verner [1978] and is available in the International Mathematical and Statistical Libraries (IMSL). RKF45 is available in the book by Forsythe et al. [1977].

4.2 Linear Multistep Methods

A k-step linear multistep formula (LMF) computes the solution, y_{n+1}, at the current point from values y_{n+1-i} and $f(x_{n+1-i}, y_{n+1-i})$, $i = 1, 2, \ldots, k$, calculated at k previous points. The general form of a LMF is

$$\sum_{i=0}^{k} \alpha_i y_{n+1-i} = h \sum_{i=0}^{k} \beta_i f(x_{n+1-i}, y_{n+1-i}).$$

Although many sets of LMFs have been derived, most LMF codes are based on the Adams formulas. Some of these codes are listed in Table 1. In this section we briefly discuss the theoretical basis of the Adams formulas before describing some of the strategies used by Adams codes.

The Adams formulas are derived as follows. Suppose that we have past approximations y_{n+1-j} to the true solution $y(x_{n+1-j}), j = 1, 2, \ldots$ and that we have approximations $f_{n+1-j} = f(x_{n+1-j}, y_{n+1-j})$ to the derivatives at these points. It is required to advance a step from x_n to x_{n+1} with step size $h_{n+1} = x_{n+1} - x_n$. Using the identity

$$y(x) = y(x_n) + \int_{x_n}^{x} f(t, y(t))\,dt,$$

the kth-order Adams–Bashforth formulas may be regarded as approximating $f(t, y(t))$ over the interval $[x_n, x_{n+1}]$ by a polynomial $Q_{k,n}(t)$ of degree $k - 1$, which interpolates the previously computed values $f_{n+1-j}, j = 1, 2, \ldots, k$. Thus the kth-order Adams–Bashforth formula is

$$y^p_{n+1} = y_n + \int_{x_n}^{x_{n+1}} Q_{k,n}(t)\,dt. \tag{4.1}$$

Similarly, if we define $P_{k,n}(t)$ as the polynomial of degree $k - 1$ that interpolates $f_{n+1-j}, j = 0, 1, \ldots, k - 1$, we derive the kth-order Adams–Moulton formula

$$y_{n+1} = y_n + \int_{x_n}^{x_{n+1}} P_{k,n}(t)\,dt. \tag{4.2}$$

The Adams–Moulton formulas are implicit, and consequently a set of nonlinear equations must be solved for y_{n+1}. Rather than writing y_{n+1} in terms of y_n as in (4.2), we may express y_{n+1} in terms of y^p_{n+1} as follows. Subtracting (4.1) from (4.2) gives

$$y_{n+1} = y^p_{n+1} + \int_{x_n}^{x_{n+1}} (P_{k,n}(t) - Q_{k,n}(t))\,dt.$$

The polynomials $P_{k,n}(t)$ and $Q_{k,n}(t)$ are of degree $k - 1$ and are equal at the $k - 1$ points $x_{n+1-j}, j = 1, 2, \ldots, k - 1$. Let $f^p_{n+1} = Q_{k,n}(x_{n+1})$. Then we may write

$$y_{n+1} = y^p_{n+1} + g_{k,n+1} h_{n+1}(f_{n+1} - f^p_{n+1}), \tag{4.3}$$

Table 1. Codes Based on Adams Formulas

Code	Reference
DVDQ	Krogh [1969]
DIFSUB	Gear [1971b, 1971c]
GEAR	Hindmarsh [1974]
VOAS	Sedgwick [1973]
EPISODE	Byrne and Hindmarsh [1975]
STEP	Shampine and Gordon [1975]
LSODE	Hindmarsh [1980]

where

$$g_{k,n+1} = \frac{1}{h_{n+1}} \int_{x_n}^{x_{n+1}} \frac{(t - x_n)\cdots(t - x_{n-k+1})}{(x_{n+1} - x_n)\cdots(x_{n+1} - x_{n-k+1})}\, dt.$$

The nonlinear equation (4.3) is solved at each step of an Adams code. When solving nonstiff equations, functional (or "simple") iteration is used to solve equation (4.3). Thus we form the approximations

$$y_{n+1}^{(m+1)} = y_{n+1}^p + g_{k,n+1}h_{n+1}(f(x_{n+1}, y_{n+1}^{(m)}) - f_{n+1}^p), \qquad m = 0, 1, \ldots,$$

where $y_{n+1}^{(0)} = y_{n+1}^p$.

The codes DIFSUB, GEAR, and LSODE iterate to convergence (i.e., until the difference between two successive approximations is sufficiently small) and allow a maximum of three iterations before a step is rejected. We may therefore refer to these codes as Adams–Moulton codes. However, the codes VOAS and STEP iterate only once and are referred to as PECE codes since each step consists of the four stages: Compute the *P*redicted value y_{n+1}^p; *E*valuate $f(x_{n+1}, y_{n+1}^p)$; compute the *C*orrected value $y_{n+1} = y_{n+1}^{(1)}$; *E*valuate $f(x_{n+1}, y_{n+1}^{(1)})$. After computing y_{n+1} the code will form an estimate of the local error. This estimate may be obtained by comparing the results of formulas of different orders. Let y_{n+1}^+ be the solution obtained from the Adams–Moulton formula of order $k + 1$; thus

$$y_{n+1}^+ = y_n + \int_{x_n}^{x_{n+1}} P_{k+1,n}(t)\, dt, \tag{4.4}$$

and we may use

$$e_{n+1} = y_{n+1}^+ - y_{n+1}$$

as an estimate of the local error. e_{n+1} may be computed as a difference between predicted and computed values as follows. From (4.4) and (4.1)

$$y_{n+1}^+ = y_{n+1}^p + \int_{x_n}^{x_{n+1}} (P_{k+1,n}(t) - Q_{k,n}(t))\, dt.$$

Thus

$$y_{n+1}^+ = y_{n+1}^p + g_{k+1,n+1}h_{n+1}(f_{n+1} - f_{n+1}^p). \tag{4.5}$$

From (4.3) and (4.5) we have

$$e_{n+1} = (g_{k+1,n+1} - g_{k,n+1})h_{n+1}(f_{n+1} - f_{n+1}^p).$$

The codes DVDQ and STEP actually accept the higher order value, y_{n+1}^+, as the approximation to $y(x_{n+1})$. Thus these PECE codes predict using an Adams–

Bashforth formula of order k and correct using an Adams–Moulton formula of order $k + 1$. Alternatively, they can be viewed as correcting the approximation y_{n+1} of order k by adding in the estimated error to form

$$y_{n+1} + e_{n+1} = \overset{+}{y}_{n+1};$$

this is again local extrapolation.

After computing the estimate of the local error, the codes will choose an appropriate step size and order for the next step. All modern Adams codes are variable order. Since at the start of the integration, values from only one point are available, variable order codes usually start by using a one-step formula and then let the order-changing technique take care of the order being used.

Codes vary the step size to take as large a step as possible consistent with the requirement that the local errors be less than a user-supplied tolerance. Two techniques are commonly used to implement variable-step size Adams methods. One technique assumes that the points x_{n+1-i}, $i = 1, 2, \ldots, k - 1$ in (4.1) and (4.2) are equally spaced. We can then express the Adams formulas in the conventional form

$$y_{n+1} = y_n + h \sum_{j=0}^{k} \beta_j f_{n+1-j}, \tag{4.6}$$

where the coefficients β_j, $j = 0, 1, \ldots, k$ are constant. In this approach, called a fixed-coefficient implementation, interpolation is used to generate approximations to the solution at evenly spaced points when a step size change is required. The other technique, called a variable-coefficient implementation, does not require past values to be equally spaced. In this case we may still express the Adams formulas in the form (4.6), but the coefficients β_j now depend on the step sizes used in the past steps.

The amount of work that each implementation requires at each step differs. For a variable-coefficient, k-step Adams formula, the coefficients β_j must be recalculated if the step size has changed during the past k steps. On the other hand, for fixed-coefficient formulas, the coefficients can be precomputed, but interpolation must be performed when the step size is changed.

Gear and Tu [1974] show that codes based on fixed-coefficient implementations must restrict the frequency with which the step size or the order is changed in order to remain stable. Both theoretical and empirical results for the techniques indicate that the variable-coefficient implementations have better stability properties [Brayton et al. 1972; Gear and Tu 1974; Piotrowski 1969]. No such restriction is required for a variable-coefficient implementation [Gear and Tu 1974; Piotrowski 1969], and variable-coefficient codes can change the step size more frequently.

Another feature that distinguishes the codes of Table 1 is the way they represent the polynomials $Q_{k,n}$ and $P_{k,n}$ of (4.1) and (4.2). Rather than representing these polynomials in ordinate form by storing the past values f_{n+1-j}, $j = 1, 2, \ldots, k - 1$, two popular representations are based on the use of either a vector of divided differences or a Nordsieck vector.

If the divided differences $[f_n; f_{n-1}; \ldots; f_{n+1-j}]$, $j = 1, 2, \ldots, k$ are used, then it is easy to update the representation of $Q_{k,n}(t)$ from one step to the next, even after a change of step size or order. Furthermore, the divided differences are useful for estimating the local errors at different orders. This is the approach taken in VOAS.

The codes DVDQ and STEP use a divided difference representation but store the scaled quantities $h_{n+1}(h_{n+1} + h_n) \cdots (h_{n+1} + h_n + \cdots + h_{n+2-j})\,[f_n; f_{n-1}; \ldots;$

Table 2. Properties of Adams Codes

Code	Type	Local extrapolation	Step size strategy	Representation
DVDQ, STEP	PECE	Yes	Variable coefficient	Scaled divided difference
DIFSUB, GEAR, and LSODE	AM	No	Fixed coefficient	Nordsieck
EPISODE	AM	No	Variable coefficient	Nordsieck
VOAS	PECE	No	Variable coefficient	Divided difference

$f_{n-j+1}]$ rather than the unscaled values. Since large values of the derivatives usually correspond to small step sizes, the scaled divided differences are less likely to vary in magnitude as much as the unscaled quantities.

Another popular representation of $Q_{k,n}(t)$ was first proposed by Nordsieck [1962] and was popularized by Gear [1971a] and is based on storing the scaled derivatives $h^j Q_{k,n}^{(j)}(t)/j!, j = 1, 2, \ldots, k$, in what is often called a Nordsieck history vector. This is used in all the other Adams codes in Table 1. Jackson and Sedgwick [1977] show that the ordinate, divided difference, and Nordsieck representations are all related by coordinate transformations.

If it is required to output the solution at user-specified points, then interpolation of the computed solution values is required. If the solution y_{out} is required at $x = x_{\text{out}}$, where $x_n \le x_{\text{out}} \le x_{n+1}$, then y_{out} may be calculated from the formula

$$y_{\text{out}} = y_n + \int_{x_n}^{x_{\text{out}}} P_{k,n}(t)\, dt.$$

Since a representation of $P_{k,n}(t)$ has already been computed by the code, y_{out} can be computed very cheaply. Thus Adams codes are well suited to applications that require the output at a large number of points. Stetter [1979b] discusses various strategies for interpolating computed values in Adams codes.

A summary of the different properties of the Adams codes appears in Table 2. The codes GEAR and LSODE are substantially modified versions of the code DIFSUB, developed by Gear [1971b, 1971c]. LSODE [Hindmarsh 1983], the more recent code, is a part of ODEPACK, a collection of codes developed at Lawrence Livermore Laboratories. Variations of these codes are widely available, appearing in both the IMSL and NAG libraries. These codes also contain options for solving stiff problems, which we discuss in Section 5.1.2. Excellent documentation for the code STEP appears in the book by Shampine and Gordon [1975], which describes all the strategies used by STEP and provides a listing of the code. The code also forms part of the DEPAC package developed at Sandia Laboratories.

4.3 Extrapolation Methods

The methods in this section are based on the Richardson extrapolation process. The technique is applicable if the computed solution $y(t, h)$ can be expressed as

$$y(t, h) = y(t) + \sum_{i=1}^{m} \tau_i(t) h^i + O(h^{m+1}).$$

If such a series exists, we may approximate it by some function $R_m(t, h)$ with $m + 1$ unknowns determined by the following requirement:

$$R_m(t, h_j) = y(t, h_j), \qquad j = 0, 1, \ldots, m.$$

$y(t)$ can now be approximated by $R_m(t, 0)$, which is an approximation of order m. $R_m(t, h)$ may be a polynomial or a rational function.

Gragg [1965] and Bulirsch and Stoer [1966] used this idea for solving nonstiff equations. The method that they suggested involves using a modified midpoint rule that has been shown to have an asymptotic expansion having only the even powers of h. A modification of the original code of Bulirsch and Stoer is presented by Fox [1971]. This code has been tested by Hull et al. [1972], Enright and Hull [1976a], and Shampine et al. [1975]. The extrapolation code DIFSY1, developed by Hussels [1973], improves the performance of the original code of Bulirsch and Stoer with respect to the starting step size. Deuflhard [1983] has modified DIFSY1 to produce a code DIFEX1, which is slightly more efficient; DIFEX1 uses polynomial extrapolation, which is more efficient than rational extrapolation.

Extrapolation methods must calculate a series of approximate solutions $y(t, h_j)$, $h_0 > h_1 > \cdots > h_m > 0$. For the extrapolation process to be stable, the step sizes h_j must decrease rapidly, and more work needs to be done in computing $y(t, h_{j+1})$ than $y(t, h_j)$. Unlike linear multistep formulas, to achieve higher orders, the amount of work per step for extrapolation methods must increase substantially. To justify this large amount of work per step, higher order extrapolation methods require that large step sizes be used. The testing of Shampine et al. [1975] shows that the efficiency of extrapolation codes is impaired by factors that constrain the step size, such as stiffness or mild stiffness, lack of smoothness in the true solution (e.g., by the presence of discontinuities in the derivatives of the true solution), or by the need to output the solution at a number of closely spaced, user-specified points. For problems in which high accuracy is required and the solution is smooth, extrapolation codes can use very high-order formulas and can be very efficient when function evaluations are relatively inexpensive.

4.4 Other Methods

We have only discussed the more commonly used methods so far. Several other schemes for solving ODEs have been suggested, and we briefly discuss two developments that seem particularly promising.

4.4.1 Taylor Series Methods

The first of these techniques is based on using the Taylor series expansion to compute approximate solutions to a given ODE. Since we have

$$y(x_{n+1}) = y(x_n) + hy'(x_n) + \frac{h^2}{2!}y''(x_n) + \cdots,$$

the first two terms of the series are known at any initial point. If some more terms of the series could be computed at x_n, a high-order approximation to $y(x_{n+1})$ may be computed. This is a classical technique, and J. C. P. Miller reportedly used it in 1946 by devising recurrence schemes for computing the terms of a Taylor series. More recently, Barton et al. [1971] designed an algorithm to automatically generate a set of recurrence relations for computing higher derivatives of y at x_n, given the formal statement of the differential equation. Several refinements of this scheme have been suggested. For example, Barton [1980] has presented a scheme for solving stiff equations, and Corliss and Chang [1982] describe a Taylor series package ATSMCC, which, unlike earlier Taylor series packages, is a portable package written in FORTRAN. Corliss and Chang report that their package is efficient, especially at stringent accuracy requirements.

Since the Taylor series method deals with high-order ODEs directly, it is likely to be more efficient in solving them than the conventional methods, which require that the higher order equation be converted to a set of first-order equations.

4.4.2 Multirate Methods

The second technique that we discuss here relates to a special class of problems. In some application areas, for example, simulation of certain electrical circuits and some real-time simulations, the set of ODEs to be solved has components of the solution that vary at very different rates. If a conventional ODE solver is used to solve such problems, the step size is chosen to track the fastest component and the slow-changing components are computed too accurately. These problems differ from stiff equations in that here the fast components need to be computed accurately, while in stiff equations the fast-changing components have usually decayed and need not be computed accurately. It may be economical to partition systems with components varying at different rates and integrate each subsystem using a different step size although this introduces additional difficulties. Such methods are called *multirate methods*. For further details about these methods, their advantages and difficulties, we refer the reader to the relevant papers [Andrus 1979; Gear 1974, 1980; Orailoglu 1979; Wells 1982].

5. SOLVING STIFF EQUATIONS

Before discussing some of the methods and software for solving stiff ODEs, we discuss the main difficulties in solving such equations.

5.1 Difficulties in Solving Stiff Equations

As noted earlier, there are two main difficulties in using classical techniques to solve a set of stiff equations:

(a) The stability properties of the classical formulas (e.g., Adams methods or explicit Runge–Kutta methods) are not adequate for the solution of stiff systems.

(b) While many implicit formulas do have adequate stability properties, they must solve a set of nonlinear equations at each step. An inexpensive method for solving nonlinear equations, such as simple iteration, can be used when an implicit formula is used to solve nonstiff ODEs but cannot be used when the equations are stiff.

Most research in stiff equations can be seen as attempts to overcome these two difficulties, and we discuss them in some detail.

5.1.1 Stability

We have already considered an A-stable formula (the backward Euler method in Section 1.3). Ideally, one would like to use a set of A-stable formulas to solve stiff equations, since such formulas are expected to perform in a numerically stable manner whenever the eigenvalues of the Jacobian of the ODEs lie in the left half-plane.

However, it is known that no LMF of order greater than 2 can be A-stable [Dahlquist 1963]. It has also been proved that the maximum order of an A-stable method can not exceed $2q$ if the number of stages or derivatives used in the formula is q [Wanner et al. 1978]. One could use $A(\alpha)$-stable formulas, but an

$A(\alpha)$-stable method can be expected to perform in a stable manner only if the eigenvalues λ_i of the Jacobian are such that $\mathrm{Im}(\lambda_i)/\mathrm{Re}(-\lambda_i) \leq \tan\alpha$. When this is violated, the stability is conditional on the step size h being used.

Because of these constraints, the search for new methods for solving stiff equations has been along the following lines:

(a) Finding LMFs of orders greater than 2 that are as close to being A-stable as possible. It is known that high-order, $A(\alpha)$-stable formulas with α close to 90 degrees exist [Kong 1977].
(b) Finding formulas that are not LMFs and therefore may be A-stable for orders greater than 2. We shall see that this approach usually introduces new difficulties.

Some of the developments along these lines are presented in Sections 5.2–5.7.

5.1.2 Algebraic Equations

We illustrate the difficulties that arise in solving nonlinear algebraic equations by considering the algebraic equations associated with LMFs. The equations for LMFs are straightfoward, and similar issues arise for implicit RKFs and other implicit methods. We follow the notation of Shampine [1979, 1980b].

At each step of an LMF we need to solve the following set of nonlinear algebraic equations:

$$y_{n+1} - h\beta_0 f(x_{n+1}, y_{n+1}) = \phi, \tag{5.1}$$

where ϕ is a constant that groups the contribution from terms at past points. The classical technique of solving the algebraic equations is simple iteration, defined as follows:

$$y_{n+1}^{(m+1)} = h\beta_0 f(x_{n+1}, y_{n+1}^{(m)}) + \phi.$$

In order for these simple iterations to converge, we must have $\| h\beta_0\, \partial f/\partial y \| < 1$. Since for stiff equations $\| \partial f/\partial y \|$ is usually large, the step size must be severely constrained to obtain convergence. To overcome this difficulty, codes for solving stiff equations usually resort to much more expensive schemes based on the Newton–Raphson method.

The Newton scheme for solving the nonlinear algebraic equations (5.1) may be written as

$$(I - h\beta_0 J)(y_{n+1}^{(m+1)} - y_{n+1}^{(m)}) = -y_{n+1}^{(m)} + h\beta_0 f(x_{n+1}, y_{n+1}^{(m)}) + \phi. \tag{5.2}$$

The matrix $(I - h\beta_0 J)$ is usually called the Newton iteration matrix or, more simply, the iteration matrix. Every iteration of the scheme (5.2) involves the following costs:

(a) Evaluation of the Jacobian and the formation of the Newton iteration matrix.
(b) Factorization of the iteration matrix into LU form.
(c) Foward and back substitution to compute the correction to be applied to $y_{n+1}^{(m)}$.

Each of these three items is expensive if N is large. The evaluation of J requires, in general, the evaluation of a matrix with N^2 elements. The Jacobian must often be approximated by numerical differencing since supplying an explicit Jacobian is a tedious and error-prone task. This requires N or $N + 1$ function evaluations. Forming the LU factors of the iteration matrix requires $O(N^3/3)$ operations, while the foward and back substitutions are $O(N^2)$ operations.

Fortunately, the costs incurred at each iteration can be substantially reduced. Since the Jacobian usually varies slowly, one approximation to the iteration matrix may be used for several steps. The resulting iterations are then called the modified Newton (or inexact Newton) iterations. Some of the widely used codes evaluate the Newton iteration matrix only once in every 6–8 steps, although the Jacobian (and possibly the step size and the coefficient β_0) may have changed. Klopfenstein [1971] and Krogh and Stewart [1984] discuss the effect of having an inaccurate Jacobian on the rate of convergence of the iterations.

In practice, very large systems almost always have sparse structure in the coupling of the ODEs, and the ability to exploit this structure is crucial to the success of any given type of stiff method. As an example of savings that may be obtained, a technique suggested by Curtis et al. [1974] may achieve substantial reductions in the cost of approximating the Jacobian by numerical differencing. This technique takes advantage of the known structure of the Jacobian. For example, banded Jacobians of half-bandwidth m can be formed by differencing using only $2m + 1$ extra function evaluations regardless of the size of the system of equations. Coleman et al. [1984] have also developed software for numerically estimating sparse Jacobian matrices.

Substantial effort has gone into finding techniques to reduce the cost of linear algebra when large systems of equations are being solved. Most of this effort has been along the following lines:

(a) Exploiting the structure of the Jacobian so that the linear equations may be solved more economically.
(b) Reducing the cost of updating the LU factors of the iteration matrix when J has not changed much but $h\beta_0$ has.
(c) Solving the algebraic equations using an iterative method rather than a direct method.
(d) Automatically switching between simple iterations and Newton iterations so that the Newton iterations are used only when necessary.
(e) Partitioning the set of equations into stiff and nonstiff subsystems and then using simple iterations for the nonstiff subsystem.
(f) Using formulas that do not require solution of nonlinear algebraic equations.

We discuss these developments in the above order.

(a) One of the major sources of large stiff ODEs is the method of lines technique for the numerical solution of partial differential equations (PDEs). Several packages for solving PDEs using the method of lines were developed in the early 1970s. Some of these packages were based on codes for solving stiff equations with modifications to make them more efficient for banded and sparse Jacobians [Carver 1976; Carver and Baudouin 1976; Chang et al. 1974; Hindmarsh 1981; Seager and Balsdon 1982; Sherman and Hindmarsh 1980; Sincovec and Madsen 1975]. Another early attempt to use sparse software for solving stiff equations related to a simulation package, the CSMP package [Gourlay and Watson 1974].

Hindmarsh describes stiff equation software that provides facilities for solving banded as well as sparse linear equations [Hindmarsh 1983]. Carver and MacEwen [1981] studied the use of sparse matrix techniques in solving large stiff differential systems that arose from using the method of lines to solve PDEs. They show that some problems can be more efficiently solved if the Jacobian is approximated by a sparse matrix.

(b) For LMFs the iteration matrix is of the form $I - h\beta_0 J$. If we choose to retain the scaled iteration matrix $J - (1/h\beta_0)I$, then if either h or β_0 changes, the new scaled iteration matrix differs from the current scaled iteration matrix by only a constant diagonal matrix. If the scaled iteration matrix is kept in its *LU* decomposed form, then the most efficient and stable way to obtain the *LU* decomposition of the new scaled iteration matrix is by forming the new scaled iteration matrix and subsequently its *LU* factors. Unfortunately, in this situation we cannot exploit the fact that the two matrices differ only by a constant diagonal matrix. Enright [1978] has proposed that the scaled iteration matrix be decomposed into LHL^{-1} where L is a lower triangular matrix and H is a Hessenberg matrix. Then if

$$J - c_1 I = LHL^{-1},$$

we have

$$J - c_2 I = L(H + (c_1 - c_2)I)L^{-1}.$$

If the step size or the leading coefficient changes, we can very efficiently update the LHL^{-1} factorized form of the scaled iteration matrix. For a system of N differential equations the LHL^{-1} factorization costs $O(5N^3/6)$ operations as compared to the *LU* factorization cost of $O(N^3/3)$ operations. Solving a linear system with the iteration matrix in LHL^{-1} form costs $O(2N^2)$ operations compared to $O(N^2)$ operations when the matrix is in *LU* form. On the other hand, updating the LHL^{-1} factorization can be done in $O(N)$ operations, while in the *LU* form the cost of a new factorization is $O(N^3)$ operations. In order to be able to solve linear systems more efficiently with the iteration matrix in the LHL^{-1} form, Enright suggests that the Hessenberg matrix itself be decomposed into *LU* factors $L1$ and $H1$, where $L1$ is bidiagonal. This reduces the cost of solving the linear system to $O(3N^2/2)$ operations while increasing the cost of updating the factorization to $O(N^2)$ operations.

The LHL^{-1} factorization can achieve considerable savings if the problem has a constant or slowly changing Jacobian and the number of equations is medium to large. It is not clear how this technique will perform when the Jacobian is not varying slowly. Also most large systems of equations are either banded or sparse, and the LHL^{-1} factorization is not able to exploit these special structural properties.

(c) Nonlinear algebraic equations associated with the implicit method do not need to be solved very accurately. Thus, when the number of equations is large, the use of iterative methods for solving the equations may be more economical than Gaussian elimination. Since a good predicted value of the solution is almost always available, very few iterations will probably be needed. Hindmarsh [1976] has designed software that uses a block successive overrelaxation iterative technique for problems where the Jacobian has a block structure. This technique uses Gaussian elimination on each block. The iterative techniques investigated recently are all based on Krylov subspace methods of which the conjugate gradient is a well-known example. Gear and Saad [1983] and Chan and Jackson [1983] present such techniques. These methods only require computation of the product Jv for any given vector v, and the Jacobian J need not be stored. Chan and Jackson [1984] discuss iterative techniques and concentrate on methods that use an explicit Jacobian and use preconditioning that involves solving an equivalent system $A^*x^* = b^*$ (which is easier to solve) instead of the system $Ax = b$ obtained in Newton iterations. Brown and Hindmarsh [1984] discuss an implementation of the incomplete orthogonalization method (IOM) of Gear and Saad [1983], which requires no matrix storage. Results of tests on equations of sizes up to 16,000 are presented. Preliminary results suggest that iterative techniques are viable and may be much more efficient for large systems, both in terms of cost and storage.

(d) Another way to reduce the cost of linear algebra is to use simple iteration whenever possible rather than Newton iteration. Most stiff problems have transient regions where small step sizes, corresponding to the dominant eigenvalues of the problem, are needed to achieve the desired accuracy. In these regions the step size is so restricted by accuracy requirements that the problem is effectively nonstiff, and simple iteration may be used. Therefore the stiff codes could be made more efficient if they could automatically change to simple iteration whenever possible. Progress has recently been made in designing type-insensitive codes, which attempt to recognize whether the problem is locally stiff or nonstiff and automatically choose a suitable formula and iterative method. The first efforts by Shampine resulted in techniques for Adams and Runge–Kutta codes to decide whether stiffness was the reason they were performing inefficiently [Shampine 1977, 1980a]. More recently, Shampine has designed codes based on implicit A-stable and $A(\alpha)$-stable formulas that can automatically recognize stiffness [Shampine 1981b, 1982a]. Shampine presents results of an experimental code where the convergence rate of the iterative method is monitored. At each step the code tries to select an iterative method that will converge quickly enough. Petzold has also developed an algorithm that detects both stiffness and the absence of stiffness and automatically selects between stiff (backward differentiation formula (BDF)) and nonstiff (Adams) methods [Petzold 1983a]. The algorithm is available in two variants of LSODE, namely LSODA and LSODAR [Hindmarsh 1983].

(e) Often not all components of the stiff system being solved need the Newton iteration for convergence. Enright and Kamel [1979] suggest one scheme in which the nonstiff and stiff components can be partitioned in two separate subsystems. The nonstiff components can then be solved using simple iterations. Watkins and Hansonsmith [1983] suggest a more precise partitioning, provided that the number of stiff components is known and small. Björck [1983] suggests another partitioning technique that uses a modified QR method to factorize the Jacobian.

(f) Although it seems that methods for solving stiff equations must be implicit, there is a class of formulas called *linearly implicit* that are not implicit in the usual sense [Lambert and Sigurdsson 1972]. These formulas may involve less linear algebra than the usual implicit methods. When implicit methods use an approximation to the Jacobian as an aid to solving the algebraic equations at each step, linearly implicit formulas incorporate an approximation to the Jacobian into the actual formula and require the solution of at least one set of linear equations at each step.

As an example, the following formula is a linearly implicit first-order formula:

$$y_{n+1} = y_n + hf(x_n, y_n) + hJ(y_{n+1} - y_n),$$

where J is an approximation to the Jacobian.

The main drawback with linearly implicit formulas is that the nature of the approximation to the Jacobian can affect the stability properties of the method. In the above example, the order of the formula is not affected by the accuracy of J but the stability properties are. If J is the exact Jacobian, then the method is A-stable, but if we replace J by the zero matrix, then the formula reduces to the forward Euler method, which is not even $A(0)$-stable.

5.2 Linear Multistep Methods

For one-step methods, stability properties can be determined using an approach similar to that used in connection with Euler's method in Section 1. For the linear problem $y' = \lambda y$, we can determine the behavior of the ratio of computed solutions at successive points. For LMFs, on the other hand, the behavior of the problem $y' = \lambda y$, is expressed by the following *difference equation*:

$$\sum_{i=0}^{k} \alpha_i y_{n+1-i} = h\lambda \sum_{i=0}^{k} \beta_i y_{n+i-1}.$$

Associated with this difference equation is a characteristic polynomial

$$\sum_{i=0}^{k} (\alpha_i - h\lambda\beta_i) r^i.$$

The LMF will produce decaying solutions for a particular $h\lambda$, if all roots of this polynomial are less than one in magnitude. A description of some of the techniques used to determine the stability regions of a LMF (the region of the $h\lambda$ plane for which decaying solutions are produced) can be found in Lambert [1973].

The most commonly used multistep methods for stiff equations are the backward differentiation formulas (BDFs), discussed by Henrici [1962, Section 5.1–5.4]. Indeed, the BDFs were being used for solving stiff equations by Curtis and Hirschfelder in 1952. Gear [1971b] implemented the BDFs in a variable-order, variable-step size code DIFSUB, which was probably the first such code for solving stiff equations. DIFSUB has proved to be one of the most efficient codes for solving stiff equations and its variants are still in widespread use.

The derivation of the BDFs is discussed by Henrici [1962]. The formulas are based on a polynomial $P_{n+1}(x)$ interpolating through $y_{n-k+1}, \ldots, y_n$ and y_{n+1}. To obtain y_{n+1} we require that $P_{n+1}(x)$ satisfy the differential equation

$$y' = f(x, y)$$

at $x = x_{n+1}$. That is, the additional constraint

$$P'_{n+1}(x_{n+1}) = f(x_{n+1}, y_{n+1})$$

defines the BDFs together with the interpolation conditions and the requirement that $P_{n+1}(x)$ has degree at most k. The first- and second-order BDFs are A-stable, and BDFs of orders up to 6 are $A(\alpha)$-stable with the following α:

Order	3	4	5	6
Stability angle α	86.0	73.4	51.8	17.9

Because of the small stability angle of the sixth-order BDF, some of the codes only use orders up to 5. An important property of the BDFs is that the characteristic polynomials of the formulas have zero roots at infinity (i.e., $h\lambda = \infty$).

In spite of their relatively poor absolute stability properties, many widely used codes for solving stiff equations are based on BDFs. As noted in Section 4.2, GEAR and LSODE are improved versions of DIFSUB produced by Hindmarsh [1974, 1980]. All three codes can handle either stiff or nonstiff equations efficiently. The user, by setting parameters in the calling sequence, can select either the BDFs or the Adams–Moulton formulas as well as an appropriate corrector iteration method. GEAR and LSODE allow several types of corrector iteration methods. Several versions of GEAR and LSODE exist which have been specifically written to solve implicit differential equations or large systems of differential equations where the Jacobian is either banded or sparse. Various versions of LSODE form part of the ODEPACK collection [Hindmarsh 1983].

The codes DIFSUB, GEAR, and LSODE use the Nordsieck representation and implement the BDFs as fixed-coefficient formulas. As discussed in Section 4.2, when fixed-coefficient formulas are used, frequent step size changes can cause instability. Therefore, when using a formula of order q, the above codes keep the step size and order constant for $q + 1$ steps after the step size is changed.

A variable-coefficient implementation of the BDFs is discussed by Brayton et al. [1972], who report that the variable-coefficient implementation was found to be much more stable than the fixed-coefficient implementation. However, the variable-coefficient implementation could also become unstable for some sequences of step size changes. Byrne and Hindmarsh [1975] discuss another implementation using the Nordsieck representation of variable-coefficient BDFs in the code EPISODE. Extensive testing of EPISODE [Byrne et al. 1977] has shown that it may sometimes solve problems that could not be solved by the fixed-coefficient implementation. Curtis [1980] discusses the reasons for this and cites the performance of his code FACSIMILE, which uses a fixed-coefficient implementation of BDFs and does not fail on these problems. He argues that the failure of the fixed-coefficient codes on these problems may actually be due to the heuristics used in the codes and not necessarily to the fixed-coefficient implementation.

The advantages of a variable-coefficient implementation are better stability and therefore the possibility of step size changes at each step. Those advantages do not overcome the two main problems of such codes:

(a) Additional computing is required at the current step to compute the variable coefficients unless the last q steps have been taken with the same order and step size.

(b) More important, if the formula's coefficients are changing at each step, then the Newton iteration matrix $W_{n+1} = I - h_{n+1}\beta_0 J$ also varies. This means that either the iteration matrix must be updated more frequently or the iterations may converge more slowly.

Jackson and Sacks-Davis [1980] have proposed a new scheme of changing step size. They suggest a technique in which the leading coefficient of the formula is unaffected by a change in the step size, and therefore the Newton iteration matrix changes as in a fixed-coefficient implementation. On the other hand, the other coefficients of the formula do depend on past step sizes, and the stability of implementations based on this technique of changing step size approaches that of a variable-coefficient implementation.

Testing of stiff ODE solvers by Enright et al. [1975] and Enright and Hull [1976b] has shown that the BDF codes are the most efficient codes available, except for problems which have Jacobians having eigenvalues with large imaginary parts. The BDF codes become very inefficient for such problems, and a lot of research has been directed toward finding formulas that will perform better than BDF codes for such problems.

One approach to finding new multistep formulas has been to consider k-step LMF, which have all roots zero at infinity but have order less than k, and thus have extra parameters that can be used to improve the $A(\alpha)$-stability angle. This approach has been adopted by Klopfenstein [1971] and more recently Varah [1978] and Skelboe and Christiansen [1981]. For example, Skelboe and Christiansen derive a sixth-order, eight-step formula which has all roots zero at infinity and is $A(55.9°)$-stable. For a given order, better $A(\alpha)$-stability angles are obtained, but always with the added overhead costs of larger local truncation error coefficients.

The second approach has been to consider kth order, k-step LMFs that do not have all roots zero at infinity. Nordsieck [1962] suggested that all LMFs are ultimately equivalent to finding a polynomial of some given degree approximating the solution of the differential equation. Wallace and Gupta [1973] develop a polynomial formulation of multistep methods. They show that a polynomial approximation to the solution is updated at each step by the addition of a multiple of a constant polynomial. This constant polynomial, which they call a modifier polynomial, uniquely determines the LMF. Wallace and Gupta [1973], Gupta [1976], and Gupta and Wallace [1975] have derived a number of families of LMFs. These families consist of kth order, k-step formulas, and can contain $A(\alpha)$-stable formulas with order as high as 12. They also derive $A(\alpha)$-stable formulas of order less than or equal to six that exhibit larger stability angles than the BDFs of the same order.

Gupta [1982] has implemented (in a code called DSTIFF) a family of formulas that use a modifier polynomial based on a least-squares approximation of previously computed values. The code implements formulas that are $A(\alpha)$-stable, $\alpha > 62.78°$ and are stable up to tenth order. The local truncation error coefficients of the formulas increase in magnitude with increasing order, and the modulus of the largest root at infinity also increases with increasing order. For the formulas of sixth order and higher, the magnitude of the largest root at infinity is greater than or equal to 0.79. Gupta [1982] has recently compared the performance of DSTIFF with that of LSODE. His results indicate that DSTIFF is comparable in efficiency to LSODE. For stiff problems in which the eigenvalues of the Jacobian are close to the imaginary axis, DSTIFF is much more efficient. Over a range of problems LSODE generally takes fewer function evaluations and DSTIFF makes fewer Jacobian evaluations. DSTIFF also keeps the local error in the computed solution below the user-specified tolerance much more successfully than does LSODE. In spite of the use of high-order formulas in DSTIFF, LSODE is more efficient at stringent tolerances. This seems partly to result from the large roots at infinity of the high-order formulas used in DSTIFF.

5.3 One-Step Methods

A considerable amount of research in recent years has been directed to the problem of solving stiff systems by using implicit one-step methods. Recently Cash [1982a] has presented a survey of implicit Runge–Kutta methods. In this section we look at some of the more promising developments in using implicit one-step methods for stiff systems.

Butcher [1964a] investigated r-stage implicit Runge–Kutta formulas (RKFs) and showed that such formulas can be of order $2r$. Ehle [1968] proved that an r-stage implicit RKF of order $2r$ is A-stable. The fact that implicit RKFs could be of high order and still be A-stable has attracted a great deal of interest, and several sets of A-stable implicit RKFs have been proposed by Butcher [1964b], Ehle [1968], and Chipman [1971]. The main problem with all these formulas is that the solution of the resulting nonlinear equations is prohibitively expensive.

Consider the following r-stage implicit RKF:

$$y_{n+1} = y_n + h \sum_{i=1}^{r} b_i f(x_n + c_i h,\ Y_i),$$

where the internal approximations Y_i are given by

$$Y_i = y_n + h \sum_{j=1}^{r} a_{ij} f(x_n + c_j h,\ Y_j), \qquad i = 1, 2, \ldots, r. \tag{5.3}$$

In order to advance a step, it is necessary to solve the nonlinear equations (5.3). If there are N differential equations in the system (1.1), then for an r-stage RKF, the nonlinear system (5.3) consists of Nr equations. The solution of (5.3) can be achieved using a modified Newton scheme.

At the kth iteration of the modified Newton scheme we compute the approximation $Y_i^{(k)}$ to Y_i, $i = 1, 2, \ldots, r$. Let the residual $z_i^{(k)}$ associated with $Y_i^{(k-1)}$ be defined by

$$z_i^{(k)} = -Y_i^{(k-1)} + y_n + h \sum_{j=1}^{r} a_{ij} f(x_n + c_j h,\ Y_j^{(k-1)}).$$

Then in the kth iteration we obtain $Y_i^{(k)}$ by

$$Y_i^{(k)} = Y_i^{(k-1)} + w_i^{(k)},$$

where $w_i^{(k)}$, $i = 1, 2, \ldots, r$ are the solution of the following set of linear equations:

$$\begin{bmatrix} I - ha_{11}J & -ha_{12}J & \cdots & -ha_{1r}J \\ -ha_{21}J & I - ha_{22}J & \cdots & -ha_{2r}J \\ \vdots & \vdots & & \vdots \\ -ha_{r1}J & -ha_{r2}J & \cdots & I - ha_{rr}J \end{bmatrix} \begin{bmatrix} w_1^{(k)} \\ w_2^{(k)} \\ \vdots \\ w_r^{(k)} \end{bmatrix} = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_r^{(k)} \end{bmatrix}. \tag{5.4}$$

Here I is the $N \times N$ unit matrix and J is an approximation to the Jacobian matrices $J(x_n + c_j h,\ Y_j^{(k-1)})$. Thus in the system (5.4) we solve for the r vectors $w_i^{(k)}$, $i = 1, 2, \ldots, r$ simultaneously. Each vector $w_i^{(k)}$ has N components.

Even for small systems of differential equations, implicit RKFs implemented in this manner may not be competitive with LMFs. Several attempts have been made to reduce the linear algebra costs associated with implicit RKFs. The earliest successful approach was by Bickart and Picel [1973]. (Although their approach was applied to composite multistep formulas, it is equally applicable to implicit RKFs.) The clearest description of a variation on the theme is that of Butcher [1976], who suggested the following scheme.

Define the $r \times r$ matrix $A = (a_{ij})$ and let the Jordan canonical form of A^{-1} be

$$T^{-1}A^{-1}T = \begin{bmatrix} \lambda_1^{-1} & 0 & 0 & \cdots & 0 \\ \mu_1 & \lambda_2^{-1} & 0 & \cdots & 0 \\ 0 & \mu_2 & \lambda_3^{-1} & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_r^{-1} \end{bmatrix},$$

where μ_i is zero if $\lambda_i \neq \lambda_{i+1}$, $i = 1, 2, \ldots, r-1$, zero or an arbitrary nonzero number if $\lambda_i = \lambda_{i+1}$. Let D be the $r \times r$ diagonal matrix with diagonal entries $\lambda_1, \lambda_2, \ldots, \lambda_r$.

Define the $r \times r$ transformation matrices $P = (p_{ij})$ and $Q = (q_{ij})$ by

$$P = DT^{-1}A^{-1},$$

$$Q = T^{-1},$$

and consider the following transformations:

$$\tilde{w}_i^{(k)} = \sum_{j=1}^{r} q_{ij} w_i^{(k)}, \qquad i = 1, 2, \ldots, r,$$

$$\tilde{z}_i^{(k)} = \sum_{j=1}^{r} p_{ij} z_i^{(k)}, \qquad i = 1, 2, \ldots, r.$$

Butcher [1976] and Bickart [1977] show that when expressed in terms of the transformed variables $\tilde{w}_i^{(k)}$ and $\tilde{z}_i^{(k)}$, the system (5.4) reduces to

$$\begin{bmatrix} I - h\lambda_1 J & 0 & \cdots & 0 \\ -h\mu_1 J & I - h\lambda_2 J & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & I - h\lambda_r J \end{bmatrix} \begin{bmatrix} \tilde{w}_1^{(k)} \\ \tilde{w}_2^{(k)} \\ \vdots \\ \tilde{w}_r^{(k)} \end{bmatrix} = \begin{bmatrix} \tilde{z}_1^{(k)} \\ \tilde{z}_2^{(k)} \\ \vdots \\ \tilde{z}_r^{(k)} \end{bmatrix}. \tag{5.5}$$

When using Butcher's technique, we form the *LU* factors of each of the distinct diagonal blocks. To solve the linear system (5.4) at each iteration, we transform $z_i^{(k)}$ to obtain $\tilde{z}_i^{(k)}$, solve the linear system (5.5), and transform $\tilde{w}_i^{(k)}$ to obtain $w_i^{(k)}$. In solving (5.4) directly we have to solve an $Nr \times Nr$ linear system: By using Butcher's technique we can solve (5.5) by solving r, $N \times N$ linear systems sequentially. Note, when A has complex eigenvalues, we may be forced to use complex arithmetic in order to implement Butcher's technique.

For large differential systems, the cost involved in forming and factoring the iteration matrix is the dominant linear algebra cost, since this is an $O(N^3)$ operation for a system of N equations. For implicit LMFs, this cost is $O(N^3/3)$. When using Butcher's technique, the cost of forming and factoring the iteration matrices is minimized when the matrix A has only one r-fold eigenvalue, in which case only one iteration matrix need be stored. This iteration matrix is of order N, and its *LU* factorization requires $O(N^3/3)$ operations.

Implicit RKFs that are r-stage and whose coefficient matrix (a_{ij}) has only a single r-fold eigenvalue are known as *singly-implicit RKFs* and have been proposed by Burrage [1978] for the solution of stiff equations. The formulas have a maximum attainable order of $r + 1$ for an r-stage method. Burrage [1978] derived singly-implicit RKFs of orders up to 15. The formulas are $A(\alpha)$-stable with $\alpha > 83°$.

At each iteration, a singly-implicit RKF requires two transformations, $\tilde{z}_i^{(k)}$ from $z_i^{(k)}$ and $w_i^{(k)}$ from $\tilde{w}_i^{(k)}$. For an r-stage formula this costs $O(2Nr^2)$ operations. Forming the residuals $z_i^{(k)}$ will require r function evaluations per iteration. For an implicit LMF, solving the linear system at each iteration will require $O(N^2)$ operations with one function evaluation necessary per iteration. For large differential systems a singly-implicit RKF will require roughly r times as much work per step as an implicit LMF, since it will require roughly r times as much linear algebra per iteration and r times as many function evaluations as an implicit LMF. However, the dominant cost is the cost of the *LU* decomposition of the iteration matrix that is normally used for a number of steps, and this cost is the same for LMFs and singly-implicit RKFs.

For small to medium-to-large differential systems, the cost of transforming the quantities $z^{(k)}$ and $\tilde{w}^{(k)}$ becomes comparable to the cost of solving the linear system (5.5). If, for example, there are 20 equations in the differential system and we are using a six-stage, singly-implicit formula, then at each iteration we are doing about half as many operations in transforming variables as in actually solving the linear system (5.5). For $N = 20$, $r = 6$, transforming variables costs $2Nr^2 = 1440$, solving linear systems costs $rN^2 = 2400$, while the $N^3/3$ cost of an LU factorization is $N^3/3 = 2667$.

Burrage et al. [1980] implemented a family of singly-implicit RKFs in a code called STRIDE, using a transformation discussed by Butcher [1979]. Error estimation in the code is handled by embedding an r-stage, rth-order formula within an $(r + 1)$-stage, $(r + 1)$st-order formula. Numerical testing by Gaffney [1984] indicates that STRIDE is expensive with respect to function evaluations, and is only likely to be competitive with BDF-based codes on problems in which the Jacobian of the system has eigenvalues close to the imaginary axis. Gaffney [1982] has also considered the cost of transforming variables in STRIDE, and found that when a stiff system of six equations was solved, approximately 48 percent of the total CPU time was spent in transforming variables versus 34 percent in solving linear equations.

Nørsett [1974] also proposed the use of a class of RKFs with reduced linear algebra costs, called *semi-implicit RKFs.* For semi-implicit RKFs the coefficient matrix A is lower triangular. Thus for these formulas the internal approximations Y_i can be computed in the order $i = 1, 2, \ldots, r$ by solving r systems of N nonlinear equations sequentially. The maximum attainable order of an r-stage, semi-implicit RKF is $r + 1$, although for r even and less than 10, the maximum attainable order is r. Semi-implicit RKFs may have high order and may be A-stable. For example, sixth-order, six-stage A-stable formulas are known [Cooper and Sayfy 1979].

Semi-implicit RKFs in which all the nonzero diagonal elements of the coefficient matrix A are equal to some number λ are termed *diagonally implicit* RK (DIRK) [Alexander 1977]. Only one iteration matrix is necessary for DIRK formulas if a Newton iteration scheme is used to solve the nonlinear equations for the Y_i. DIRK formulas are actually a subset of singly-implicit RKFs and may be implemented using Butcher's scheme. However, most codes that implement DIRK formulas solve the sets of nonlinear equations sequentially. DIRK formulas require less computational effort per step than singly-implicit RKFs implemented using Butcher's technique, but singly-implicit RKFs may achieve higher order for a given number of stages than DIRK formulas.

Monoimplicit RKFs [Cash 1975, 1982b], (called *implicit endpoint quadrature rules* by Van Bokhoven [1980]), are a generalization of implicit RKFs. If we consider r-stage formulas of the type

$$y_{n+1} - y_n = h \sum_{j=1}^{r} b_j f(x_n + c_j h, Y_j),$$

$$Y_j = \delta_j y_{n+1} + (1 - \delta_j) y_n + h \sum_{i=1}^{r} a_{ij} f(x_n + c_i h, Y_i),$$

where b_j, δ_j, c_j, and a_{ij}, $i = 1, 2, \ldots, r$ and $j = 1, 2, \ldots, r$ are constants, then if $\delta_j = 0$ for all j, the formula is an implicit RKF. If, however, we have $a_{ij} = 0, j \le i$, then the resulting formula is a monoimplicit RKF that is only implicit in a single unknown, y_{n+1}. The nonlinear system to be solved at each step is of size N, N being the number of equations in the differential system, but the Newton iteration matrix is now a polynomial of degree r in J, an approximation to the Jacobian.

Singhal [1980] considered *singly-implicit, monoimplicit* RKFs, formulas in which the Newton iteration matrix factorizes exactly as the power of a single matrix, but found that such formulas were not in general competitive with LMFs. Cash [1982b] therefore considers an approach used by Skeel and Kong [1977], and derives low-order formulas for which the coefficient matrix of the Newton scheme nearly factorizes as a power of a single matrix. Cash also presents some numerical results which suggest that such formulas may be competitive with LMFs at low error tolerances.

For problems in which the explicit Jacobian is easy to obtain and is not much more expensive than the evaluation of the function f, the following modified Rosenbrock formulas (also called *generalized RKFs* and *ROW methods*) may be suitable:

$$y_{n+1} = y_n + h \sum_{i=1}^{r} m_i k_i,$$

$$Ek_i = f(x_n + A_i h, y_n + \sum_{j=1}^{i-1} a_{i,j} k_j) + B_i h f_x(x_n, y_n) + \sum_{j=1}^{i-1} c_{i,j} k_j, \qquad i = 1, 2, \ldots, r,$$

$$E = I - h\gamma f_y(x_n, y_n).$$

ROW methods are a generalization, due to Wolfbrandt [1977], of a class of formulas proposed by Rosenbrock [1963]. All these formulas are linearly implicit. A more computationally efficient formulation of ROW methods was found by Kaps and Wanner [1981]. An r-stage ROW method has a maximum order of $r + 1$ and requires the evaluation of the Jacobian and an *LU* decomposition at each step. It is, of course, possible to use an approximation to the Jacobian, but since J appears in the formulas itself, the use of an approximate Jacobian may affect the order of the method.

Kaps and Rentrop [1979] have implemented two three-stage, fourth-order ROW methods. One of the codes is based on a formula that is $A(89.3°)$-stable, while the other is based on an A-stable formula. Because the formulas are low order, Kaps and Rentrop are able to approximate the Jacobian by numerical differencing at each step. The results of using the two codes to solve the 25 test problems in Enright et al. [1975] are presented and compared with the performance of a BDF code. Kaps and Rentrop conclude that their codes were reliable and efficient in solving problems for which the number of equations is small and for which low accuracy was required.

Shampine [1982c] discusses various aspects of designing a code based on Rosenbrock formulas. He presents details of a code, DEGRK, based on a pair of A-stable formulas of orders 3 and 4. Shampine's code does not require the differential system in autonomous form, as the codes of Kaps and Rentrop [1979] do, but the user must explicitly supply not only the Jacobian but also the derivatives $\partial f/\partial x$. Results of numerical testing are presented to show that for the restricted class of small problems, DEGRK is efficient. The code is type insensitive and uses a RK–Fehlberg pair to solve the equations whenever the equations are locally nonstiff.

Since an *LU* decomposition is required at each step, Rosenbrock methods are not likely to be suitable for medium-to-large systems. Steihaug and Wolfbrandt [1979] have considered a subset of ROW methods in which the Jacobian matrix in a ROW method can be replaced by a general matrix A without affecting the order properties of the formula. Such formulas are called *W methods.* As the matrix A is no longer assumed to be the exact Jacobian, W methods must satisfy a large number of order requirements. Currently, only low-order examples are known.

Day and Murthy [1982] derive a second-order, A-stable W method that requires two function evaluations and the solution of five linear systems per step, and a third-order, A-stable formula that requires three function evaluations and seven linear systems per step. Both formulas have an embedded lower order formula that can be used as an error estimator. Day and Murthy also derive a third-order, A-stable formula with a lower order embedded error estimator that requires two function evaluations and six linear systems per step. This formula is like a ROW method in that it requires the exact evaluation of the Jacobian at some point, but is also like a W method in that it can use this Jacobian approximation in subsequent steps.

Since the stability properties of the formulas depend on how the matrix A approximates the Jacobian, any implementation of these W methods will have to monitor the outdatedness of the Jacobian approximation. This may prove an obstacle to an efficient implementation. For instance, the nonlinear stability of linearly implicit one-step methods such as ROW and W methods over a large class of nonlinear problems was investigated by Hairer et al. [1982]. They showed that even if the methods were A-stable, extra conditions for the parameters were needed for the methods to produce contractive numerical solutions for sufficiently small h. This restriction on step size did not, however, depend on the stiffness of the differential equation, but on the second derivatives of the true solution of the differential equation.

5.4 Extrapolation Methods

Dahlquist [1963] discussed the use of extrapolation in solving stiff equations, indicating that extrapolation could be applied using the trapezoidal rule:

$$y_{n+1} = y_n + \frac{h}{2}\,(f(x_{n+1}, y_{n+1}) + f(x_n, y_n)).$$

Although the trapezoidal rule is A-stable, if these extrapolated values are used in subsequent computation, the resultant method is no longer A-stable.

Lindberg [1971, 1972, 1973] has studied various questions connected with extrapolation using the trapezoidal rule and the following implicit midpoint rule:

$$y_{n+1} = y_n + hf\left(x_{n+\frac{1}{2}}, \frac{(y_n + y_{n+1})}{2}\right).$$

Both these methods have global error expansions in even powers of h only and thus seem well suited for repeated extrapolation. However, numerical oscillations are obtained when solving stiff equations using extrapolation, and the amplitude of these oscillations may sometimes be large. Lindberg suggested the use of smoothing and extrapolation together to get rid of these oscillations. He developed a code IMPEX2; details are given in Lindberg [1973]. The numerical testing of Enright et al. [1975] and Enright and Hull [1977b] indicates that a version of IMPEX2 was efficient at large tolerances but had difficulty solving problems which have strong nonlinear coupling.

Liniger and Odeh [1972] discuss methods based on *averaging* of multistep methods, a technique similar to extrapolation. While in extrapolation you solve an ODE using one formula and various step sizes, in averaging solutions are obtained using several low-order multistep formulas. These solutions are then combined to obtain higher order solutions. For certain sets of formulas, the averages are A-stable. Algorithms of up to order 6 that are A-stable are presented by Liniger [1976].

Bader and Deuflhard [1983] have developed an extrapolation method based on a linearly implicit analog of the explicit midpoint rule. For the method to achieve order $2m$, m subintegrations must be performed. Each subintegration starts with a step taken using a linearly implicit analog of the backward Euler formula, and then m steps are taken using the linearly implicit midpoint rule. Each subintegration requires an LU factorization of the iteration matrix. To achieve fourth order requires two subintegrations, two LU decompositions, five function evaluations, and the solution of eight linear systems involving the iteration matrix. Achieving sixth order requires three subintegrations, three LU decompositions, ten function evaluations, and the solution of 15 linear systems.

Bader and Deuflhard [1983] compared the performance of their extrapolation code METAN1 with the BDF-based code GEAR on 25 problems at three tolerances. They found that both codes took approximately the same amount of time to solve most problems, but that GEAR took fewer function evaluations and required far fewer LU decompositions. (At the tolerance of 10^{-6} METAN1 required more than twice as many LU decompositions as GEAR.) Thus, the fact that extrapolation codes require a number of LU decompositions per step means that such codes are not well suited to large problems unless they can take advantage of the structure of the Jacobian of the problem. Deuflhard et al. [1980] have also incorporated sparse matrix techniques into METAN1 to produce a code METAS1, which is used in a package for the simulation of large differential systems arising in chemical kinetics.

Shampine [1982b] has modified METAN1 to produce a type-insensitive code. If the code diagnoses the problem to be locally nonstiff, a trivial Jacobian approximation of $J = 0$ is used. The code therefore avoids always having to solve a number of linear systems at each step.

5.5 Second-Derivative Methods

Since no LMF of order greater than two can be A-stable, considerable interest has been shown in classes of formulas that do not suffer from this limitation. Enright [1972] derived a family of k-step *second-derivative multistep formulas* (SDFs). The formulas in this family are A-stable for order less than 5 and $A(\alpha)$-stable for order 5–9 inclusive. These formulas are also more accurate than the BDFs of corresponding orders. However, in order to apply these formulas, an approximation to the second derivative must be computed at each step. If the differential system is written in the autonomous form

$$y'(x) = f(y(x)),$$

then we have the relation $y''(x) = J(y(x))y'(x)$. All current codes based on Enright's formulas compute the second-derivative terms in this manner. They require that the equations be in autonomous form and that the user supply a routine for evaluating the Jacobian and a routine for evaluating the function f.

Enright's formulas are k-step LMF of the form

$$y_{n+1} = y_n + h \sum_{j=0}^{k} \beta_j y'_{n+1-j} + h^2 \gamma_0 y''_{n+1}. \tag{5.6}$$

Like the Adams formulas, these formulas have the property that all extraneous roots at the origin are zero, and like the BDFs, all roots at infinity are zero. Enright chose the parameters β_j, $j = 0, 1, \ldots, k$ and γ_0 to achieve maximum order. Thus the formulas (5.6) that are k step are of order $k + 2$. The formulas up to order 4 are A-stable.

In order to solve stiff systems using Enright's formulas, the resulting systems of nonlinear equations have a Newton iteration matrix of the form

$$W_{n+1} = I - h\beta_0 J - h^2 \gamma_0 J^2, \tag{5.7}$$

where J is some approximation to the Jacobian at the current point. In order to form W_{n+1} a matrix multiplication is required. Thus forming the Newton iteration matrix in its LU factored form requires $O(4N^3/3)$ operations compared to $O(N^3/3)$ for LMF.

Implementations of Enright's formulas perform a lot more linear algebra per step than implementations of the BDFs. In addition to the matrix multiplications discussed above, Enright's formulas entail other costs that do not appear in codes based on the BDFs, such as the matrix–vector multiplications required to form the second-derivative terms. On the other hand, since Enright's formulas are of up to order 9 and have much smaller truncation error coefficients than the BDFs, they tend to require far fewer steps than codes based on the BDFs. Also, because of their better stability properties, SDF codes do not experience the difficulties that BDF codes have in solving stiff problems that contain eigenvalues with large imaginary parts. Test results indicate that for small-to-medium problems, SDF codes are competitive with BDF codes. However, for large problems the cost of forming W_{n+1} as in (5.7) makes them relatively much more expensive. Also the requirement that the user supply an exact Jacobian makes SDF codes less suitable as general purpose codes.

In his thesis, Enright [1972] presented a code SDBASIC for solving stiff systems. This code is a variable-step, variable-order implementation. It uses a one-step, two half-step error estimate and restricts step size changes to halving and doubling. Since then more efficient implementations, which use a predictor–corrector approach and estimate the error as a difference between predicted and corrected values, have been developed. These include the codes SECDER and SDSTEP [Addison 1979; Sacks-Davis 1980].

Variations of Enright's original formulas have been proposed. Since matrix squaring when forming W_{n+1} involves $O(N^3)$ operations, and may reduce sparsity and increase bandwidth, Enright [1974] considered ways of avoiding matrix squaring. He proposed a class of k-step SDFs of the form (5.6) that are of order $k + 1$ and for which $\beta_0^2 + 4\gamma_0 = 0$. For these formulas, the iteration matrix W_{n+1} of (5.7) can be expressed in the form

$$W_{n+1} = \left(I - \frac{h\beta_0 J}{2}\right)^2.$$

Thus the need to form a matrix multiplication is avoided. However, the solution of a system of equations $W_{n+1}x = b$ will now require two forward and backward substitutions rather than one. The resulting set of SDFs have poorer $A(\alpha)$-stability properties than Enright's original formulas.

In another approach, Sacks-Davis and Shampine [1981] show that SDFs can be used to solve both stiff and nonstiff problems in a type-insensitive code. Since Enright's formulas have small truncation error coefficients, they are suitable for solving nonstiff problems as well as stiff systems. The availability of the exact Jacobian leads to a natural test for stiffness based on the norm of $h_{n+1}J$.

5.6 Blended Linear Multistep Methods

Lambert and Sigurdsson [1972] present multistep formulas whose coefficients depend on both the step size and the Jacobian of the equations being solved. The accuracy of the Jacobian does not affect the order of the method. Skeel and Kong [1977] have proposed a subclass of these formulas as suitable for a general-purpose ODE solver. One of the aims of Skeel and Kong was to design a code that is efficient for stiff as well as nonstiff problems.

Skeel and Kong's approach is to take a linear multistep formula involving higher derivatives and convert it into a variable matrix coefficient LMF, in such a way that the order of accuracy and the stability properties are unaffected. For example, for linear problems, the $(k+2)$-order SDF of Enright [1972] may be expressed as a blend of the $(k+2)$-order Adams–Moulton formula and the $(k+1)$-order BDF,

$$\{\text{Enright}^{k+2}\} \rightarrow \{\text{AMF}^{k+2}\} + \gamma_k^* hJ\{\text{BDF}^{k+1}\},$$

where J is the Jacobian and γ_k^* is a constant. For example, Enright's third-order formula is

$$y_{n+1} - y_n - \tfrac{2}{3}hf_{n+1} - \tfrac{1}{3}hf_n + \tfrac{1}{6}h^2y''_{n+1} = 0,$$

the third-order Adams–Moulton formula is

$$y_{n+1} - y_n - \tfrac{5}{12}hf_{n+1} - \tfrac{2}{3}hf_n + \tfrac{1}{12}hf_{n-1} = 0,$$

while the second-order BDF is

$$\tfrac{3}{2}y_{n+1} - 2y_n + \tfrac{1}{2}y_{n-1} - hf_{n+1} = 0,$$

and the blended formula that has the same absolute stability properties as Enright's third-order formula is

$$y_{n+1} - y_n - \tfrac{5}{12}hf_{n+1} - \tfrac{2}{3}hf_n + \tfrac{1}{12}hf_{n-1} - \tfrac{1}{6}hJ(\tfrac{3}{2}y_{n+1} - 2y_n + \tfrac{1}{2}y_{n-1} - hf_{n+1}) = 0.$$

It is easy to verify that Enright's formula and this blended formula are identical for the linear problem $y' = Jy + C$. By adjusting the coefficient γ_k^* it is possible to improve the absolute stability region without sacrificing accuracy. In this case the corresponding SDF is $k+1$ step, $(k+2)$ order. As remarked earlier, the SDFs of Enright [1972] have Newton iteration matrices that are quadratic polynomials in the Jacobian, and cannot be reduced to a product of real linear factors. Skeel and Kong have implemented their formulas in a code called BLENDED DIFSUB, and avoid this difficulty by choosing to use an approximate Newton iteration matrix, which can be expressed as a perfect square in a real linear factor. It is not clear what effect this approximation has on the rate of convergence of the iterations.

Compared to the LMFs, the blended formulas require an extra matrix–vector multiplication per iteration, and, when stiff equations are being solved, the exact Newton iteration matrix is quadratic in the Jacobian. However, the family of formulas presented by Skeel and Kong [1977] are A-stable up to order 4 and $A(\alpha)$-stable up to order 12. Skeel and Kong compared the performances of the BDF code DIFSUB and a modified form of DIFSUB that implements blended LMFs, and found that the blended code was better than the BDF code in solving nonstiff equations and much better in solving stiff oscillatory problems. For other problems, it seems the performance of blended code is similar to that of DIFSUB. However, as we have noted, the blended LMFs do incur some extra computational expense at each iteration.

5.7 Composite and Cyclic Multistep Formulas

One way of overcoming the limitations of LMFs with respect to the achievable order of A-stable formulas is to use a number of LMFs together to simultaneously compute the solution at a number of points. Such a scheme is known as a composite LMF.

Rosser [1967] and Shampine and Watts [1969] propose the following type of formulas, which compute a block of r new values of y at each stage by using r LMFs of the following type:

$$\sum_{j=1}^{r} a_{ij} y_{n+j} = e_i y_n + h d_i f_n + \sum_{j=1}^{r} b_{ij} f_{n+j}, \qquad i = 1, 2, \ldots, r. \tag{5.8}$$

The r LMFs are used together to find $y_{n+1}, y_{n+2}, \ldots, y_{n+r}$ simultaneously. The main advantages of such methods are the possibility of high-order, A-stable methods. In the composite scheme (5.8), the computation of a new block of values requires past values from only one point, and therefore changes in the step size can be made easily after each block has been computed. Schemes like (5.8) are also referred to as *block-implicit one-step* methods and have been discussed by Watts and Shampine [1971], Bickart and Picel [1973], and Williams and DeHoog [1974]. Sloate and Bickart [1973] present a fourth-order A-stable method using two three-step LMFs, and they also present details of an implementation. Their formulas require values at two past points.

In common with implicit Runge–Kutta schemes, these schemes generally have one main drawback. If a composite LMF calculates r new values at r points, it generally requires the solution of a system of rN nonlinear equations, N being the number of equations in the differential system. The methods proposed for reducing the linear algebra costs associated with implicit RKFs can also be applied to composite LMFs. In fact, the technique of Bickart [1977] was originally applied to composite multistep formulas.

We saw in Section 5.3 that a semi-implicit RKF can be implemented in such a way that r sets of N implicit equations need to be solved sequentially in each step. Semi-implicit RKFs have composite LMF analogs in the cyclic composite LMFs, also known as *cyclic LMFs*. These methods use r LMFs in a set order to calculate the solution sequentially at r points. In this way, r systems of N nonlinear equations are solved successively to generate the solution at r points. If all the formulas in the cyclic formula have the same Newton iteration matrix, the cost per cycle is the same as that of using a single LMF for computing the solution at all r points.

Donelson and Hansen [1971] derived stable cyclic LMFs from k-step LMFs of order $2k - 1$ for $k = 2$, 3, and 4. In so doing they showed that a cyclic LMF could be stable even though its constituent LMFs were not. Furthermore, their methods were shown to be globally convergent to order $2k$. Thus they also demonstrated that a cyclic method could have higher order than the order of its constituent formulas. Albrecht [1978] has given general conditions for determining the order of a wide class of methods, and shows how his results apply to composite LMFs and in particular to cyclic methods. Mihelcic [1977] derives $A(\alpha)$-stable cyclic LMFs that have order greater than their step number.

Tendler et al. [1978] derive $A(\alpha)$-stable cyclic LMFs that are kth order, k step for $k = 3, 4, \ldots, 7$ and display better $A(\alpha)$-stability properties, order for order, than the BDFs. The third- and fourth-order cyclic formulas use three LMFs cyclically, while the higher order cyclic formulas use four. By incorporating the first- and second-order BDFs, Tendler et al. [1978] have implemented their cyclic formulas in a code called STINT. STINT is efficient for stiff problems where the Jacobian of the system of equations has some eigenvalues with negative real part and close to the imaginary axis, but otherwise it is less efficient than a BDF-based code such as GEAR or LSODE.

Tischer and Sacks-Davis [1983] focus their attention on two-stage cyclic LMFs (i.e., cyclic LMFs that only use two LMFs cyclically), and show that a two-stage cyclic LMF has the same stability properties as a SDF. By constructing two-stage cyclic LMFs to have the same stability properties as a family of SDFs, a family of cyclic LMFs is derived that is A-stable up to order 4 and $A(\alpha)$-stable up to order 8. These formulas have better $A(\alpha)$-stability properties, order for order, than the formulas used in STINT and have certain other advantages. In particular, these formulas use fewer stages, and all the stages in the cycle have the same leading coefficient and therefore the same Newton iteration matrix. These formulas also have better $A(\alpha)$-stability properties, order for order, than the BDFs, but the local truncation error coefficients of the stages of the cyclic formulas are larger than the local truncation error coefficient of the BDF of the corresponding order. Tischer and Gupta [1983] discuss several more sets of cyclic LMFs, and present results of a performance evaluation showing that some sets of cyclic LMFs are quite efficient for solving stiff equations.

We have considered a number of alternatives to implicit LMFs for solving stiff equations, but only cyclic LMFs can be implemented in a way that requires the same amount of linear algebra per step as implicit LMFs. Since there are cyclic LMFs that have better $A(\alpha)$-stability properties, order by order, than the BDFs, it seems that the use of cyclic LMFs for solving stiff systems of equations may lead to codes that are more efficient than BDF codes.

6. SPECIAL PROBLEMS

In practice, differential equations often have additional features that ODE solvers should be able to deal with. As discussed by Gear [1981, 1982] and Thomson and Tuttle [1982], one or more of the following features occur frequently in problems being solved by practitioners.

6.1 Implicit Equations

Suppose that the differential equations to be solved are of the form

$$F(x, y, y') = 0.$$

If $\partial F/\partial y$ is nonsingular, the equations are called implicit ODEs and the methods we have discussed so far do not directly apply to them. They can arise, for example, when the method of lines based on collocation is applied to the solution of PDEs.

Many implicit ODEs can be written in the form

$$A(x, y)y' = g(x, y),$$

where $A(x, y)$ is a $N \times N$ matrix. An obvious approach to solving such implicit equations would be to invert the matrix A and convert the equations to the conventional form (1.1), but this is neither necessary nor desirable. Hindmarsh [1981, 1983] discusses LSODI, a version of LSODE, for solving implicit equations directly.

6.2 Differential-Algebraic Equations

If the set of equations to be solved consists of both algebraic and differential equations, the equations are called *differential–algebraic* equations (DAEs). The equations may be written as

$$F_1(x, y, z, y') = 0, \qquad y(x_0) = y_0,$$
$$F_2(x, y, z) = 0,$$

where F_1 is a set of N equations and F_2 of M equations ($N \geq M$). The implicit equations are obviously a subset of the differential–algebraic equations.

An early approach to solving these equations was suggested by Gear [1971d]. Gear suggested using ODE methods for solving the above equations. For example, by using the backward Euler method, y' may be approximated by a backward difference of y and then the following nonlinear equations need to be solved at each step:

$$F_1\left(x_{n+1}, y_{n+1}, z_{n+1}, \frac{(y_{n+1} - y_n)}{(x_{n+1} - x_n)}\right) = 0,$$

$$F_2(x_{n+1}, y_{n+1}, z_{n+1}) = 0.$$

Such techniques seem to work quite well for many DAEs, but only recently it has been discovered that such methods may not always converge, especially if the step size is not fixed [Petzold 1982; Gear and Petzold 1984]. A code for solving DAEs is described by Petzold [1983b].

6.3 Discontinuities

One or more derivatives of the solution of a set of differential equations sometimes have discontinuities because of the nature of the model. For example, the model may involve some valves that close or open at some critical value of a variable.

If an ODE solver is not designed to locate a discontinuity, the code is likely to become very inefficient near a discontinuity. A discontinuity may appear without warning. However, the user often knows when a discontinuity will occur, in which case, a code should have facilities to enable the user to describe the discontinuity in the form of a function that becomes zero at the discontinuity. Root-finding techniques can then be used to locate the discontinuity, and appropriate action may then be taken. This may include a special technique for passing the discontinuity and restarting techniques if the code is based on multistep methods. A version of LSODE, called LSODAR, has facilities for root finding to help locate a discontinuity [Hindmarsh 1983].

If, however, it is desired that a code should be able to detect a discontinuity that appears without warning, then the code must make additional tests at each step at which the error test fails. Further details about dealing with discontinuities are given by Carver [1977], Ellison [1980], and Gear and Østerby [1984].

6.4 High-Frequency Solution

Differential equations which have highly oscillatory solutions that do not damp out cannot be efficiently solved using the methods discussed so far. The presence of high-frequency solutions in ODEs is sometimes erroneously referred to as stiffness. This causes unnecessary confusion because obtaining solutions to the two different problem types requires quite different approaches.

Several types of methods have been suggested for solving oscillatory problems, depending on the computational objectives. One class of methods attempts to accurately compute future oscillations. Such methods are needed for applications like computing satellite orbits. Another class of methods attempts to predict the long-term behavior of the solution to an oscillatory problem without following the oscillations closely. One such approach involves calculating a curve that is the *envelope* of the solution. An approximate differential equation is derived which defines a *quasi-envelope* that is smooth. The approximate differential equation is then solved using step sizes very much larger than the period of the oscillations [Gallivan 1983; Gear and Gallivan 1982; Petzold 1978; Scheid 1983].

7. CONCLUSIONS

Although solving ODEs is an old topic and some of the software for solving ODEs uses formulas developed around the turn of the century, significant advances have been made during the last 15 years or so. These advances have been most noticeable in the area of solving stiff equations, but significant advances have also been made in the production of more robust, user-friendly, and efficient software for solving nonstiff equations.

The Adams formulas and RK–Fehlberg formulas are the basis for most software for solving nonstiff equations. Although the BDFs continue to be the basis of most widely used codes in solving stiff equations, a number of promising alternatives based on other LMFs, singly-implicit RKFs, blended LMFs, and cyclic LMFs have been investigated. These new methods have better stability properties and higher orders than the BDFs with no apparent increase in overhead costs. These methods, however, have not yet been implemented in quality production codes.

Modern codes for solving ODEs automatically vary the step size and order, estimate the local error, and provide facilities to compute the solution at intermediate points via interpolation. In addition, stiff codes provide options for the efficient solution of linear equations, such as facilities for handling sparse or banded matrices. However, recent techniques, such as those based on the iterative solution of linear equations, are not yet generally available in ODE solvers. Other advances have been made in the automatic detection of stiffness, the development of type-insensitive codes, the estimation of global errors, and the solution of special problems such as systems of differential–algebraic equations. Again, many of these advances have not been reflected in standard ODE software.

ACKNOWLEDGMENTS

The authors are particularly grateful to Dr. L. F. Shampine and Professor G. G. Dahlquist for their encouraging comments on an early version of this paper. The authors would also like to acknowledge the helpful contributions of Professor R. D. Skeel, Professor K. R. Jackson, and the three referees.

REFERENCES

ADDISON, C. A. 1979. Implementing a stiff method based upon the second derivative formulas. Tech. Rep. 130, Dept. of Computer Science, Univ. of Toronto, Ont., Canada.

AIKEN, R. C. 1982. Stiff review 1974–1982: 1. Applications. In *Proceedings of the International Conference on Stiff Computation* (Park City, Utah, Apr.). To be published in *Stiff Computation*, R. C. Aiken, Ed. Oxford Univ. Press, London and New York, 1985.

ALBRECHT, P. 1978. On the order of composite multistep methods for ordinary differential equations. *Numer. Math. 29*, 381–396.

ALEXANDER, R. 1977. Diagonally implicit Runge–Kutta methods for stiff O.D.E.'s. *SINUM 14*, 1006–1021.

ANDRUS, J. F. 1979. Numerical solution of solution of ordinary differential equations separated into subsystems. *SIAM J. Numer. Anal. 16*, 605–611.

BADER, G., AND DEUFLHARD, P. 1983. A semi-implicit mid-point rule for stiff differential systems of ordinary differential equations. *Numer. Math. 41*, 373–398.

BARTON, D. 1980. On Taylor series and stiff equations. *ACM Trans. Math. Softw. 6*, 3 (Sept.), 280–294.

BARTON, D., WILLERS, I. M., AND ZAHAR, R. V. M. 1971. Taylor series methods for ordinary differential equations—An evaluation. In *Mathematical Software*, J. R. Rice, Ed. Academic Press, New York, pp. 369–390.

BEDET, R. A., ENRIGHT, W. H., AND HULL, T. E. 1975. STIFF DETEST: A program for comparing numerical methods for stiff ordinary differential equations. Tech. Rep. 81, Dept. of Computer Science, Univ. of Toronto, Ont., Canada.

BETTIS, D. G. 1978. Efficient embedded Runge-Kutta methods. In *Numerical Treatment of Differential Equations*, Lecture Notes in Mathematics, vol. 631. Springer-Verlag, Berlin and New York, pp. 9–18.

BICKART, T. A. 1977. An efficient solution process for implicit Runge-Kutta methods. *SINUM 14*, 1022–1027.

BICKART, T. A., AND PICEL, Z. 1973. High order stiffly stable composite multistep methods for numerical integration of stiff differential equations. *BIT 13*, 272–286.

BJÖRCK, A. 1983. A block QR algorithm for partitioning stiff differential equations. *BIT 23*, 329–345.

BJUREL, G., DAHLQUIST, G. G., LINDBERG, B., LINDE, S., AND ODEN, L. 1970. Survey of stiff ordinary differential equations. Rep. NA 70.11, Dept. of Information Processing, Royal Institute of Technology, Stockholm, Sweden.

BRAYTON, R. K., GUSTAVSON, F. G., AND HATCHEL, G. D. 1972. A new efficient algorithm for solving differential-algebraic systems using implicit backward differentiation formulas. *Proc. IEEE 60*, 98–108.

BROWN, P. N., AND HINDMARSH, A. C. 1984. Matrix free methods for stiff ODEs. Rep. UCRL-90770, Lawrence Livermore National Laboratory, Lawrence, Calif., May 1984.

BULIRSCH, R., AND STOER, J. 1966. Numerical treatment of ordinary differential equations by extrapolation methods. *Numer. Math. 8*, 1–13.

BURRAGE, K. 1978. A special family of Runge-Kutta methods for solving stiff differential equations. *BIT 28*, 22–41.

BURRAGE, K., BUTCHER, J. C., AND CHIPMAN, F. 1980. An implementation of singly-implicit Runge-Kutta methods. *BIT 20*, 326–340.

BUTCHER, J. C. 1964a. Implicit Runge-Kutta processes. *Math. Comput. 18*, 50–64.

BUTCHER, J. C. 1964b. Integration processes based on Radau quadrature formulas. *Math. Comput. 18*, 233–243.

BUTCHER, J. C. 1976. On the implementation of implicit Runge-Kutta methods. *BIT 16*, 237–240.

BUTCHER, J. C. 1979. A transformed implicit Runge-Kutta method. *J. ACM 26*, 4 (Oct.), 731–738.

BYRNE, G. D., AND HINDMARSH, A. C. 1975. A polyalgorithm for the numerical solution of ordinary differential equations. *ACM Trans. Math. Softw. 1*, 1 (Mar.), 71–96.

BYRNE, G. D., HINDMARSH, A. C., JACKSON, K. R., AND BROWN, H. G. 1977. A comparison of two ODE codes: GEAR and EPISODE. *Comput. Chem. Eng. 1*, 133–147.

CARVER, M. B. 1976. The choice of algorithms in automated method of lines solution of partial differential equations. In *Numerical Method for Differential Systems*, L. Lapidus and W. E. Schiesser, Eds. Academic Press, Orlando, Fla., pp. 243–265.

CARVER, M. B. 1977. Efficient handling of discontinuities and time delays in ordinary differential equation systems. In *Proceedings of the International Conference SIMULATION 77* (Montreaux, Switzerland), pp. 153–158.

CARVER, M. B. 1978. Efficient implementation over discontinuities in ordinary differential equation simulations. *Math. Comput. Simulat. 20*, 190–196.

CARVER, M. B., AND BAUDOUIN, A. P. 1976. Solution of reactor kinetics problems using sparse matrix techniques in an ODE integrator for stiff equations. Atomic Energy of Canada Limited, Rep. AECL-5177, Jan. 1976.

CARVER, M. B., AND MACEWEN, S. R. 1981. On the use of sparse matrix approximation to the Jacobian in integrating large sets of ordinary differential equations. *SIAM J. Sci. Stat. Comput. 2*, 51–64.

CASH, J. R. 1975. A class of implicit Runge-Kutta methods for the numerical integration of stiff ordinary differential equations. *J. ACM 22*, 4 (Oct.), 504–511.

CASH, J. R. 1982a. A survey of Runge-Kutta methods for the numerical integration of stiff differential systems. In *Proceedings of the International Conference on Stiff Computation* (Park City, Utah, Apr.). To be published in *Stiff Computation*, R. C. Aiken, Ed. Oxford Univ. Press, New York and London, 1985.

CASH, J. R. 1982b. Mono-implicit Runge-Kutta formulae for the numerical solution of stiff differential systems. *IMA J. Numer. Anal. 2*, 289–301.

CHAN, T. F., AND JACKSON, K. R. 1983. Nonlinearly preconditioned Krylov subspace methods for discrete Newton algorithms. *SIAM J. Sci. Stat. Comput. 5*, 533–542.

CHAN, T. F., AND JACKSON, K. R. 1984. The use of iterative linear-equation solvers in codes for large systems of stiff IVPs for ODEs. Tech. Rep. TR 170/84, Dept. of Computer Science, Univ. of Toronto, Ont., Canada.

CHANG, J. S., HINDMARSH, A. C., AND MADSEN, N. K. 1974. Simulation of chemical kinetics transport in the stratosphere. In *Stiff Differential Systems*, R. Willoughby, Ed. Plenum, New York, pp. 51–65.

CHIPMAN, F. H. 1971. *A*-stable Runge–Kutta processes. *BIT 11*, 384–388.

COLEMAN, T. F., GRABOW, B. S., AND MORÉ, J. J. 1984. Software for estimating sparse Jacobian matrices. *ACM Trans. Math. Softw. 10*, 3 (Sept.), 329–345.

COOPER, J. F., AND SAYFY, A. 1979. Semi-explicit, *A*-stable Runge–Kutta methods. *Math. Comput. 33*, 541–556.

CORLISS, G., AND CHANG, Y. F. 1982. Solving ordinary differential equations using Taylor series. *ACM Trans. Math. Softw. 8*, 2 (June), 114–144.

CURTIS, A. R. 1980. The FACSIMILE numerical integrator for stiff ordinary differential problems. In *Computational Techniques for Ordinary Differential Equations*, I. Gladwell and D. K. Sayers, Eds. Academic Press, Orlando, Fla.

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

CURTIS, C. E., AND HIRSCHFELDER, J. O. 1952. Integration of stiff equations. *Proc. Nat. Acad. Sci. U.S. 38*, 235–243.

DAHLQUIST, G. G. 1963. A special stability property for linear multistep methods. *BIT 3*, 27–43.

DAHLQUIST, G. G. 1981. On the control of the global error in stiff initial value problems. In *Numerical Analysis, Dundee 1981*, Springer Lecture Notes in Mathematics, vol. 912, Springer-Verlag, Berlin and New York, pp. 38–49.

DAHLQUIST, G. G., AND BJÖRCK, A. 1974. *Numerical Methods*. Prentice-Hall, Englewood Cliffs, N.J.

DAY, J. D., AND MURTHY, D. N. 1982. Two classes of internally *S*-stable generalized Runge–Kutta processes which remain consistent with an inaccurate Jacobian. *Math. Comput. 39*, 491–509.

DEUFLHARD, P. 1983. Order and stepsize control in extrapolation methods. *Numer. Math. 41*, 399–422.

DEUFLHARD, P., BADER, G., AND NOWAK, U. 1980. LARKIN—A software package for the numerical integration of LARge systems arising in chemical reaction KINetics. SFB 123, Tech. Rep. 100, University of Heidelberg, Heidelberg, West Germany.

DEW, P. M., AND WEST, M. R. 1979. Estimating and controlling the global error in Gear's method. *BIT 19*, 135–137.

DONELSON, J., III, AND HANSEN, E. 1971. Cyclic composite multistep predictor–corrector methods. *SINUM 8*, 137–147.

EHLE, B. L. 1968. High order *A*-stable methods for the numerical solution of systems of differential equations. *BIT 8*, 276–278.

EHLE, B. L. 1972. A comparison of numerical methods for solving certain stiff ordinary differential equations. Rep. 70, Dept. of Mathematics, Univ. of Victoria, B.C., Canada.

ELLISON, D. 1980. Efficient automatic integration of ordinary differential equations with discontinuities. *Math. Comput. Simulat. 23*, 12–20.

ENRIGHT, W. H. 1972. Studies in the numerical solution of stiff ordinary differential equations. Tech. Rep. 46, Dept. of Computer Science, University of Toronto, Ont., Canada.

ENRIGHT, W. H. 1974. Optimal second derivative methods for stiff systems. In *Stiff Differential Systems*, R. Willoughby, Ed. Plenum, New York, pp. 95–109.

ENRIGHT, W. H. 1978. Improving the efficiency of matrix operations in the numerical solution of stiff ordinary differential equations. *ACM Trans. Math. Softw. 4*, 2 (June), 127–136.

ENRIGHT, W. H. 1982. Pitfalls in the comparison of numerical methods for stiff ordinary differential equations. In *Proceedings of the International Conference on Stiff Computation* (Park City, Utah, Apr.). To be published in *Stiff Computation*, R. C. Aiken, Ed. Oxford Univ. Press, New York and London, 1985.

ENRIGHT, W. H., AND HULL, T. E. 1976a. Test results on initial value methods for non-stiff ordinary differential equations. *SINUM 13*, 944–961.

ENRIGHT, W. H., AND HULL, T. E. 1976b. Comparing numerical methods for the solution of stiff systems of ODEs arising in chemistry. In *Numerical Methods for Differential Systems*, L. Lapidus and W. E. Schiesser, Eds. Academic Press, Orlando, Fla., pp. 45–66.

ENRIGHT, W. H., AND KAMEL, M. S. 1979. Partitioning of stiff systems and exploiting the resulting structure. *ACM Trans. Math. Softw. 5*, 4 (Dec.), 374–385.

ENRIGHT, W. H., HULL, T. E., AND LINDBERG, B. 1975. Comparing numerical methods for stiff systems of ODEs. *BIT 15*, 10–48.

FEHLBERG, E. 1969. Klassische Runge–Kutta–Formeln fünfter und siebenter Ordnung mit Schrittweiten-Kontrolle. *Computing 4*, 93–106.

FORSYTHE, G. E., MALCOLM, M. A., AND MOLER, C. B. 1977. *Computer Methods for Mathematical Computation.* Prentice-Hall, Englewood Cliffs, N.J.

FOX, P. A. 1971. Integration of first order systems of differential equations. In *Mathematical Software*, J. Rice, Ed. Academic Press, Orlando, Fla., chap. 9, pp. 477–507.

GAFFNEY, P. W. 1982. A survey of FORTRAN subroutines suitable for solving stiff oscillatory differential equations. Tech. Memo. ORNL/CSD/TM-134, Oak Ridge National Laboratory, Oak Ridge, Tenn.

GAFFNEY, P. W. 1984. A performance evaluation of some FORTRAN subroutines for the solution of stiff oscillatory ordinary differential equations. *ACM Trans. Math. Softw. 10*, 1 (Mar.), 58–72.

GALLIVAN, K. A. 1983. An algorithm for the detection and integration of highly oscillatory ordinary differential equations using a generalized unified modified divided difference representation. Tech. Rep. UIUCDCS-R-83-1121, Dept. of Computer Science, Univ. of Illinois, Urbana–Champaign, June.

GEAR, C. W. 1971a. The automatic integration of ordinary differential equations. *Commun. ACM 14*, 3 (Mar.), 176–190.

GEAR, C. W. 1971b. Algorithm 407—DIFSUB for solution of ordinary differential equations. *Commun. ACM 14*, 3 (Mar.), 185–190.

GEAR, C. W. 1971c. *Numerical Initial Value Problems in Ordinary Differential Equations.* Prentice-Hall, Englewood Cliffs, N.J.

GEAR, C. W. 1971d. Simultaneous numerical solution of differential/algebraic equations. *IEEE Trans. Circuit Theory CT-18*, 89–95.

GEAR, C. W. 1974. Multirate methods for ordinary differential equations. Tech. Rep. UIUCDCS-74-880, Dept. of Computer Science, Univ. of Illinois, Urbana–Champaign, Sept.

GEAR, C. W. 1980. Automatic multirate methods for ordinary differential equations. Tech. Rep. UIUCDCS-R-80-1000, Dept. of Computer Science, Univ. of Illinois, Urbana–Champaign, Jan.

GEAR, C. W. 1981. Numerical solution of ordinary differential equations: Is there anything left to do? *SIAM Rev. 23*, 10–24.

GEAR, C. W. 1982. Stiff software: What do we have and what do we need? In *Proceedings of the International Conference on Stiff Computation* (Park City, Utah, Apr.). To be published in *Stiff Computation*, R. C. Aiken, Ed. Oxford Univ. Press, London and New York, 1985.

GEAR, C. W., AND GALLIVAN, K. A. 1982. Automatic methods for highly oscillatory ordinary differential equations. In *Proceedings of the Biennial Dundee Conference on Numerical Analysis, Dundee*, Lecture Notes in Mathematics, vol. 912. Springer-Verlag, Berlin, and New York, pp. 115–124.

GEAR, C. W., AND ØSTERBY, O. 1984. Solving ordinary differential equations with discontinuities. *ACM Trans. Math. Softw. 10*, 1 (Mar.), 23–44.

GEAR, C. W., AND PETZOLD, L. R. 1984. ODE methods for the solution of differential/algebraic systems. *SIAM J. Numer. Anal. 21*, 716–728.

GEAR, C. W., AND SAAD, Y. 1983. Iterative solution of linear equations in ODE codes. *SIAM J. Sci. Stat. Comput. 4*, 583–601.

GEAR, C. W., AND TU, K. W. 1974. The effect of variable mesh size on the stability of multistep methods. *SINUM 11*, 1025–1043.

GLADWELL, I. 1979. Initial value routines in the NAG Library. *ACM Trans. Math. Softw. 5*, 4 (Dec.), 386–400.

GLADWELL, I., CRAIGIE, J. A. I., AND CROWTHER, C. R. 1979. Testing initial-value problem subroutines as black boxes. Numerical Analysis Rep. 34, Dept. of Mathematics, Univ. of Manchester, U.K.

GOURLAY, A. R., AND WATSON, D. D. 1974. An implementation of Gear's algorithm for CSMP III. In *Stiff Differential Systems*, R. Willoughby, Ed. Plenum, New York, pp. 123–134.

GRAGG, W. B. 1965. On extrapolation algorithms for ordinary initial value problems. *SINUM 2*, 384–403.

GUPTA, G. K. 1976. Some new high-order multistep formulae for solving stiff equations. *Math. Comput. 30*, 417–432.

GUPTA, G. K. 1980. A note about overhead costs in ODE solvers. *ACM Trans. Math. Softw. 6*, 3 (Sept.), 319–326.

GUPTA, G. K. 1982. Description and evaluation of a stiff ODE code DSTIFF. Paper presented at the International Conference on Stiff Computation (Park City, Utah). To appear in *SIAM J. Sci. Stat. Comput.*

GUPTA, G. K., AND WALLACE, C. S. 1975. Some new multistep methods for solving ordinary differential equations. *Math. Comput. 29*, 489–500.

HAIRER, E., BADER, G., AND LUBICH, C. 1982. On the stability of semi-implicit methods for ordinary differential equations. *BIT 22*, 211–232.

HALL, G., ENRIGHT, W. H., HULL, T. E., AND SEDGWICK, A. E. 1973. DETEST: A program for comparing numerical methods for ordinary differential equations. Tech. Rep. 60, Dept. of Computer Science, Univ. of Toronto, Ont., Canada.

HENRICI, P. 1962. *Discrete Variable Methods for Ordinary Differential Equations*. Wiley, New York.

HINDMARSH, A. C. 1974. GEAR: Ordinary differential equation system solver. Rep. UICD-30001, Rev. 3, Univ. of California, Lawrence Livermore Laboratories, Lawrence, Calif.

HINDMARSH, A. C. 1976. Preliminary documentation of GEARBI: Solution of ODE systems with block-iterative treatment of the Jacobian. Rep. UICD-30149, Lawrence Livermore National Laboratories, Lawrence, Calif., Dec.

HINDMARSH, A. C. 1980. LSODE and LSODI, two new initial value ordinary differential equation solvers. *ACM SIGNUM Newsl. 15*, 10–11.

HINDMARSH, A. C. 1981. ODE solvers for use with the method of lines. In *Advances in Computer Methods for Partial Differential Equations—IV*, R. Vichnevetsky and R. S. Stepleman, Eds. IMACS, New Brunswick, N.J., pp. 312–316.

HINDMARSH, A. C. 1983. ODEPACK, A systematized collection of ODE solvers. In *Scientific Computing*, R. S. Stepleman et al., Eds., North-Holland Publ., Amsterdam, pp. 55–64.

HULL, T. E., AND ENRIGHT, W. H. 1974. A structure for programs that solve ordinary differential equations. Tech. Rep. 66, Dept. of Computer Science, Univ. of Toronto, Ont., Canada.

HULL, T. E., ENRIGHT, W. H., FELLEN, B. M., AND SEDGWICK, A. E. 1972. Comparing numerical methods for ordinary differential equations. *SINUM 9*, 603–637.

HULL, T. E., ENRIGHT, W. H., AND JACKSON, K. R. 1976. User's guide for DVERK—A subroutine for solving non-stiff ODE's. Tech. Rep. 100, Dept. of Computer Science, Univ. of Toronto, Ont., Canada.

HUSSELS, H. G. 1973. Schrittenweitensteurung bei der Integration gewöhnlicher Differentialgleichungen mit Extrapolationsverfahren. Master's thesis, Univ. of Cologne, Cologne, West Germany.

JACKSON, K. R., AND SACKS-DAVIS, R. 1980. An alternative implementation of variable step-size multistep formulas for stiff ODEs. *ACM Trans. Math. Softw. 6*, 3 (Sept.), 295–318.

JACKSON, L. W., AND SEDGWICK, A. E. 1977. Vandermonde matrices, coordinate transformations and Adams' methods. Tech. Rep. TR77-5, Dept. of Computing Science, Univ. of Alberta, Alberta, Canada.

KAPS, P., AND RENTROP, P. 1979. Generalized Runge–Kutta methods of order four with stepsize control for stiff ordinary differential equations. *Numer. Math. 33*, 55–68

KAPS, P., AND WANNER, G. 1981. A study of Rosenbrock type methods of high order. *Numer. Math. 38*, 279–298.

KLOPFENSTEIN, R. W. 1971. Numerical differentiation formulas for stiff systems of ordinary differential equations. *RCA Rev. 32*, 447–462.

KONG, A. K. 1977. A search for better linear multistep methods for stiff problems. Tech. Rep. UIUCDCS-R-77-899, Dept. of Computer Science, Univ. of Illinois, Urbana–Champaign, Dec.

KROGH, F. T. 1969. A variable step, variable order multistep method for the numerical solution of ODE's. In *Information Processing 68*, A. J. H. Morrel, Ed. North-Holland Publ., Amsterdam, pp. 194–199.

KROGH, F. T. 1973. On testing a subroutine for the numerical integration of ordinary differential equations. *J. ACM 20*, 4 (Oct.), 545–562.

KROGH, F. T., AND STEWART, K. 1984. Asymptotic ($h \rightarrow \infty$) absolute stability for BDFs applied to stiff differential equations. *ACM Trans. Math. Softw. 10*, 1 (Mar.), 45–57.

LAMBERT, J. D. 1973. *Computational Methods in Ordinary Differential Equations*. Wiley, New York.

LAMBERT, J. D. 1980. Stiffness. In *Computational Techniques for Ordinary Differential Equations*, I. Gladwell and D. K. Sayers, Eds. Academic Press, Orlando, Fla.

LAMBERT, J. D., AND SIGURDSSON, S. T. 1972. Multistep methods with variable matrix coefficients. *SINUM 9*, 715–733.

LINDBERG, B. 1971. On smoothing and extrapolation for the trapezoidal rule. *BIT 11*, 29–52.

LINDBERG, B. 1972. Error estimate and stepsize strategy for the implicit midpoint rule with smoothing and extrapolation. Rep. NA 72.59, Dept. of Information Processing, Royal Institute of Technology, Stockholm, Sweden.

LINDBERG, B. 1973. IMPEX2, a procedure for solution of systems of stiff differential equations. Rep. NA 73.03, Dept. of Information Processing, Royal Institute of Technology, Stockholm, Sweden.

LINIGER, W. 1976. High Order A-stable averaging algorithms for stiff differential equations. In *Numerical Methods for Differential Systems*, L. Lapidus and W. E. Schiesser, eds. Academic Press, Orlando, Fla., pp. 1–23.

LINIGER, W., AND ODEH, F. 1972. A-stable accurate averaging of multistep methods for stiff differential equations. *IBM J. Res. Develop. 16*, 335–348.

MIHELCIC, M. 1977. Fast A-stabile Donelson–Hansensche zyklische Verfahren zur numerischen Integration von 'stiff' Differentialgleichungssystemen. *Angew. Inform. 7*, 299–305.

NORDSIECK, A. 1962. On numerical integration of ordinary differential equations. *Math. Comput. 16*, 22–49.

NØRSETT, S. P. 1974. Semi-explicit Runge–Kutta methods. Mathematics and Computation, Rep. No. 6/74, Mathematics Dept., Univ. of Trondheim, Trondheim, Norway.

ORAILOGLU, A. 1979. A multirate ordinary differential equation integrator. Tech. Rep. UIUCDCS-R-79-959, Dept. of Computer Science, Univ. of Illinois, Urbana–Champaign, Mar.

PETZOLD, L. R. 1978. An efficient numerical method for highly oscillatory ordinary differential equations. *SINUM 18*, 455–479.

PETZOLD, L. R. 1982. Differential/algebraic equations are not ODEs. *SIAM J. Sci. Stat. Comput. 3*, 367–384.

PETZOLD, L. R. 1983a. Automatic selection of methods for solving stiff and non-stiff systems of ordinary differential equations. *SIAM J. Sci. Stat. Comput. 4*, 136–148.

PETZOLD, L. R. 1983b. A description of DASSL, a differential–algebraic system solver. In *Scientific Computing*, R. S. Stepleman et al., Eds. North-Holland Publ., Amsterdam, pp. 65–68.

PIOTROWSKI, P. 1969. Stability, consistence and convergence of variable, K-step methods for numerical integration of systems of ordinary differential. In *Lecture Notes in Mathematics*, vol. 109. Springer-Verlag, Berlin and New York, pp. 221–227.

ROSENBROCK, H. H. 1963. Some implicit processes for the numerical solution of differential equations. *Comput. J. 5*, 329–330.

ROSSER, J. B. 1967. A Runge–Kutta for all seasons. *SIAM Rev. 9*, 417–452.

SACKS-DAVIS, R. 1980. Fixed leading coefficient implementation of SD-formulas for stiff ODEs. *ACM Trans. Math. Softw. 6*, 4 (Dec.), 540–562.

SACKS-DAVIS, R., AND SHAMPINE, L. F. 1981. A type-insensitive ODE code based on second derivative formulas. *Comput. Math. Appl. 7*, 487–495.

SCHEID, R. E., JR. 1983. The accurate numerical solution of highly oscillatory ordinary differential equations. *Math. Comput. 41*, 487–509.

SEAGER, M. K., AND BALSDON, S. 1982. LSODIS, a sparse implicit ODE solver. In *Proceedings of the 10th IMACS World Congress* (Montreal, Aug.). IMACS, New Brunswick, N. J.

SEDGWICK, A. E. 1973. An effective variable order variable stepsize Adams method. Tech. Rep. 53, Dept. of Computer Science, Univ. of Toronto, Ont., Canada.

SHAMPINE, L. F. 1977. Stiffness and nonstiff differential equation solvers II: Detecting stiffness with Runge–Kutta methods. *ACM Trans. Math. Softw. 3*, 1 (Mar.), 44–53.

SHAMPINE, L. F. 1979. Evaluation of implicit formulas for the solution of ODEs. *BIT 19*, 495–502.

SHAMPINE, L. F. 1980a. Lipschitz constants and robust ODE codes. In *Computational Methods in Nonlinear Mechanics*, J. T. Oden, Ed. North-Holland Publ., Amsterdam.

SHAMPINE, L. F. 1980b. Implementation of implicit formulas for the solution of ODEs. *SIAM J. Sci. Stat. Comput. 1*, 119–130.

SHAMPINE, L. F. 1981a. Evaluation of a test set for stiff ODE solvers. *ACM Trans. Math. Softw. 7*, 4 (Dec.), 409–420.

SHAMPINE, L. F. 1981b. Type-insensitive ODE codes based on implicit A-stable methods. *Math. Comput. 36*, 499–510.

SHAMPINE, L. F. 1982a. Type-insensitive codes based on $A(\alpha)$-stable formulas. *Math. Comput. 39*, 109–123.

SHAMPINE, L. F. 1982b. Type-insensitive codes based on extrapolation methods. Tech. Rep. SAND82-1195, Sandia National Laboratories, Albuequerque, N. M., June.

SHAMPINE, L. F. 1982c. Implementation of Rosenbrock methods. *ACM Trans. Math. Softw. 8*, 2 (June), 93–113.

SHAMPINE, L. F. 1982d. What is stiffness? Tech. Rep. SAND82-0782, Sandia Laboratories, Albuquerque, N. M., May.

SHAMPINE, L. F. 1982e. Global error estimation for stiff ODEs. Tech. Rep. SAND82-2517, Sandia Laboratories, Albuquerque, N. M., Dec.

SHAMPINE, L. F. 1983. Measuring stiffness. Tech. Rep. SAND83-1119, Sandia Laboratories, Albuquerque, N. M., June.

SHAMPINE, L. F., AND GEAR, C. W. 1979. A user's view of solving stiff ordinary differential equations. *SIAM Rev. 21*, 1–17.

SHAMPINE, L. F., AND GORDON, M. K. 1975. *Computer Solution of Ordinary Differential Equations.* Freeman, San Francisco, Calif.

SHAMPINE, L. F., AND WATTS, H. A. 1969. Block implicit one-step methods. *Math. Comput. 23*, 731–740.

SHAMPINE, L. F., AND WATTS, H. A. 1976. Global error estimation for ordinary differential equations. *ACM Trans. Math. Softw. 2*, 2 (June), 172–186.

SHAMPINE, L. F., AND WATTS, H. A. 1977. The art of writing a Runge–Kutta code, Part I. In *Mathematical Software III*, J. R. Rice, Ed. Academic Press, Orlando, Fla., pp. 257–276.

SHAMPINE, L. F., AND WATTS, H. A. 1979. The art of writing a Runge–Kutta code, Part II. *Appl. Math. Comput. 5*, 93–121.

SHAMPINE, L. F., AND WATTS, H. A. 1980. DEPAC: Design of a user oriented package of ODE solvers. Tech. Rep. SAND79-2374, Sandia Laboratories, Albuquerque, N. M.

SHAMPINE, L. F., AND WATTS, H. A. 1984. Software for ordinary differential equations. In *Sources and Development of Mathematical Software*, L. R. Cowell, Ed. Prentice-Hall, Englewood Cliffs, N. J., pp. 112–133.

SHAMPINE, L. F., WATTS, H. A., AND DAVENPORT, S. M. 1975. Solving non-stiff ordinary differential equations—The state of the art. *SIAM Rev. 18*, 376–411.

SHAMPINE, L. F., GORDON, M. K., AND WISNIEWSKI, J. A. 1980. Variable order Runge-Kutta Codes. In *Computational Techniques for Ordinary Differential Equations*, I. Gladwell and D. K. Sayers, Eds. Academic Press, Orlando, Fla., pp. 83–102.

SHERMAN, A. H., AND HINDMARSH, A. C. 1980. GEARS: A package for the solution of sparse stiff ordinary differential equations. In *Electrical Power Problems: The Mathematical Challenge*, A. M. Erisman, K. W. Neves, and M. H. Dwarakanath, Eds. SIAM, Philadelphia, Pa., pp. 190–200.

SINCOVEC, R. F., AND MADSEN, N. K. 1975. Software for nonlinear partial differential equations. *ACM Trans. Math. Softw. 1*, 3 (Sept.), 232–260.

SINGHAL, A. 1980. Implicit Runge–Kutta formulae for the numerical integration of ODE's. Ph.D. dissertation, Dept. of Mathematics, Univ. of London, London, England.

SKEEL, R. D., AND KONG, A. K. 1977. Blended linear multistep methods. *ACM Trans. Math. Softw. 3*, 4 (Dec.), 326–345.

SKELBOE, S., AND CHRISTIANSEN, B. 1981. Backward differentiation formulas with extended regions of absolute stability. *BIT 21*, 221–231.

SLOATE, H. M., AND BICKART, T. A. 1973. *A*-Stable composite multistep methods. *J. ACM 20*, 1 (Jan.) 7–26.

STEIHAUG, T., AND WOLFBRANDT, A. 1979. An attempt to avoid exact Jacobian and non-linear equations in the numerical solution of stiff differential equations. *Math. Comput. 33*, 521–534.

STETTER, H. J. 1974. Economical global error estimation. In *Stiff Differential Systems*, R. A. Willoughby, Ed. Plenum, New York, pp. 245–258.

STETTER, H. J. 1978. Global error estimation in ODE-solvers. In *Proceedings of the Biennial Dundee Conference*, Lecture Notes in Mathematics, vol. 630. Springer-Verlag, Berlin and New York, pp. 179–189.

STETTER, H. J. 1979a. Global error estimation in Adams PC-codes. *ACM Trans. Math. Softw. 5*, 4 (Dec.) 415–430.

STETTER, H. J. 1979b. Interpolation and error estimation in Adams PC-codes. *SINUM 16*, 311–322.

TENDLER, J. M., BICKART, T. A., AND PICEL, Z. 1978. A stiffly stable integration process. *ACM Trans. Math. Softw. 4*, 4 (Dec.), 339–368.

THOMSON, S., AND TUTTLE, P. G. 1982. The solution of several representative stiff problems in an industrial environment: The evolution of an ODE solver. In *Proceedings of the International Conference on Stiff Computation* (Park City, Utah, Apr.). To be published in *Stiff Computation*, R. C. Aiken, Ed. Oxford Univ. Press, London and New York, 1985.

TISCHER, P. E., AND GUPTA, G. K. 1983. Some new cyclic linear multistep formulas for stiff systems. Tech. Rep. 40, Dept. of Computer Science, Monash Univ., Clayton, Victoria 3168, Australia.

TISCHER, P. E., AND SACKS-DAVIS, R. 1983. A new class of cyclic multistep formulae for stiff systems. *SIAM J. Sci. Stat. Comput. 4*, 733–747.

VAN BOKHOVEN, W. M. G. 1980. Efficient high order implicit one-step methods for integration of stiff differential systems. *BIT 20*, 34–43.

VARAH, J. M. 1978. Stiffly stable linear multistep methods of extended order. *SINUM 16*, 1234–1246.

VERNER, J. H. 1978. Explicit Runge–Kutta methods with estimates of the local truncation error. *SINUM 15*, 772–790.

VERNER, J. H. 1979. Families of embedded Runge–Kutta methods. *SINUM 5*, 857–875.

WALLACE, C. S., AND GUPTA, G. K. 1973. General linear multistep methods to solve ordinary differential equations. *Aust. Comput. J. 5*, 62–69.

WANNER, G., HAIRER, E., AND NØRSETT, S. P. 1978. Order stars and stability theorems. *BIT 18*, 475–489.

WATKINS, D. S., AND HANSONSMITH, R. W. 1983. The numerical solution of separably stiff systems by precise partitioning. *ACM Trans. Math. Softw. 9*, 3 (Sept.), 293–301.

WATTS, H. A. 1983. Starting step size for an ODE solver. *J. Comput. Appl. Math. 9*, 177–191.

WATTS, H. A., AND SHAMPINE, L. F. 1971. *A*-Stable block-implicit methods. *BIT 12*, 252–266.

WELLS, D. R. 1982. Multirate linear multistep methods for the solution of systems of ordinary differential equations. Tech. Rep. UIUCDCS-R-82-1093, Dept. of Computer Science, Univ. of Illinois, Urbana–Champaign, July.

WILLIAMS, J., AND DEHOOG, F. 1974. A class of *A*-stable advanced multistep methods. *Math. Comput. 28*, 163–178.

WOLFBRANDT, A. 1977. A study of Rosenbrock processes with respect to order conditions and stiff stability. Ph.D. dissertation, Chalmers Univ. of Technology, Gøteborg, Sweden.

ZADUNAISKY, P. E. 1976. On the estimation of errors propagated in the numerical integration of ordinary differential equations. *Numer. Math. 27*, 21–39.

Received November 1983; final revision accepted February 1985.