12 Numerical Methods for Ordinary Differential Equations

The dynamic analysis of a mechanical system requires the solution of the equations of motion. The equations of motion, planar and spatial, are either a set of differential equations or a set of mixed differential and algebraic equations. In general, the equations of motion must be solved numerically, although it might be possible to obtain a closed-form solution to the equations of motion for highly simplified systems.

This chapter provides a brief review of numerical methods for solving ordinary differential equations. It is assumed that the reader has some background in the area of numerical methods.

12.1 INITIAL-VALUE PROBLEMS

A first-order differential equation may be written as

$$\dot{y} = f(y, t) \tag{12.1}$$

This equation has a family of solutions $y(t)$. The choice of an initial value y^0 serves to determine one of the solutions of the family. The initial value y^0 could be defined for any value of t^0, although it is often assumed that a transformation has been made so that $t^0 = 0$. This does not affect the solution or methods used to approximate the solution.

If more than one dependent variable is involved, the problem then is to solve a system of first-order equations; e.g.,

$$\begin{aligned} \dot{y}_1 &= f_1(y_1, y_2, t) \\ \dot{y}_2 &= f_2(y_1, y_2, t) \end{aligned} \tag{12.2}$$

Given the initial values y_1^0 and y_2^0, if the functions f_1 and f_2 are regular; i.e., continuously differentiable, then there will be a unique solution of the system.

Any nth-order ordinary differential equation that can be written with the highest-order derivatives on the left-hand side may be written as a system of n first-order equations by defining $n - 1$ new variables. For example, the second-order equation

$$\ddot{y}_1 = f(y_1, \dot{y}_1, t) \tag{12.3}$$

can be written as the system

$$\begin{aligned} \dot{y}_1 &= y_2 \\ \dot{y}_2 &= f(y_1, y_2, t) \end{aligned} \tag{12.4}$$

In discussing methods of solving initial-value problems, it is convenient to think of a single equation in the form of Eq. 12.1, although the same methods can also be applied to a system of equations. These methods involve a step-by-step process in which a sequence of discrete points $t^0, t^1, t^2, \ldots$ is generated. The discrete points may have either constant or variable spacing $h^i = t^{i+1} - t^i$, where h^i is the step size for any discrete point t^i. At each point t^i, the *solution* $y(t^i)$ is approximated by a *number* y^i.

Since no numerical method is capable of finding $y(t^i)$ exactly, the quantity

$$\varepsilon^i = |y(t^i) - y^i| \tag{12.5}$$

represents the *total error* at $t = t^i$. The total error consists of two components: a *truncation error* and a *round-off error*. The truncation error depends on the nature of the numerical algorithm used in computing y^i. The round-off error is due to the finite word length in a computer. In the rest of the text, when the term *numerical error* is used, we mean truncation error unless stated otherwise.

Although there exist many algorithms for solving initial-value problems, most of them are based on two basic approaches: *Taylor series expansion* and *polynomial approximation*. The objective of Sec. 12.2 and 12.3 is to present general formulas for some of these methods, without proof. Interested readers may refer to numerous textbooks for the derivation of details and for error analysis for these methods. Unless otherwise stated, in order to simplify matters, a *uniform* step size $h^i = h$ is assumed in formulation of the algorithms.

12.2 TAYLOR SERIES ALGORITHMS[2]

Assume that $y(t)$ is the exact solution to the initial-value problem

$$\dot{y} = f(y, t) \tag{12.6}$$

where

$$y^0 = y(t^0) \tag{12.7}$$

If $y(t)$ is expanded in a Taylor series about $t = t^i$ and the expansion is evaluated at $t = t^{i+1}$, it is found that

$$y(t^{i+1}) = y(t^i) + \frac{\dot{y}(t^i)}{1!}h + \frac{\ddot{y}(t^i)}{2!}h^2 + \text{higher-order terms} \tag{12.8}$$

Substituting Eq. 12.6 into Eq. 12.8 yields

$$y(t^{i+1}) = y(t^i) + hf(y^i, t^i) + \frac{h^2}{2!}\dot{f}(y^i, t^i) + \text{higher-order terms} \tag{12.9}$$

The *first-order Taylor algorithm,* also known as the *forward Euler algorithm,* is obtained by eliminating $\dot{f}(y^i, t^i)$ and the higher-order terms in Eq. 12.9:

$$y^{i+1} = y^i + hf(y^i, t^i) \tag{12.10}$$

Truncating Eq. 12.9 of all but the first two terms makes Eq. 12.10 an approximate solution to the initial-value problem of Eqs. 12.6 and 12.7.

The *second-order Taylor* algorithm is obtained by truncating Eq. 12.9 of only the higher-order terms, to obtain

$$y^{i+1} = y^i + hf(y^i, t^i) + \frac{h^2}{2!}\dot{f}(y^i, t^i) \tag{12.11}$$

The term $\dot{f}(y^i, t^i)$ can be expressed as follows:

$$\begin{aligned} \dot{f}(y^i, t^i) &= \frac{\partial f}{\partial y}\dot{y} + \frac{\partial f}{\partial t} \qquad \text{at } t = t^i \\ &= \frac{\partial f}{\partial y}f + \frac{\partial f}{\partial t} \qquad \text{at } t = t^i \end{aligned} \tag{12.12}$$

Similarly, higher-order Taylor algorithms may be derived. However, the higher the order, the more derivative terms of $f(y, t)$ with respect to t must be evaluated. This is a major drawback of the Taylor algorithms. Consequently, they are seldom used except for the lower-order algorithms.

12.2.1 Runge-Kutta Algorithms[2]

The Runge-Kutta algorithms obviate the need for evaluating the partial derivatives and still retain the same order of accuracy as the Taylor algorithms. A *second-order Runge-Kutta algorithm* is stated as

$$y^{i+1} = y^i + hg \tag{12.13}$$

where

$$g = (1 - a)f(y^i, t^i) + af\left(y^i + \frac{h}{2a}f(y^i, t^i), t^i + \frac{h}{2a}\right) \tag{12.14}$$

Note that $a \neq 0$ is a free parameter. Consequently, an entire family of second-order Runge-Kutta algorithms can be derived by assigning different values to a. One common choice is the *modified trapezoidal algorithm,* in which $a = \frac{1}{2}$; then,

$$y^{i+1} = y^i + \frac{h}{2}f(y^i, t^i) + \frac{h}{2}f(y^i + hf(y^i, t^i), t^i + h) \tag{12.15}$$

Another common choice is the *modified Euler-Cauchy algorithm,* in which $a = 1$:

$$y^{i+1} = y^i + hf\left(y^i + \frac{h}{2}f(y^i, t^i), t^i + \frac{h}{2}\right) \tag{12.16}$$

For a larger step size and for greater accuracy, the *fourth-order Runge-Kutta algorithm* is most widely used. This algorithm is given by

$$y^{i+1} = y^i + hg \tag{12.17}$$

where

$$g = \tfrac{1}{6}(f_1 + 2f_2 + 2f_3 + f_4)$$
$$f_1 = f(y^i, t^i)$$
$$f_2 = f\left(y^i + \frac{h}{2}f_1, t^i + \frac{h}{2}\right)$$
$$f_3 = f\left(y^i + \frac{h}{2}f_2, t^i + \frac{h}{2}\right)$$
$$f_4 = f\left(y^i + hf_3, t^i + h\right)$$

Since the algorithm is a fourth-order one, the truncation error will remain relatively small even for a relatively large step size. The major disadvantage of this algorithm is that the function $f(y, t)$ must be evaluated four times at each time step. In addition, the values of the function are not used in any subsequent computations. Hence, this algorithm is not as computationally efficient as some of the multistep algorithms presented in Secs. 12.3 through 12.3.3.

12.2.2 A Subroutine for a Runge-Kutta Algorithm

The fourth-order Runge-Kutta algorithm in the preceding section is represented here in the form of a subroutine that can be embedded in a program to solve a set of ordinary differential equations. This subroutine is written in its simplest form and can be modified easily.[†]

Subroutine RUNGK4. The argument parameters in this subroutine are:

H	Time step
NSTEP	Number of time steps
N	Number of dependent variables (same as the number of differential equations)
Y	An N-vector of dependent variables $\mathbf{y}$
F	An N-vector which upon return will contain $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}, t)$
F1, F2, F3, F4, YY	N-vectors of working arrays

Subroutine RUNGK4 is as follows:

```
SUBROUTINE RUNGK4 (T,H,NSTEP,N,Y,F,F1,F2,F3,F4,YY)
DIMENSION Y(N),F(N),F1(N),F2(N),F3(N),F4(N),YY(N)
HH=0.5*H
TS=T
WRITE (1,200)
DO 100 I=1,NSTEP
   WRITE (1,210) T,(Y(J),J=1,N)
   CALL DIFEQN (T,N,Y,F)
   DO 10 J=1,N
```

[†]The subroutine RUNG4 in the program DAP of Chap. 10 is a slightly modified version of this subroutine.

```
 10        F1(J)=H*F(J)
        TT=T+HH
        DO 20 J=1,N
 20        YY(J)=Y(J)+0.5*F1(J)
        CALL DIFEQN (TT,N,YY,F)
        DO 30 J=1,N
           F2(J)=H*F(J)
 30        YY(J)=Y(J)+0.5*F2(J)
        CALL DIFEQN (TT,N,YY,F)
        TT=T+H
        DO 40 J=1,N
           F3(J)=H*F(J)
 40        YY(J)=Y(J)+F3(J)
        CALL DIFEQN (TT,N,YY,F)
        T=TS+H*FLOAT(I)
        DO 50 J=1,N
           F4(J)=H*F(J)
 50        Y(J)=Y(J)+(F1(J)+2.0*F2(J)+2.0*F3(J)+F4(J))/6.0
100 CONTINUE
200 FORMAT (5X,'    TIME    Y')
210 FORMAT (5X,4F10.6)
    RETURN
    END
```

This subroutine is written in a form that will handle one or more first-order differential equations. It calls subroutine DIFEQN for evaluating $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}, t)$.

Example 12.1

Write a computer program, making use of subroutine RUNGK4, to solve $\dot{y} = -y^2$ with the initial condition $y^0 = 1$.

Solution A main program and two subroutines INITL and DIFEQN for this problem are:

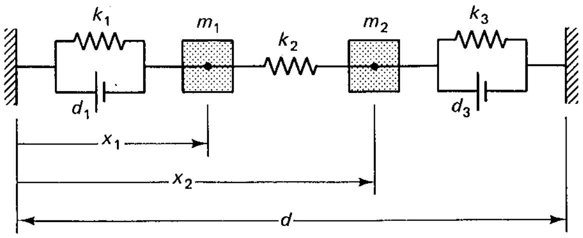
```
C*****MAIN PROGRAM*******
      DIMENSION A(80)
C.....Data
      N=1
      T=0.0
      H=0.1
      NSTEP=50
C.....Pointers
      N1=1
      N2=N1+N
      N3=N2+N
      N4=N3+N
      N5=N4+N
      N6=N5+N
      N7=N6+N
C.....Initial Conditions
      CALL INITL (N,A(N1))
C.....Integration
      CALL RUNGK4 (T,H,NSTEP,N,A(N1),A(N2),A(N3),A(N4),A(N5),
     +              A(N6),A(N7))
      STOP
      END

      SUBROUTINE INITL (N,Y)
      DIMENSION Y(N)
      Y(1)=1.0
      RETURN
      END
```

```
SUBROUTINE DIFEQN (T,N,Y,F)
DIMENSION Y(N),F(N)
F(1)=-Y(1)**2
RETURN
END
```

Subroutine INITL is used to specify the initial conditions. The result of this numerical computation can be compared against the exact solution $y = 1/(1 + t)$.

Example 12.2

The equations of motion for the spring-mass system shown in the schematic diagram are:



$$m_1\ddot{x}_1 = -k_1(x_l - l_1^0) - d_1\dot{x}_1 + k_2(x_2 - x_1 - l_2^0)$$

$$m_2\ddot{x}_2 = -k_2(x_2 - x_1 - l_2^0) + k_3(d - x_2 - l_3^0) - d_3\dot{x}_2$$

Solve these equations numerically for

$$d = 3, \qquad l_1^0 = l_2^0 = l_3^0 = 1, \qquad k_1 = k_2 = k_3 = 100,$$
$$m_1 = m_2 = 4, \qquad d_1 = d_3 = 40$$

and the initial conditions

$$x_1^0 = 1.0, \qquad \dot{x}_1^0 = 0, \qquad x_2^0 = 1.9, \qquad \dot{x}_2^0 = 0$$

Solution The second-order differential equations can be converted to first-order equations by defining four new variables:

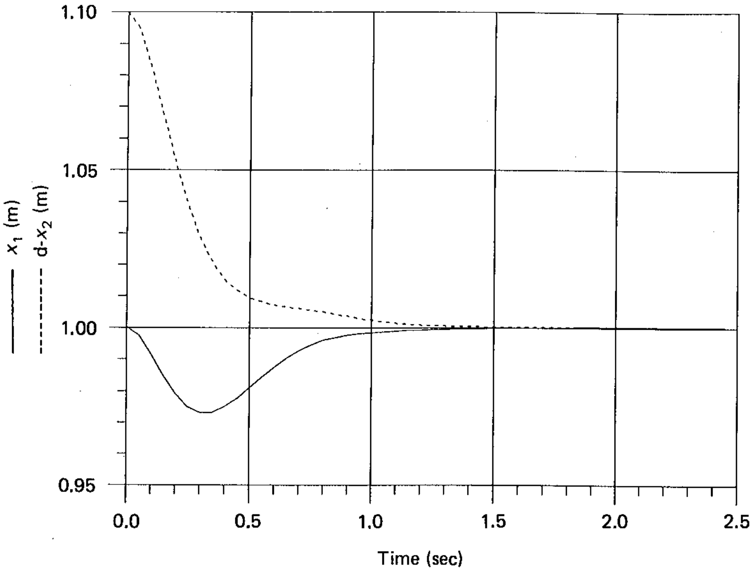
$$\dot{y}_1 = y_2$$
$$\dot{y}_2 = -\frac{k_1}{m_1}(y_1 - l_1^0) - \frac{d_1}{m_1}y_2 + \frac{k_2}{m_1}(y_3 - y_1 - l_2^0)$$
$$\dot{y}_3 = y_4$$
$$\dot{y}_4 = -\frac{k_2}{m_2}(y_3 - y_1 - l_2^0) + \frac{k_3}{m_2}(d - y_3 - l_3^0) - \frac{d_3}{m_2}y_4$$

In the main program, N is set to N = 4, and the INITL and DIFEQN subroutines are written as follows:

```
SUBROUTINE INITL (N,Y)
DIMENSION Y(N)
Y(1)=1.0
Y(2)=0.0
Y(3)=1.9
Y(4)=0.0
RETURN
END
```

```
SUBROUTINE DIFEQN (T,N,Y,F)
DIMENSION Y(N),F(N)
DATA A1/25.0/,A2/10.0/,A3/25.0/,A4/25.0/,A5/10.0/
DATA EL01/1.0/,EL02/1.0/,EL03/1.0/,D/3.0/
F(1) =  Y(2)
F(2) = -A1*(Y(1)-EL01)-A2*Y(2)+A3*(Y(3)-Y(1)-EL02)
F(3) =  Y(4)
F(4) = -A3*(Y(3)-Y(1)-EL02)+A4*(D-Y(3)-EL03)-A5*Y(4)
RETURN
END
```

The result of this computation is shown in the accompanying graph for x_1 and $d - x_2$ plotted versus time.



12.3 POLYNOMIAL APPROXIMATION[2]

Any algorithm that is capable of calculating the exact value $y(t^{i+1})$ for an initial-value problem that has an exact solution in the form of a kth-degree polynomial is called a *numerical integration formula* of order k. Of course, if the exact solution is not a polynomial, a numerical integration formula will generally give only an approximate value y^{i+1} and not the exact value $y(t^{i+1})$. In view of a classical theorem that asserts that any continuous function can be approximated arbitrarily within any closed interval by a polynomial of sufficiently high degree, it is clear that even if the solution is not a polynomial, a numerical integration formula of sufficiently high order can, in principle, be used to calculate $y(t^{i+1})$ to any desired accuracy. In practice, however, the amount of computation and round-off error increases with the order of the integration formula and only orders of $k < 10$ are of practical value.

In contrast to the procedure in the Taylor and Runge-Kutta algorithms, information from previous time steps is utilized in most numerical integration formulas to compute y^{i+1}. A numerical integration formula is generally of the following form:

$$\begin{aligned} y^{i+1} &= a_0 y^i + a_1 y^{i-1} + \cdots + a_p y^{i-p} \\ &\quad + h[b_{-1} f(y^{i+1}, t^{i+1}) + b_0 f(y^i, t^i) + \cdots + b_p f(y^{i-p}, t^{i-p})] \end{aligned} \tag{12.18}$$

where $a_0, a_1, \ldots, a_p, b_{-1}, b_0, b_1, \ldots, b_p$ are $2p + 3$ coefficients that are to be determined such that, if the exact solution is a polynomial and if the previously calculated values $y^i, y^{i-1}, \ldots, y^{i-p}$ and $f(y^i, t^i), f(y^{i-1}, t^{i-1}), \ldots, f(y^{i-p}, t^{i-p})$ are assumed to be exact, then Eq. 12.18 gives the exact value of y^{i+1}. A numerical integration algorithm with $p \geq 1$ is called a *multistep* algorithm, in contrast to the Taylor and Runge-Kutta algorithms, which are *single-step* algorithms.

Note that Eq. 12.18 defines y^{i+1} only *implicitly*, since the unknown y^{i+1}appears on both sides of the equation. Thus, algorithms with $b_{-1} \neq 0$ are called *implicit algorithms*. If $b_{-1} = 0$, the algorithm is an *explicit algorithm,* since the unknown y^{i+1} does not appear on the right side of the formula. Taylor and Runge-Kutta algorithms can be classified as explicit algorithms.

12.3.1 Explicit Multistep Algorithms[2]

Explicit multistep algorithms known as *Adams-Bashforth algorithms* are obtained by setting

$$\begin{aligned} p &= k - 1 \\ a_1 &= a_2 = \ldots = a_{k-1} = 0 \\ b_{-1} &= 0 \end{aligned} \tag{12.19}$$

in Eq. 12.18, where k is the degree of the polynomial. Table 12.1 shows four formulas for first- through fourth-order Adams-Bashforth algorithms. Examination of Table 12.1 shows that the kth-order Adams-Bashforth algorithm requires k starting values $y^i, y^{i-1}, \ldots, y^{i-k+1}$.

TABLE 12.1 Adams-Bashforth Algorithms

Order	$y^{i+1} =$
1st	$y^i + hf^i$
2nd	$y^i + h(\frac{3}{2}f^i - \frac{1}{2}f^{i-1})$
3rd	$y^i + h(\frac{23}{12}f^i - \frac{16}{12}f^{i-1} + \frac{5}{12}f^{i-2})$
4th	$y^i + h(\frac{55}{24}f^i - \frac{59}{24}f^{i-1} + \frac{37}{24}f^{i-2} - \frac{9}{24}f^{i-3})$
where	$f^i \equiv f(y^i, t^i)$
	$f^{i-j} \equiv f(y^{i-j}, t^{i-j}), \qquad j = 1, 2, 3$

12.3.2 Implicit Multistep Algorithms[2]

Implicit multistep algorithms known as *Adams-Moulton algorithms* are obtained by setting

$$\begin{aligned} p &= k - 2 \\ a_1 &= a_2 = \ldots = a_{k-2} = 0 \end{aligned} \tag{12.20}$$

in Eq. 12.18, where k is the degree of the polynomial. Table 12.2 shows four formulas for first- through fourth-order Adams-Mouton algorithms. Examination of Table 12.2

TABLE 12.2 Adams-Moulton Algorithms

Order	$y^{i+1} =$
1st	$y^i + hf^{i+1}$
2nd	$y^i + h(\frac{1}{2}f^{i+1} + \frac{1}{2}f^i)$
3rd	$y^i + h(\frac{5}{12}f^{i+1} + \frac{8}{12}f^i - \frac{1}{12}f^{i-1})$
4th	$y^i + h(\frac{9}{24}f^{i+1} + \frac{19}{24}f^i - \frac{5}{24}f^{i-1} + \frac{1}{24}f^{i-2})$
where	$f^{i+1} = f(y^{i+1}, t^{i+1})$
	$f^i = f(y^i, t^i)$
	$f^{i-j} = f(y^{i-j}, t^{i-j}), \quad j = 1, 2$

shows that the kth-order Adams-Moulton algorithm requires only $k - 1$ starting values $y^i, y^{i-1}, \ldots, y^{i-k+2}$. These formulas, if employed by themselves, are solved iteratively. In every time step, an initial estimate is given for y^{i+1}, and then $f(y^{i+1}, t^{i+1})$ is evaluated. These values for y and f substituted in the formula yields an improved value for y^{i+1}, and the sequence is repeated until very little change in y^{i+1} is observed. The number of iterations to achieve convergence on y^{i+1} depends on the estimated value of y^{i+1}.

12.3.3 Predictor-Corrector Algorithms

Consider the implicit numerical integration algorithms in Sec. 12.3.2. In these algorithms, an estimate of y^{i+1} is required to start the iteration of the formula. In order to obtain a relatively good estimate for y^{i+1}, an explicit formula can be used. For example, consider the fourth-order Adams-Moulton formula of Table 12.2 and the third-order Adams-Bashforth formula of Table 12.1. Both formulas require values of $f(y, t)$ at t^i, t^{i-1}, and t^{i-2}. If the third-order explicit formula is employed, a good approximation on y^{i+1} can be obtained. This step is known as a *predictor* step. Then, the fourth-order implicit formula is employed to correct the predicted value of y^{i+1}. This step is known as a *corrector* step. Sometimes an algorithm may iterate on the corrector step.

In most predictor-corrector algorithms, if a kth-order implicit formula is used as the corrector, a k − 1st-order explicit formula is used as the predictor. Although it is possible to employ lower-order Taylor or Runge-Kutta formulas as predictors, numerically it is more efficient to stay with Admas-Bashforth formulas for prediction.

12.3.4 Methods for Starting Multistep Algorithms[2]

In contrast to single-step algorithms, multistep numerical integration algorithms are not *self-starting*, since initially only y^0 and t^0 are given. For example, it suffices to consider the simpler case in which $b_{-1} = 0$ in Eq. 12.18 and write out y^1 explicitly as follows:

$$y^1 = a_0 y^0 + a_1 y^{-1} + \ldots + a_p y^{-p} + h[b_0 f(y^0, t^0) + b_1 f(y^{-l}, t^{-1} + \ldots + b_p f(y^{-p}, t^{-p})] \qquad (12.21)$$

Equation 12.21 shows that the values $y^{-1}, y^{-2}, \ldots, y^{-p}$ must be given, in addition to y^0 and t^0, in order to compute y^1. In general, to compute y^{i+1}, the $p + 1$ preceding values of y are needed, assuming a uniform step size h. To obtain these values, a single-step algorithm must be used at least $p + 1$ times before a multistep algorithm can be initiated. Because of its high degree of accuracy, the fourth-order Runge-Kutta algorithm is frequently used to provide these initiating values.

Efficient and accurate numerical algorithms for solving initial-value problems are almost always a combination of single-step and multistep algorithms, with the former used only to obtain the starting values for initiating the latter. Multistep algorithms are used to compute the remaining points, because they are often computationally more efficient and with them the propagation of both truncation and round-off errors can be more easily controlled.

12.4 ALGORITHMS FOR STIFF SYSTEMS[2]

A stiff system is referred to as any initial-value problem in which the complete solution consists of *fast* and *slow* components. Technically, when the *eigenvalues* are widely spread, the system is said to be stiff. If a numerical solution is to display the entire transient response, integration must be performed over a relatively long time interval in order to cover the slow component(s) of the response. Furthermore, in order to capture the fast component(s) of the response and keep the numerical error within bounds, the selected step size must be relatively small. It is clear that carrying integration with small time steps over a long time interval can make the computer time, even for a small problem, prohibitive or unrealistic. A family of formulas that allow relatively large time steps and that guarantee stability and bounded numerical error is available. These multistep formulas are known as Gear algorithms.

The kth-order Gear algorithm is an implicit formula of the form

$$y^{i+1} = a_0(k)y^i + a_1(k)y^{i-1} + \cdots + a_{k-1}(k)y^{i-k+1} + hb_{-1}(k)f(y^{i+1}, t^{i+1}) \qquad (12.22)$$

where the designation $a_j(k)$ emphasizes each coefficient's dependence on the order k. The $k + 1$ coefficients $a_0(k), \ldots, a_{k-1}(k)$, and $b_{-1}(k)$ are to be determined so that Eq. 12.22 is exact for all polynomial solutions of degree k. Table 12.3 shows four formulas for first- through fourth-order Gear algorithms. Examination of Table 12.3 shows that the kth-order Gear algorithm requires k starting values $y^i, y^{i-1}, \ldots, y^{i-k+1}$.

TABLE 12.3 Gear Algorithms

Order	$y^{i+1} =$
1st	$y^i + hf^{i+1}$
2nd	$\frac{4}{3}y^i - \frac{1}{3}y^{i-1} + \frac{2}{3}hf^{i+1}$
3rd	$\frac{18}{11}y^i - \frac{9}{11}y^{i-1} + \frac{2}{11}y^{i-2} + \frac{6}{11}hf^{i+1}$
4th	$\frac{48}{25}y^i - \frac{36}{25}y^{i-1} + \frac{16}{25}y^{i-2} - \frac{3}{25}y^{i-3} + \frac{12}{25}hf^{i+1}$
where	$f^{i+1} = f(y^{i+1}, t^{i+1})$

Since the kth-order Gear algorithm is an implicit multistep algorithm, it is necessary to solve an implicit equation in each time step. The kth-order formula of Eq. 12.22, for a system of equations

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}, t)$$

can be recast into the form

$$\mathbf{y}^{i+1} - hb_{-1}\mathbf{f}(\mathbf{y}^{i+1}, t^{i+1}) - \sum_{j=0}^{k-1} (a_j \mathbf{y}^{i-j}) = \mathbf{0} \tag{12.23}$$

Applying the Newton-Rhapson algorithm to Eq. 12.23 yields

$$\Delta\mathbf{y}^{i+1} = \left(\mathbf{I} - b_{-1}\frac{\partial \mathbf{f}^{i+1}}{\partial \mathbf{y}^{i+1}}\right)^{-1}\left[\mathbf{y}^{i+1} - hb_{-1}\mathbf{f}^{i+1} - \sum_{j=1}^{k-1} (a_j \mathbf{y}^{i-j})\right] \tag{12.24}$$

where **I** is an identity matrix. Equation 12.24 is the Newton-Raphson *corrector* for implementing the Gear algorithm.

12.5 ALGORITHMS FOR VARIABLE ORDER AND STEP SIZE

So far, it has been implicitly assumed that, given an initial-value problem, a numerical integration algorithm of certain "order" is selected and the order remains *fixed* during the entire integration process. Under this assumption, the *step size* for each time step may be optimized by choosing the largest possible value of h for which the truncation error remains bounded below the user-specified *maximum allowable error,* and for which the algorithm remains numerically stable. For large systems of equations, the amount of computation does not increase substantially when the order of the algorithm is increased. Consequently, it often turns out to be more efficient to vary both the *order* and the *step size* during each time step.

From a programming point of view, changing the order requires only selecting a set of coefficients that define the multistep algorithm of the desired order. Increasing (decreasing) the order would require an increase (decrease) in the number of coefficients, with a corresponding increase (decrease) in storage space. In most cases of practical interest, the order may vary from $k = 1$ to $k = 10$. Thus, enough "past" values must be stored that the highest-order algorithm can be implemented whenever called for. However, stored "past" values may not be needed if a lower-order algorithm is used. In any event, the unused values cost very little, since they require only a modest amount of storage space.

Unlike change of order, which requires little extra programming and computational effort, changing the step size could entail considerable additional computation time. Often, the previously stored "past" values corresponding to step size h must be interpolated to yield a set of transformed "past" values corresponding to the new step size h.

PROBLEMS

12.1 Solve the following types of problem with subroutine RUNGK4:

(a) A first-order differential equation

(b) A second-order differential equation

(c) A system of second-order differential equations.

Repeat each problem for different values of integration time steps and compare the results. If the exact solution to the problem is available, compare that against the numerical solutions.

12.2 Refer the Table 12.1 and develop subroutines for the following integration algorithms:

- **(a)** The third-order Adams-Bashforth
- **(b)** The fourth-order Adams-Bashforth

12.3 Refer to Table 12.2 and develop subroutines for the following integration algorithms:

- **(a)** The third-order Adams-Moulton
- **(b)** The fourth-order Adams-Moulton

12.4 Develop a predictor-corrector integration subroutine by employing a third-order Adams-Bashforth algorithm and a fourth-order Adams-Moulton algorithm.

12.5 Compare the subroutines developed in Probs. 12.2 to 12.4 in terms of accuracy and computational efficiency by following a process similar to that stated in Prob. 12.1.

12.6 Refer to the software library of your computer and experiment with the available integration subroutines.

12.7 Find a computer program for a variable step/order predictor-corrector numerical integration algorithm.[†]

- **(a)** Implement this program on your computer.
- **(b)** Compare this algorithm with the subroutines developed in Probs. 12.2 to 12.4.

12.8 Experiment with numerical integration programs based on Gear algorithms.[‡] Compare these algorithms and those developed in Probs. 12.2 to 12.4.

[†]An excellent variable step/order predictor-corrector algorithm can be found in L. F. Shampine, and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: The Initial Value Problem,* W. H. Freeman, San Francisco, 1975.

[‡]Most scientific software libraries furnish numerical integration packages based on Gear algorithms. For more detailed discussion on these algorithms, refer to C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations,* Prentice-Hall, Englewood Cliffs, N.J., 1971.