

Numerical Solution of Ordinary Differential Equations

Goal of these notes

These notes were prepared for a standalone graduate course in numerical methods and present a general background on the use of differential equations. The numerical material to be covered in the 501A course starts with the section on the plan for these notes on the next page.

Background on differential equations

Many engineering problems are defined in terms of differential equations. Most students encounter their first application of differential equations to physical problems in the analysis of motion. Here, there are two common differential equations. The first relates the displacement along a one-dimensional path, s , to the velocity, v ; the second, which is Newton's second law relates the displacement to the applied force divided by the mass, F/m . The symbol t indicates the time.

$$\frac{ds}{dt} = v \qquad \frac{d^2s}{dt^2} = \frac{F}{m} \qquad [1]$$

Another application of differential equations is in electrical circuits. The current, I , in a circuit with a capacitance, C , an inductance, L , a resistance, R , and an applied voltage, $V(t)$ is governed by the following differential equation. (In this equation, $V(t)$ is known.)

$$L \frac{d^2I}{dt^2} + R \frac{dI}{dt} + \frac{1}{C} I = \frac{dV(t)}{dt} \qquad [2]$$

Differential equations are classified in terms of the highest order of the derivative that appears in the equation. Thus, equation [2] is a second order differential equation. The two differential equations in [1] are, respectively, first-order equation and second-order differential equations. The equations in [1] and [2] are linear differential equations. In these equations, the dependent variable, and all its derivatives, appear to the first power only.

We can use the definition of velocity to write Newton's second law as two first-order differential equations.

$$\frac{ds}{dt} = v \qquad \frac{dv}{dt} = \frac{F}{m} \qquad [3]$$

Similarly, we can use the definition of voltage drop across the inductor, e_L as the Inductance times the first derivative of the current, to rewrite equation [2] as the following pair of equations.

$$e_L = L \frac{dI}{dt} \qquad \frac{de_L}{dt} + \frac{e_L}{R} + \frac{1}{C} I = \frac{dV(t)}{dt} \qquad [4]$$

In this system of equations, we have one independent variable, t , and two dependent variables, l and e_L . This approach of writing second-order equations as sets of first-order equations is possible for any higher order differential equation. We will use it subsequently to apply algorithms designed for the analysis of first order equations to systems of higher order equations.

Some differential equations can be solved by simple integration. An example of this is shown below.

$$\frac{ds}{dt} = v(t) \quad \text{so that} \quad \int ds = \int v(t)dt \quad \text{or} \quad s = \int v(t)dt + C \quad [4]$$

The constant of integration, C , can be found if one point in the relationship, typically called an initial condition, $(s_{initial}, t_{initial})$ is known. With the initial condition, we can find the value of s for any value of t by the following integration.

$$\int_{s_{initial}}^s ds = s - s_{initial} = \int_{t_{initial}}^t v(t')dt' \quad \text{or} \quad s(t) = s_{initial} + \int_{t_{initial}}^t v(t')dt' \quad [5]$$

We have used the dummy variable, t' , in the integral to indicate that the final dependence of $s(t)$ depends on the upper limit of the integral.

We could perform the integration in equations [4] and [5] because the derivative expression was a function of the time only. We are interested in the more general problem of what happens when the derivative in equation [4] is a function of both time and displacement. That is, we are interested in solving the general problem

$$\frac{ds}{dt} = v(t, s) \quad [6]$$

We can try to write this as we did in equation [4] or [5], but we cannot perform the integration because v is a function of both s and t .

$$\int ds = \int v(t, s)dt \quad \text{or} \quad s = s_{initial} + \int_{t_{initial}}^t v(t', s)dt' \quad [7]$$

There are many cases in which we can solve differential equations like equation [6] analytically. However, when we cannot do so, we have to find numerical methods for solving this equation.

Plan for these notes

The general approach to the numerical solution of ordinary differential equations defines a general initial value problem (IVP) which is shown in equation [8].

$$\frac{dy}{dx} = f(x, y) \quad \text{with a known initial condition} : y(x_0) = y_0 \quad [8]$$

We will develop our algorithms for this simple problem of a single differential equation.

Initially we will describe a general approach for solving the IVP, including a discussion of the notation and error terms. Next, we will examine some simple algorithms that we can use. These

simple algorithms will help us see how the solutions proceed in general and allow us to examine the kinds of errors that occur in the numerical solution of ODEs. We will address considerations of accuracy and the selection of a step size that provides the desired accuracy.

Next, we will consider applying our algorithms to systems of equations. As discussed above, we will reduce higher-order equations to systems of first-order equations. In addition to this method for obtaining systems of equations, we will be able to address engineering problems that involve systems of differential equations. Many such problems occur in “networks” which may be a transient electrical circuit, the behavior of a structure in an earthquake, or a transportation network. In general, codes for the numerical solution of ODEs are written for systems of equations and can then be applied to any number of equations, including a single equation.

The simple algorithms that we will consider initially are called self-starting algorithms; they require no information from previous steps for their operation. However, we will want to consider multistep algorithms which use information from previous steps. These algorithms can obtain results with similar accuracy to self-starting algorithms with less computational effort. (Of course, we will need to link these with a self-starting algorithm to start the calculations from the initial condition.)

The final section of the notes will discuss approaches for more boundary value problems and eigenvalue problems. Boundary value problems have fixed values of y at two different values of x (as opposed to initial value problems where we know the value of y and some of the derivatives of y at an initial value of x). Eigenvalue problems typically occur when the number of boundary conditions is larger than the order of the differential equation.

General Approach

The general problem for which we will develop an algorithm is called the initial value problem or IVP. The definition of this problem from equation [8] above is repeated below.

$$\frac{dy}{dx} = f(x, y) \quad \text{with a known initial condition : } y(x_0) = y_0 \quad [8]$$

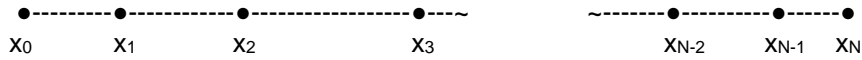
We will generally have an equation to compute $f(x,y)$ for any x and y values. We want to find a numerical approximation to the behavior of the function $y(x)$ between the initial value x_0 , and some final (given) value, x_{\max} .

Although the derivative is regarded as a function of x and y , the independent variable in the differential equation (x in equation [8]) is a variable that we will control to determine the intervals, Δx , to which we apply our algorithms. If we could solve the equation analytically we could obtain y as a function of x . The goal of the numerical solution is to produce a table of numerical results giving the values of y for a given set of values for x .

All algorithms work by dividing the region between x_0 and x_{\max} into a grid of $N+1$ discrete points at the locations $x = x_i$, where i ranges from 0 to N . The spacing between the points may be uniform or non-uniform. The coordinate of the first node, x_0 is the same as the initial point, which is also called x_0 . The final grid node, called x_N , is located at the final value of x , x_{\max} . The spacing between any two grid nodes, x_i and x_{i-1} , has the symbol $h_i = \Delta x_i$. These relations are summarized below.

$$x_0 = x_{\min} \quad x_N = x_{\max} \quad h_i = x_i - x_{i-1} = \Delta x_i \quad [9]$$

A non-uniform grid, with different spacing between different nodes, is illustrated below.



For a uniform grid, all values of Δx_i are the same. In this case, the uniform grid spacing, in a one-dimensional problem, is given the symbol, h . I.e., $h = x_i - x_{i-1}$ for all values of i .

There are N values of h_i , with i ranging from 1 to N in the definition of h_i . (There is one more grid point, x_i , than the number of grid steps, h_i .) If the grid spacing is uniform, we can calculate the value of h from the following equation.

$$h = \frac{x_{\max} - x_0}{N} \tag{10}$$

Furthermore, in this uniform case the value of any x coordinate x_i is simply found as follows.

$$x_i = x_0 + ih \tag{11}$$

The user selects an appropriate value of h (or N) to provide the desired accuracy. In more general algorithms, the value of h is adjusted during the calculation to provide small step sizes in regions where there is a large variation in $f(x,y)$ and larger step sizes where the variation in $f(x,y)$ is low.

We use the following notation in discussing the numerical solution of ODEs.

- x_i is the value of the x point along the grid. This is determined from the value of h (or the series of h_i values) determined by the user. Since x is the independent variable, it's value can always be specified exactly.
- y_i is the value of the numerical solution at the point where $x = x_i$.
- f_i is the value of the derivative computed from the known value of x_i and the numerical value, y_i . I.e., $f_i = f(x_i, y_i)$.
- $y(x_i)$ is the exact value of y at $x = x_i$. This value is usually not known. This notation is used in the error analysis of algorithms.
- $f(x_i, y(x_i))$ is the exact value of the derivative at $x = x_i$. This value is generally not known, but the notation is used in error analysis of algorithms.
- $e_1 = y(x_1) - y_1$ is the local truncation error; this is defined as the error after the first step from the initial point, x_0 , where the initial value of y , $y(x_0)$ is known.
- $E_j = y(x_j) - y_j$ is the global truncation error. Although both the local and global truncation error appear to have the same definition, we notice that the local error is defined as the error after one step. That is, the local error is the error **in one step from a known initial condition**.

The basic idea for the numerical solution of ODEs is quite simple. If we replace the derivative term in equation [8] by finite differences over the two points x_{i+1} and x_i , and replace the value of $f(x,y)$ as some suitable average between points x_i and x_{i+1} , we get the following result.

$$\frac{y_{i+1} - y_i}{x_{i+1} - x_i} = \frac{y_{i+1} - y_i}{h_{i+1}} = f_{average} \quad \Rightarrow \quad y_{i+1} = y_i + h_{i+1} f_{average} \tag{12}$$

The whole approach to solving ODEs numerically is the derivation of equations for $f_{average}$ that are both accurate and easy to use.

We start the numerical process with the known the initial condition, (x_0, y_0) . We then use the result of equation [12] to take the step from y_0 to y_1 . (The value of x_1 is found from equation [9], $x_1 = x_0 + h_1$.) Once we have a value for x_1 and y_1 . We can then use equation [12] to take the next step from (x_1, y_1) to (x_2, y_2) . We continue this process until we reach the desired ending point, x_{max} . We now have to address the issue of the computation of $f_{average}$.

Euler's Method

Euler's method is the simplest algorithm for the numerical solution of ordinary differential equations. It is never used in practice, but it is helpful to illustrate the general approach used in solving ODEs numerically.

In Euler's method, the new value of the independent variable is given by the following equation.

$$y_{i+1} = y_i + h_{i+1}f(x_i, y_i) \quad \text{with} \quad x_{i+1} = x_i + h_{i+1} \quad [13]$$

In this approach, we are taking a simple, but crude approximation to $f_{average}$. We are assuming that the value of the derivative at the start of the step, $f(x_i, y_i)$, is the average value over the entire step. We use this as a basic tool for analyzing the error in numerical solutions of ordinary differential equations.

Error in the Solution of ODEs

Two different error terms are defined in the numerical solution of ordinary differential equations. The first, called the **local error**, is the error obtained in one step when the starting point is known exactly. This is usually true only in the first solution step when we are starting from the initial condition. We are generally more interested in the **global error**. That is the error after some number of steps.

We can analyze the error in the Euler method by writing a Taylor series for the exact value of y after one grid step in terms of the initial value of y . The usual Taylor series for $y(x)$ expressed in terms of $y(a)$, the value of y at a point where $x = a$, is shown below.

$$y(x) = y(a) + \left. \frac{dy}{dx} \right|_{x=a} (x - a) + \left. \frac{1}{2!} \frac{d^2 y}{dx^2} \right|_{x=a} (x - a)^2 + \left. \frac{1}{3!} \frac{d^3 y}{dx^3} \right|_{x=a} (x - a)^3 + \dots \quad [14]$$

We want a Taylor series for the value of y at the end of the first step, $y(x_0 + h)$, expressed as a Taylor series about the initial point, x_0 . We get this series from equation [14] by setting $a = x_0$ and $x = x_0 + h$. When we do this the $(x - a)$ terms become $(x_0 + h - x_0) = h$. The resulting Taylor series that we want is shown below.

$$y(x_0 + h) = y(x_0) + \left. \frac{dy}{dx} \right|_0 h + \left. \frac{1}{2!} \frac{d^2 y}{dx^2} \right|_0 h^2 + \left. \frac{1}{3!} \frac{d^3 y}{dx^3} \right|_0 h^3 + \dots \quad [15]$$

In the above equation, we use the notation $|_0$ on the derivatives to indicate that they are evaluated at $x = x_0$. We know that dy/dx has the symbol f for the usual derivative in our initial value problem. Using this definition, we can rewrite equation [15] and identify the terms that are used in the Euler Algorithm. The remaining terms are the local truncation error.

$$y(x_0 + h) = y(x_0) + hf(x_0, y(x_0)) + \frac{1}{2!} \frac{d^2 y}{dx^2} \Big|_0 h^2 + \frac{1}{3!} \frac{d^3 y}{dx^3} \Big|_0 h^3 + \dots \quad [16]$$

Euler Algorithm Truncation Error

We see that the Euler method has a local truncation error that is second order.

$$e_1 = y(x_0 + h) - y_0^{Euler} = \frac{1}{2!} \frac{d^2 y}{dx^2} \Big|_0 h^2 + \frac{1}{3!} \frac{d^3 y}{dx^3} \Big|_0 h^3 + \dots = y_0^{Euler} + O(h^2) \quad [17]$$

We now want to prove the following general result: if the local truncation error for an algorithm is $O(h^n)$, its global truncation error is $O(h^{n-1})$. To do this we assume that an error like the local truncation error is produced in each step. Thus, after we take k steps, we have a global error that is approximately k times the local truncation error. If the local truncation error is $O(h^n) \approx Ah^n$, we can write the global error, for k steps of size h , as follows.

$$E_k(h) \approx ke_1(h) \approx kAh^n \quad [18]$$

If we cut the step size by a factor of r , so that the new step size is h/r , we can rewrite equation [18] as follows for the new step size.

$$E_k\left(\frac{h}{r}\right) \approx ke_1\left(\frac{h}{r}\right) \approx kA\left(\frac{h}{r}\right)^n \quad [19]$$

To get to the same value of x with the smaller step size, we have to take kr steps. Thus, the global error at the same x location is obtained by substituting kr for k in equation [19].

$$E_{kr}\left(\frac{h}{r}\right) \approx kre_1\left(\frac{h}{r}\right) \approx krA\left(\frac{h}{r}\right)^n \quad [20]$$

We now examine the ratio of the two global truncation errors, at the same x location, given by equations [20] and [18].

$$\frac{E_{kr}\left(\frac{h}{r}\right)}{E_k(h)} \approx \frac{krA\left(\frac{h}{r}\right)^n}{kA(h)^n} = \frac{1}{r^{n-1}} \quad [21]$$

Equation [20] tells us that when we cut the step size by a factor of r , the error decreases by a factor of r^{n-1} . This is the result we obtain for an error that has an order $n-1$. Consequently, we conclude that a method, which has a local truncation error that is $O(h^n)$, has a global truncation error which is $O(h^{n-1})$.

An example problem

We will apply Euler's method to the following simple problem.

$$\frac{dy}{dx} = x + y \quad \text{with : } y_0 = 0 \text{ at } x_0 = 0 \quad [22]$$

The analytical solution to this equation is shown below. You can verify that this solution satisfies the differential equation and the initial condition.

$$y = e^x - x - 1 \tag{23}$$

Normally we will apply numerical methods to differential equations that we cannot solve analytically. However, we will use the differential equation in [22] and its analytical solution in [23], to check the error in our numerical methods.

For purposes of illustration we will pick a constant step size, $h = 0.1$. We will consider the problem of selecting step sizes to obtain the accuracy desired by the user later.

If we compare the specific example in equation [22] with the general problem statement in equation [8], we see that $f(x,y) = x + y$ in this problem. At the initial condition of $x_0 = 0$ and $y_0 = 0$, we have $f_0 = f(x_0,y_0) = 0 + 0 = 0$. So Euler's method from equation [13] gives us $y_1 = y_0 + hf_0 = 0 + (0.1)(0) = 0$. From equation [9] we get $x_1 = x_0 + h = 0 + 0.1 = 0.1$.

We next take the step from x_1 to $x_2 = 0.1 + 0.1 = 0.2$. At x_1 , the derivative, $f_1 = f(x_1, y_1) = x_1 + y_1 = 0.1 + 0 = 0.1$. The Euler algorithm in equation [13] gives $y_2 = y_1 + hf_1 = 0 + (0.1)(0.1) = 0.01$. The table below shows these first two steps as well as additional steps for the method. The analytical solution in equation [23] is used to compute the errors.

Table1. Euler method for solution of equation [22]							
i	x_i	y_i	f_i	y_i exact	y_i error	f_i exact	f_i error
0	0	0	0	0	0	0	0
1	0.1	0	0.1	0.005171	0.005171	0.105171	0.005171
2	0.2	0.01	0.21	0.021403	0.011403	0.221403	0.011403
3	0.3	0.031	0.331	0.049859	0.018859	0.349859	0.018859
4	0.4	0.0641	0.4641	0.091825	0.027725	0.491825	0.027725
5	0.5	0.11051	0.61051	0.148721	0.038211	0.648721	0.038211
6	0.6	0.171561	0.771561	0.222119	0.050558	0.822119	0.050558
7	0.7	0.248717	0.948717	0.313753	0.065036	1.013753	0.065036
8	0.8	0.343589	1.143589	0.425541	0.081952	1.225541	0.081952
9	0.9	0.457948	1.357948	0.559603	0.101655	1.459603	0.101655
10	1	0.593742	1.593742	0.718282	0.124539	1.718282	0.124539

Table 1 shows that the error grows with x . This is due to two factors. When we take the initial steps, we have errors in y . Thus, at a given value of x , our algorithm which is basically computing a Δy is adding the new Δy to an incorrect value of y . In addition, we are using the incorrect value of y to compute the derivative, f . Thus the incorrect value of y enters in at two points: (1) as the value of y_i in the equation $y_{i+1} = y_i + hf_i$, and (2) in the computation of $f_i = f(x_i, y_i)$.

Table 1 shows that the Euler method produces a considerable error with the step size, $h = 0.1$. If we reduce the step size, we expect to reduce the error. Since the Euler method has a second-order local truncation error, it should have a first order global truncation error. To verify this, we examine the errors in the initial step and the final solution for $h = 0.1, 0.01, \text{ and } 0.001$ in Table 2.

Table 2 – Euler method errors		
Step size	Initial step error	Final error
$h = 0.1$	5.17×10^{-3}	1.25×10^{-1}
$h = 0.01$	5.02×10^{-5}	1.35×10^{-2}
$h = 0.001$	5.00×10^{-7}	1.36×10^{-3}

The results in this table show that the Euler method has a global error that is first order: cutting the step size by a factor of ten cuts the error by a factor of ten. Table 2 also shows that the local truncation error, measured as the error in the initial step, is second order. Cutting the step size by a factor of ten cuts the error in the first step by a factor of 100.

At this point we can either try to use more steps to reduce our error or try to find a better algorithm. The latter path is generally the best one to pursue. Some of the simpler algorithms used in numerical solution of ordinary differential equations, known as Runge-Kutta methods, are described below.

Runge-Kutta methods

Runge-Kutta methods use two or more evaluations of the derivative over the step from x_i to x_{i+1} . These methods are called self-starting methods because they require no information from previous data points. However, they do require more work per step than the predictor-corrector methods listed below, which use information from previous steps.

Runge-Kutta methods come in various orders. The lowest Runge-Kutta methods are second-order methods, which means that they have a second-order global truncation error. Two of these methods have individual names. The first of these, known as Heun's Method, has an initial step that defines a predicted value of y_{i+1} , called y_{i+1}^0 . This value is then used to estimate the derivative at x_{i+1} . The actual derivative used to get the final value of y_{i+1} is the average of the two derivative values.

Huen's method proceeds in the following steps.

$$\begin{aligned}
 y_{i+1}^0 &= y_i + h_{i+1}f(x_i, y_i) & x_{i+1} &= x_i + h_{i+1} \\
 y_{i+1} &= y_i + \frac{h_{i+1}}{2} [f(x_i, y_i) + f(x_{i+1}, y_{i+1}^0)] = \frac{y_i + y_{i+1}^0 + h_{i+1}f(x_{i+1}, y_{i+1}^0)}{2}
 \end{aligned}
 \tag{24}$$

This follows the general approach outlined in equation [12]. Here we take the average derivative to be the arithmetic mean of two evaluations at the two end points of the interval. Huen's method has a second-order global truncation error.

The other second-order Runge-Kutta method is known as the modified Euler method. This is similar to the Huen method in that it takes a first step, obtains a new estimate of the derivative, and then takes the final step. The modified Euler method takes a half step to evaluate y at the midpoint of the step. This value is called $y_{i+1/2}$. This value of y , as well as the corresponding value of x is used to compute the derivative at the midpoint. This derivative value is then used as the average value of f to compute the new y value. The equations for the modified Euler method are shown below.

$$\begin{aligned}
 y_{i+1/2} &= y_i + \left[\frac{h_{i+1}}{2} \right] f(x_i, y_i) & x_{i+1/2} &= x_i + \frac{h_{i+1}}{2} \\
 y_{i+1} &= y_i + h_{i+1}f(x_{i+1/2}, y_{i+1/2})
 \end{aligned}
 \tag{25}$$

The fourth-order Runge-Kutta method is a long-time favorite of numerical analysts for simple problems, but is not used in modern computer applications. However, it provides a useful illustration of higher-order methods and their effectiveness. This method uses four calculations of the derivative over the step: one at each endpoint and two at the midpoint. The weighted results of these derivative evaluations are then used in the computation of the final y value. The classical notation for the Runge-Kutta algorithm uses the notation k_m to denote the product of a

derivative estimate times the step size. This notation is used below to define the fourth-order (global truncation error) algorithm for this method.

$$\begin{aligned}
 y_{i+1} &= y_i + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} & x_{i+1} &= x_i + h_{i+1} \\
 k_1 &= h_{i+1}f(x_i, y_i) \\
 k_2 &= h_{i+1}f\left(x_i + \frac{h_{i+1}}{2}, y_i + \frac{k_1}{2}\right) \\
 k_3 &= h_{i+1}f\left(x_i + \frac{h_{i+1}}{2}, y_i + \frac{k_2}{2}\right) \\
 k_4 &= h_{i+1}f(x_i + h_{i+1}, y_i + k_3)
 \end{aligned}
 \tag{26}$$

We can illustrate the application of the fourth-order Runge-Kutta method to the example problem (and initial conditions) in equation [22]. For this example $f(x,y) = x + y$. Choosing $h = .1$ gives the following steps.

$$k_1 = hf(x_0, y_0) = h(x_0, y_0) = (0.1)(0 + 0) = 0$$

$$k_2 = hf(x_0+h/2, y_0+k_1/2) = h(x_0+h/2 + y_0+k_1/2) = (0.1)(0+0.1/2 + 0+0/2) = 0.005$$

$$k_3 = hf(x_0+h/2, y_0+k_2/2) = h(x_0+h/2 + y_0+k_2/2) = (0.1)(0+0.1/2 + 0+0.005/2) = 0.00525$$

$$k_4 = hf(x_0+h, y_0+k_3) = h(x_0+h + y_0+k_3) = (0.1)(0+0.1 + 0+0.00525) = 0.10525$$

$$y_1 = y_0 + (k_1 + 2k_2 + 2k_3 + k_4)/6 = 0 + [0 + 2(0.005) + 2(0.00525) + 0.10525]/6 = 0.0051708333$$

Additional steps would be done in the same manner. Table 3 shows the results of applying the fourth-order Runge-Kutta algorithm in [26] to the differential equation in [22] for ten steps.

Table 3 – Results from fourth-order Runge-Kutta method for solution of equation [22], h = 0.1									
i	x _i	y _i	k ₁	k ₂	k ₃	k ₄	Delta y	y _i exact	y _i error
0	0	0	0	0.005	0.00525	0.01053	0.00517	0	0
1	0.1	0.00517	0.01052	0.01604	0.01632	0.02215	0.01623	0.00517	8.47x10 ⁻⁸
2	0.2	0.02140	0.02214	0.02825	0.02855	0.03500	0.02846	0.02140	1.87x10 ⁻⁷
3	0.3	0.04986	0.03499	0.04174	0.04207	0.04919	0.04197	0.04986	3.11x10 ⁻⁷
4	0.4	0.09182	0.04918	0.05664	0.05701	0.06488	0.05690	0.09182	4.58x10 ⁻⁷
5	0.5	0.14872	0.06487	0.07312	0.07353	0.08222	0.07340	0.14872	6.32x10 ⁻⁷
6	0.6	0.22212	0.08221	0.09132	0.09178	0.10139	0.09163	0.22212	8.38x10 ⁻⁷
7	0.7	0.31375	0.10138	0.11144	0.11195	0.12257	0.11179	0.31375	1.08x10 ⁻⁷
8	0.8	0.42554	0.12255	0.13368	0.13424	0.14598	0.13406	0.42554	1.37x10 ⁻⁷
9	0.9	0.55960	0.14596	0.15826	0.15887	0.17185	0.15868	0.55960	1.70x10 ⁻⁷
10	1	0.71828	0.17183	0.18542	0.18610	0.20044	0.18588	0.71828	2.08x10 ⁻⁷

Although algorithms like the fourth-order Runge-Kutta look complex to calculate, they are able to produce accurate solutions with much less computational work. To show this we solve the following differential equation

$$\frac{dy}{dx} = e^{-x-y} \quad \text{with } y(0) = 0 \quad [27]$$

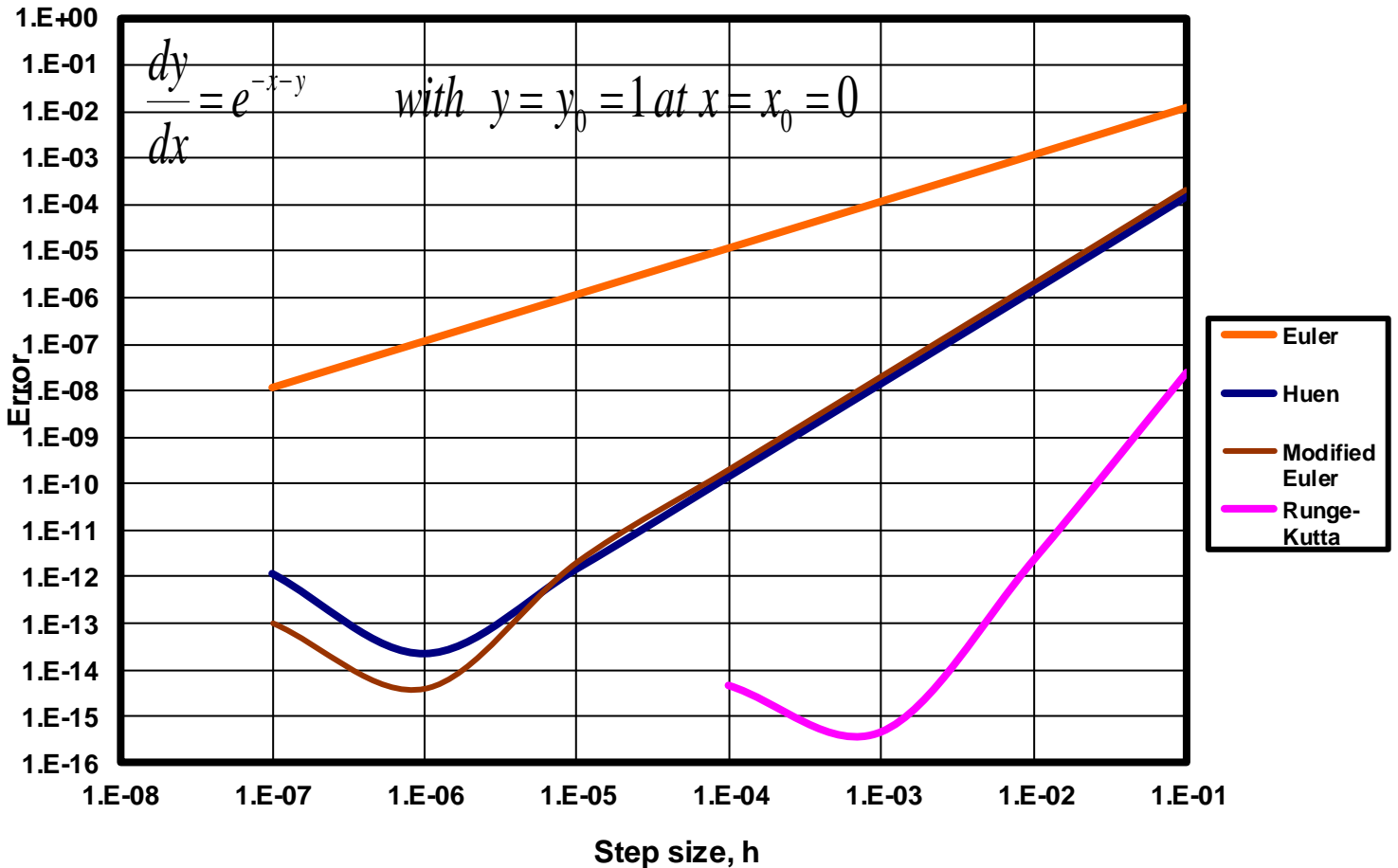
The solution to this differential equation is shown below.

$$y = \ln(e^{y(0)} + e^{-x(0)} - e^{-x}) \quad [28]$$

Can you verify that this solution satisfies the differential equation and the boundary conditions?

The error in the numerical solutions of equation [27] by different methods is shown in the log-log plot below. The first thing to note about this figure is the slope of the log(error) versus log(h) at

Error versus Step Size for Simple ODE Solvers



the right side of the plot, before the roundoff error enters the calculations. We see that this slope, which is the order of the error, has the expected values of 1, 2, 2, and 4, respectively for the Euler, Huen, Modified Euler, and (fourth order) Runge Kutta. The reduction in error by going to a more complex algorithm is particularly dramatic in these results. Reducing the error from 10^{-2} to 10^{-8} for the Euler algorithm requires cutting the step size from $h = 0.1$ to 0.0000001 . This improvement in error this requires 1,000,000 times as much work for the solution with an error of 10^{-8} . In contrast, keeping the step size the same and switching algorithms from Euler's method to

the fourth-order Runge-Kutta reduces the error almost as much without any change in step size. Of course, the fourth-order Runge-Kutta takes more work per step. A rough evaluation of the relative computational work of different algorithms is the number of derivative evaluations required per step. Since the fourth-order Runge-Kutta requires four derivative evaluations per step, compared to the one evaluation for Euler's method, the improvement in switching algorithms provides a million-fold improvement in error with a four-fold increase in computational effort.

Other Runge-Kutta methods are available that include step size control. These methods (known as Runge-Kutta-Fehlberg and Runge-Kutta-Verner) compute two different estimates of y_{i+1} . The difference between these two estimates (whose approximations are based on expressions whose order is different by one) is used as a measure of the error. This error measure is then used to adjust the step size. In this approach the step size is adjusted continuously. Equation [29] shows the typical equation that is used for adjusting the step size and the specific example of the equation that is used for the Runge-Kutta-Fehlberg method. In the general equation, C is a constant, typically less than one, to provide a conservative decrease in the step size and n is the overall order of the method.

$$h_{new} = h_{old} C \left(\frac{error}{desired\ error} \right)^{1/n} \quad [29]$$

The Runge-Kutta-Fehlberg method, which uses two expressions that have a global error of 4 and 5, uses the following set of k .

$$\begin{aligned} k_1 &= hf(x_i, y_i) \\ k_2 &= hf\left(x_i + \frac{h}{4}, y_i + \frac{k_1}{4}\right) \\ k_3 &= hf\left(x_i + \frac{3h}{8}, y_i + \frac{3k_1}{32} + \frac{9k_2}{32}\right) \\ k_4 &= hf\left(x_i + \frac{12h}{13}, y_i + \frac{1932k_1}{2197} - \frac{7200k_2}{2197} + \frac{7296k_3}{2197}\right) \\ k_5 &= hf\left(x_i + h, y_i + \frac{432k_1}{216} - 8k_2 + \frac{3680k_3}{513} - \frac{845k_4}{4104}\right) \\ k_6 &= hf\left(x_i + \frac{h}{2}, y_i - \frac{8k_1}{27} + 2k_2 - \frac{3544k_3}{2565} + \frac{1859k_4}{4104} - \frac{11k_5}{40}\right) \end{aligned} \quad [30]$$

With these k values the following expression is used to compute y_{n+1} .

$$y_{n+1} = y_{n+1} + \left(\frac{16k_1}{135} + \frac{6656k_3}{12825} + \frac{28561k_4}{56430} - \frac{9k_5}{50} + \frac{7k_6}{55} \right) \quad [31]$$

The k values are also used to compute an estimate of the error (as an absolute value) in the algorithm.

$$Error = \left| \frac{k_1}{60} - \frac{128k_3}{4275} + \frac{2197k_4}{75240} + \frac{k_5}{50} + \frac{2k_6}{55} \right| \quad [32]$$

The value of y_{n+1} is used as the value of the dependent variable at the end of the step. The error estimate is used to compute a new step size, based on the desired error (also an absolute value). Note this is an absolute error with the same dimensions as the dependent variable y , not a relative error

$$h_{new} = 0.84h \left(\frac{\text{Desired Error}}{\text{Error}} \right)^{1/4} \quad [33]$$

The basic MATLAB ODE solver, ODE45, uses a similar approach with a combination of a fourth-order and fifth-order Runge-Kutta algorithm known as the Dormand-Prince pair.¹

Multistep Methods

We have previously seen various methods, especially Runge-Kutta methods, which can obtain accurate solutions for the numerical integration of ordinary differential equations. However, these methods require a large amount of work per step. Several derivative evaluations are required at each step and this can increase the work if the derivatives are complex. The large number of derivative evaluations per step is required in these methods to obtain a high order truncation error. These methods have the advantage of being self-starting; the integration step from x_i to x_{i+1} does not require any information from grid points before $x - x_i$.

An alternative approach is to use information from past integration steps to derive a higher order expression for integration the differential equation. An obvious disadvantage of this approach is that the resulting methods will not be self-starting. Consequently, it will be necessary to provide some other method, such as a Runge-Kutta method, to start the integration with a multistep method.

Multistep methods are usually predictor-corrector methods. We have already seen an example of a predictor-corrector method in Huen's method, which was a modification of the Euler method. In that method we used a predicted value of y_{i+1} to compute an estimate of the derivative $f(x_{i+1}, y_{i+1})$. We then used that estimated derivative to compute a final (corrected) value of y_{i+1} . Multistep predictor-corrector methods proceed in a similar way, but they use information from previous steps to get higher order expressions for more accurate results. In addition, the difference between the predictor and the corrector can be used as an estimate of the error for step-size control.

A common example of predictor-corrector methods is the fourth-order Adams predictor-corrector method. The method proceeds from the current value y_i at x_i to obtain y_{i+1} at x_{i+1} in three steps. First, the most recent values of x_i and y_i are used to get a fresh estimate of the derivative, f_i .

$$f_i = f(x_i, y_i) \quad [34]$$

Next, this derivative value and the values of the derivatives at three previous steps are used to compute a predicted value of y at x_{i+1} .

$$y_{i+1}^{(p)} = y_i + \frac{h}{24} [-9f_{i-3} + 37f_{i-2} - 59f_{i-1} + 55f_i] \quad [35]$$

¹ <https://www.mathworks.com/help/matlab/ref/ode45.html?requestedDomain=www.mathworks.com>

The derivative at x_{i+1} , computed with this predicted y_{i+1} value, is then used to obtain the final (corrected) value of y_{i+1} by the following equation. Note the notation for the estimated derivative at x_{i+1} is $f(x_{i+1}, y_{i+1}^{(p)})$

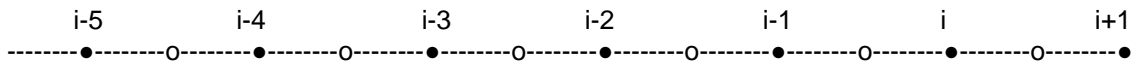
$$y_{i+1}^{(c)} = y_i + \frac{h}{24} [f_{i-2} - 5f_{i-1} + 19f_i + 9f(x_{i+1}, y_{i+1}^{(p)})] \quad [36]$$

The difference between the predictor and corrector is used to provide an estimate of the truncation error in the new calculation. The error estimate is given by the following equation.

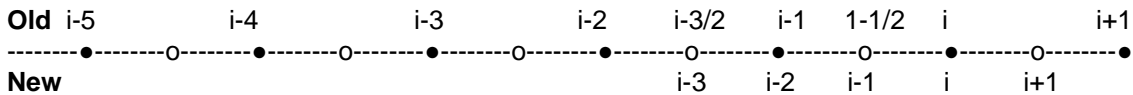
$$e_{i+1} = \frac{19}{270} [y_{i+1}^{(c)} - y_{i+1}^{(p)}] \quad [37]$$

This error estimate may be used for step size control as described below. Note that the sequence for the calculation starts, in equation [34], with a fresh evaluation of $f_{i+1} = f(x_{i+1}, y_{i+1})$. The derivative at x_{i+1} , used to calculate the corrected value in equation [36], is not used to start the new step in the calculation.

Since multistep methods are not self-starting, step-size adjustments for these methods are usually limited to halving or doubling the step size. When the step size is halved or doubled, it is necessary to obtain the required values of f_{i-1} , f_{i-2} , and f_{i-3} with the new step size. To see how this is done we first show the grid used for the step from i to $i+1$ using the existing step size. Here the black points (●) represent grid nodes, x_i and the hollow points (o) represent midpoints between the grid nodes used in the calculation.



If the error estimate found in equation [37] is too large, the value of y_{i+1} is not accepted and the step size is halved. The point i remains the same, but the new point $i+1$ is now midway between x_i and the previous value of x_{i+1} . The grid below shows the previous grid indexing on top and the new grid indexing, after the step size is halved, below the grid.



The values of f_{i-3} and f_{i-1} (in the new notation) that are required in equations [35] and [36] have to be found by interpolation. The following interpolations are consistent with the fourth order accuracy of the method. In both equations [38] and [39], the old grid numbering is used

$$f_{i-1/2} = \frac{1}{128} [-5f_{i-4} + 28f_{i-3} - 70f_{i-2} + 140f_{i-1} + 35f_i] \quad [38]$$

$$f_{i-3/2} = \frac{1}{64} [3f_{i-4} - 16f_{i-3} + 54f_{i-2} + 24f_{i-1} - f_i] \quad [39]$$

When the step size is doubled, the values of x_{i+1} and y_{i+1} are accepted and this point is shifted to the current point x_i, y_i from which we start our new integration step. The new grid indexing, after doubling the step size, is shown below the grid in the figure below.

$$\begin{aligned}
 y_{m,i+1} &= y_{m,i} + \frac{k_{1,m} + 2 * k_{2,m} + 2 * k_{3,m} + k_{4,m}}{6} & m = 1, \dots, N \\
 k_{1,m} &= hf_m(x_i, y_{1,i}, y_{2,i}, \dots, y_{N,i}) \\
 k_{2,m} &= hf_m(x_i + h/2, y_{1,i} + k_{1,1}/2, y_{2,i} + k_{1,2}/2, \dots, y_{N,i} + k_{1,N}/2) & [43] \\
 k_{3,m} &= hf_m(x_i + h/2, y_{1,i} + k_{2,1}/2, y_{2,i} + k_{2,2}/2, \dots, y_{N,i} + k_{2,N}/2) \\
 k_{4,m} &= hf_m(x_i + h, y_{1,i} + k_{3,1}, y_{2,i} + k_{3,2}, \dots, y_{N,i} + k_{3,N})
 \end{aligned}$$

In equation [43] all values of $k_{1,m}$ must be computed before any values of $k_{2,m}$ may be computed. Similarly, all $k_{2,m}$ must be computed before any $k_{3,m}$ are computed. The sample Visual Basic code shown below applies the fourth-order Runge-Kutta method to a system of equations. This code uses variables like $k1(m)$ to represent the increments like k_1 for the m^{th} variable in the fourth-order Runge-Kutta algorithm. The VBA code below shows how each step of the algorithm is applied to each equation before taking the next step. The routine `fsub`, which is used to compute all the derivatives at one time, is discussed further below.

```

For i = 1 to Nsteps                                'Do all steps from x0 to xmax
  Call fsub( x, y, f )                             'Compute all derivatives
  For m = 1 to Neqns                               'Loop over all equations
    k1(m) = h * f(m)                               'Compute all k1 values
    'Compute intermediate y values for calling fsub
    YInt(m) = y(m) + k1(m) / 2
  Next m
  'Repeat process in previous loop over all equations
  'to compute k2, k3, and k4 arrays
  Call fsub( x+h/2, YInt, f )
  For m = 1 to Neqns
    k2(m) = f * f(m)
    YInt(m) = y(m) + k2(m) / 2
  Next m
  Call fsub( x+h/2, YInt, f )
  For m = 1 to Neqns
    k3(m) = f * f(m)
    YInt (m) = y(m) + k3(m)
  Next m
  Call fsub( x+h, YInt, f )
  For m = 1 to Neqns
    y(m) = y(m) + ( k1(m) + 2 * k2(m) +2 * k3(m) _
      + h * f(m) ) / 6
  Next m
  X = x + h
Next i                                             'End of loop over all steps from x0 to xmax
    
```

In this code, the subroutine `fsub` is used to compute the derivative of all equations for input values of x and the y array. The values of the derivatives are returned in the f array. For example, consider the system of three differential equations shown below.

$$\begin{aligned}
 \frac{dy_1}{dx} &= -y_1 + \sqrt{y_2} + y_3 e^{2x} & y_1(0) &= 1 \\
 \frac{dy_2}{dx} &= -2y_1^2 & y_2(0) &= 1 \\
 \frac{dy_3}{dx} &= -3y_1 y_2 & y_3(0) &= 1
 \end{aligned}
 \tag{44}$$

The VBA fsub routine shown below calculates the necessary derivatives for this system of equations.

```

Sub fsub( x as Double, y() as Double, f() as Double )
    f(1) = -y(1) + Sqr(y(2)) + y(3) * Exp( 2 * x)
    f(2) = -2 * y(1)^2
    f(3) = -3 * y(1) * y(2)
End Sub
    
```

The same idea – that each step of the algorithm must be applied to each equation – is also true for multistep methods, like predictor-corrector methods. In predictor-corrector methods, a system of equations is solved by first applying the predictor to each equation. Then the derivative of each differential equation is evaluated at the end of the current step based on the predictor values for all variables. Finally, the corrector is applied to each equation to get the final results for the step.

Implicit methods

The methods we have discussed so far are called explicit methods. In these methods, the values at the new step of the independent variable are found in terms of values at previous steps. An alternative approach uses implicit methods where the values at the end of the new step are used as part of the algorithm for finding the results at the end of the step. This is best illustrated by the example of the **trapezoid method**. In this method, we start with the same Taylor series that we used for developing the Euler method.

$$y_{n+1} = y_n + f_n h + \frac{h^2 y_n''}{2} + O(h^3) \quad [45]$$

We next write a similar Taylor series giving the value of y_n as an expansion about y_{n+1} .

$$y_n = y_{n+1} - f_{n+1} h + \frac{h^2 y_{n+1}''}{2} + O(h^3) \quad [46]$$

Subtracting equation [46] from equation [45] gives the following result.

$$y_{n+1} - y_n = y_n - y_{n+1} + f_n h + f_{n+1} h + \frac{h^2 (y_n'' - y_{n+1}'')}{2} + O(h^3) \quad [47]$$

We can combine the y_n and y_{n+1} terms and introduce the Taylor series $y_{n+1}'' = y_n'' + h y_n''' + O(h^2)$ into equation [47]; when we do this all the higher-order terms are of order h^3 or higher which we can represent as $O(h^3)$.

$$\begin{aligned}
 y_{n+1} &= y_n + \frac{f_n h + f_{n+1} h}{2} + \frac{h^2 \{y_n'' - [y_n'' + h y_n''' + O(h^2)]\}}{4} + O(h^3) \\
 &= y_n + \frac{(f_n + f_{n+1}) h}{2} - \frac{h^2 [h y_n''']}{4} + O(h^3) = y_n + \frac{(f_n + f_{n+1}) h}{2} + O(h^3)
 \end{aligned} \quad [48]$$

The problem with this final result is that we have the value of the derivative expression, f , at the new step, which we do not know yet, in our equation. We can estimate this derivative from a multivariate Taylor series for the derivative, f .

$$f_{n+1} = f_n + \left(\frac{\partial f}{\partial x}\right)_n h + \left(\frac{\partial f}{\partial y}\right)_n (y_{n+1} - y_n) + O(h^2) \quad [49]$$

Substituting this expression for f_{n+1} into the final result in equation [48] gives

$$y_{n+1} = y_n + \frac{f_n h}{2} + h \left[f_n + \left(\frac{\partial f}{\partial x}\right)_n h + \left(\frac{\partial f}{\partial y}\right)_n (y_{n+1} - y_n) + O(h^2) \right] \quad [50]$$

Solving this equation for y_{n+1} gives the following result for the Trapezoid method.

$$y_{n+1} = y_n + \frac{hf_n + \left(\frac{\partial f}{\partial x}\right)_n \frac{h^2}{2}}{1 - \frac{h}{2} \left(\frac{\partial f}{\partial y}\right)_n} + O(h^3) \quad [51]$$

We can apply the trapezoid method to a simple example, whose analytical solution we know, $dy/dx = -ay$, with $y = y_0$ at $x = 0$. Here $f = -ay$ so that $\partial f/\partial x = 0$ and $\partial f/\partial y = -a$. For this example, the general equation in [51] becomes

$$y_{n+1} = y_n + \frac{hf_n + \left(\frac{\partial f}{\partial x}\right)_n \frac{h^2}{2}}{1 - \frac{h}{2} \left(\frac{\partial f}{\partial y}\right)_n} = y_n + \frac{-hay_n + 0}{1 - \frac{h}{2}(-a)} = \frac{y_n \left(1 + \frac{ha}{2}\right) - hay_n}{1 + \frac{ha}{2}} \quad [52]$$

A final rearrangement gives the following result for this example.

$$y_{n+1} = y_n \frac{2-ha}{2+ha} \quad [53]$$

We will use this result below when we consider the topic of stability.

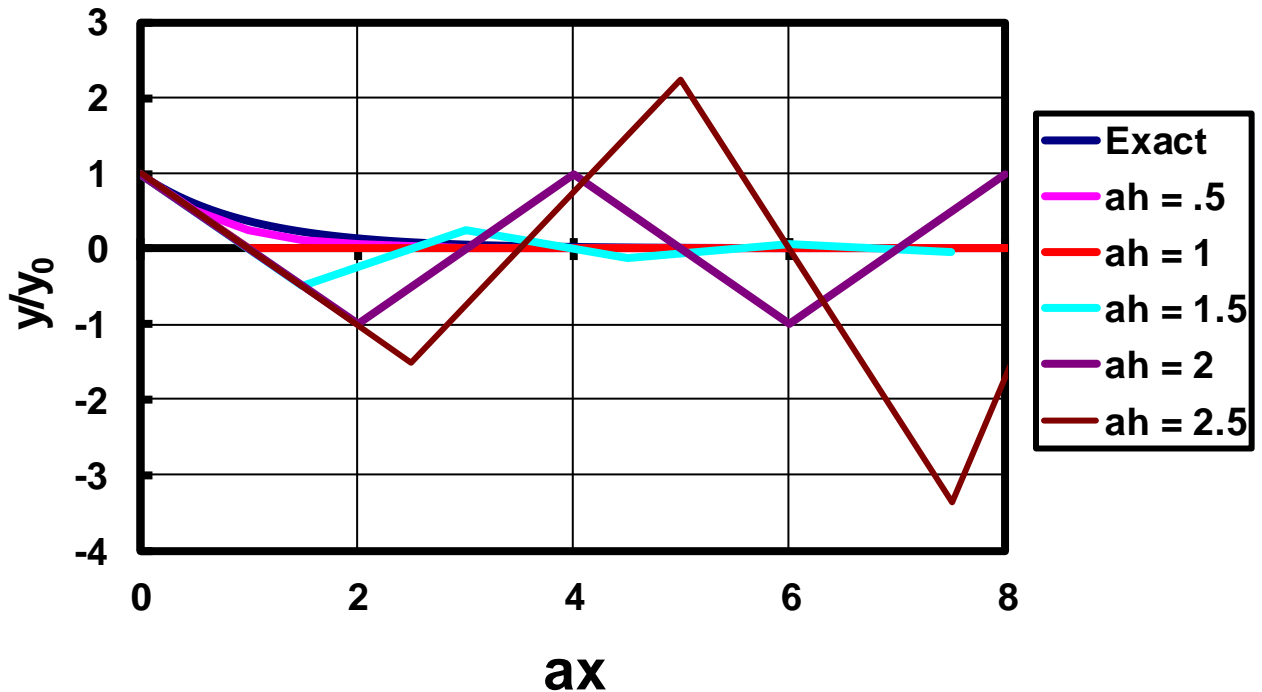
Stability of numerical solutions of ODEs

A numerical method is said to be stable if the error does not grow without bound. A method for which this is true regardless of the problem conditions is said to be absolutely stable. A method for which this is only true under certain conditions is said to be conditionally stable. We know that the solution of the differential equation $dy/dx = ay$ is $y = y_0 e^{at}$. This solution grows without bound as t increases unless a is negative. The stability of numerical methods is usually tested by being applied to the differential equation $dy/dx = -ay$, where a is a positive constant. The solution to this equation is $y = y_0 e^{-at}$. What do we get if we try to solve this equation using Euler's method, $y_{n+1} = y_n + hf_n$. In this case we have $f_n = -ay_n$ so $y_{n+1} = y_n + h(-ay_n) = y_n(1 - ah)$. We can solve this equation for different values of ah and obtain the results shown titled "Stability of the Euler Method on the next page.

The solutions for $ah = 0.5$ is close to the exact solution, but solutions for $ah = 1$ and greater and not physically realistic. The solution for $ah = 2$ is a series of straight lines, but the solutions remains bounded. It produces values of either $y = -1$ (at $x = 2, 6, 10, \dots$) or $y = 1$ (at $x = 4, 8, 12, \dots$). However, this solution is the limit of stability. For $ah = 2$, the solution, though grossly inaccurate, does not grow without bound. Beyond $ah = 2$ (as seen for $ah = 2.5$ in this plot) the solution grows without bound.

In our discussion of the trapezoid method we applied that algorithm to the test problem for stability just discussed: $dy/dx = -ay$. Equation [53] showed that $y_{n+1}/y_n = (2 - ha)/(2 + ha)$ when the trapezoid method was applied to $dy/dz = -ay$. The results of this numerical solution for different values of ah are shown in the second figure on the next page titled "Trapezoid Method Stability."

Stability of Euler Method

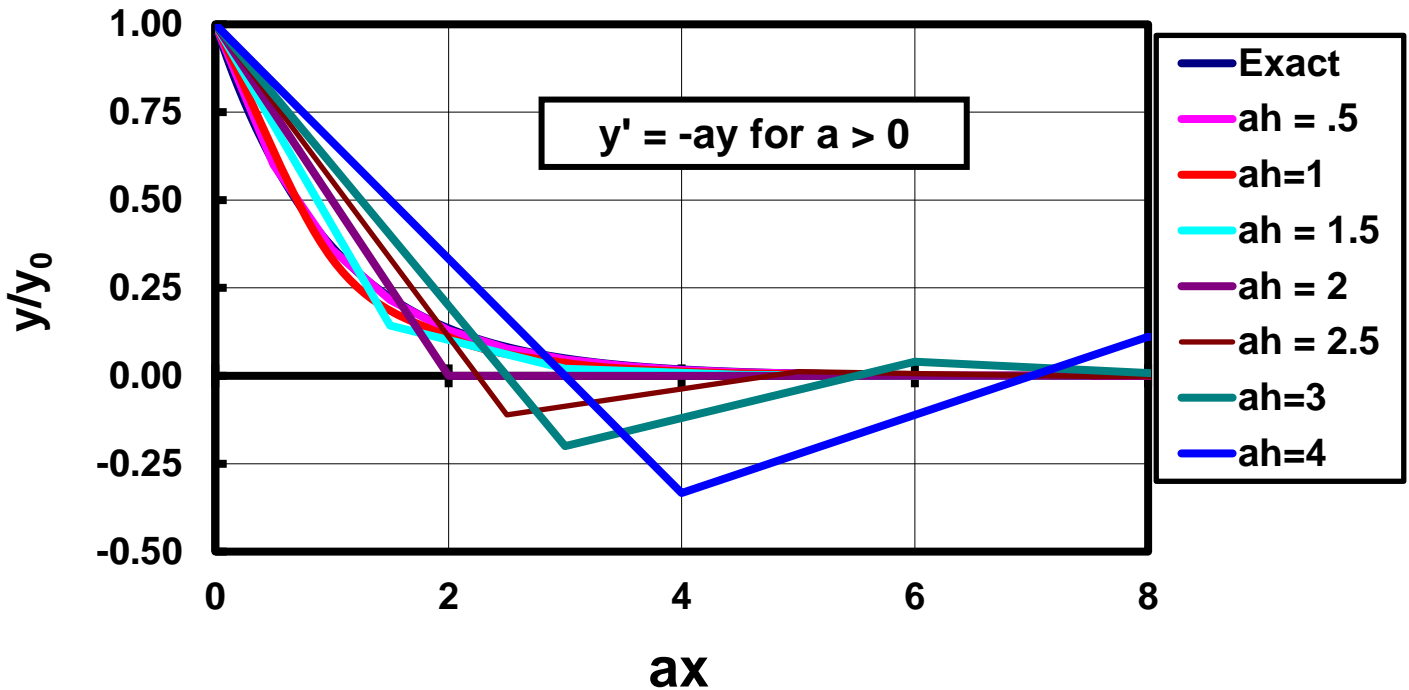


The "Trapezoid Method Stability" figure shows that the trapezoid method is stable, if not accurate, for any value of ah . Remember that stability only means that the solution will not grow without bound. On the scale of this plot, the solutions for $ah = 0.5$ and $ah = 1.0$ appear to be almost the same as the exact solution. Larger values of ah provide solutions where each individual step is so large that the solutions are a sequence of straight lines; for smaller values of ah , the solutions are also a sequence of straight lines, but each line in the sequence is so short that the solutions appear curved. Although, the solutions for larger values of ah are clearly incorrect, they are not unstable. They do not grow without bound. Hence, we consider the trapezoid method to be absolutely stable. However, stability, by itself, is not sufficient. We must also have accuracy. In fact, you might ask why do we even care about stability; isn't accuracy the only thing we have to be concerned about? The answer to this question comes in the section below dealing with stiff systems of equations.

Boundary-value problems

The numerical solutions of differential equations we have considered so far deal with the initial-value problem. For such problems, we have sufficient initial conditions at a single starting point to allow us to solve the numerical problem. However, it is also possible to have boundary conditions where that specify the values of the solution at two different points. A general second-order differential equation, $d^2y/dx^2 = g(x, y, dy/dx)$, could have set conditions such as $y = a$ at $x = 0$ and

Trapezoid Method Stability



$y = b$ at $x = L$. It is even possible to have more complex boundary conditions such as a $dy/dx + by = c$ at any specified value of the independent variable, x . There are two basic approaches to the solution of boundary-value problems: shoot-and-try and finite differences. We will consider each of these in turn.

To discuss the **shoot-and-try** method, we consider the solution of the following problem:

$$\frac{d^2 y}{dx^2} = g\left(x, y, \frac{dy}{dx}\right) \quad \text{with : } y = a \text{ at } x = 0 \text{ and } y = b \text{ at } x = L \quad [54]$$

We can convert this equation into a pair of first-order equations as we did for initial-value problems by defining a new variable $z = dy/dx$, and getting equations for dy/dx and dz/dx .

$$\frac{dy}{dx} = z \quad \text{and} \quad \frac{dz}{dx} = g(x, y, z) \quad \text{with : } y = a \text{ at } x = 0 \text{ and } y = b \text{ at } x = L \quad [55]$$

Here we have the problem that we do not have a value of the first derivative, z , at $x = 0$. We also have no ready way to use the boundary condition at $x = L$ to start the problem. The shoot-and-try method is a trial-and-error method in which we guess a value of the first derivative at $x = 0$, $y'(0) = z(0)$, then use any of the methods we have discussed for solving initial value problems to integrate the equations from $x = 0$ to $x = L$. Once we have completed this integration, we can compare the value we have just found for y at $x = L$, to the specified boundary condition, $y(L)$. We can use the difference between these two values to adjust our initial guess for the first derivative at $x = 0$. We can continue repeating this process until the difference between the computed and specified value of y at $x = L$ is less than some specified maximum error.

To do this process, we define the difference between the computed value at $x = L$, denoted as $y^{(m)}(L)$ as the result for the m^{th} trial, and the specified boundary condition, $y(L)$, as the error for trial m , $E^{(m)}$.

$$E^{(m)} = y^{(m)}(L) - y(L) \quad [56]$$

We can define an iterative process that assumes a linear relationship between two trial values of the initial slope, $z^{(m)}(0)$ and $z^{(m-1)}(0)$, and the corresponding errors in the boundary values at $z = L$, $E^{(m)}$ and $E^{(m-1)}$; we can use this linear relationship to find the value of the next guess for the initial slope $z^{(m+1)}(0)$.

$$z^{(m+1)}(0) = z^{(m)}(0) + \frac{z^{(m)}(0) - z^{(m-1)}(0)}{E^{(m)} - E^{(m-1)}} (E^{(m+1)} - E^{(m)}) \quad [57]$$

We want the error on the next iteration, $E^{(m+1)}$, to be zero. Setting $E^{(m+1)}$ equal to zero in [57] gives the following iteration equation for the new value of the initial slope, $z^{(m+1)}(0)$.

$$z^{(m+1)}(0) = z^{(m)}(0) - E^{(m)} \frac{z^{(m)}(0) - z^{(m-1)}(0)}{E^{(m)} - E^{(m-1)}} \quad [58]$$

Because this equation requires two previous trials we have to use other equations, like those in equation [59], to get the value of the initial slope for the first and second iterations.

$$z^{(1)}(0) = \frac{y(L) - y(0)}{L} \quad z^{(2)}(0) = \frac{2y(L) - y^{(1)}(L) - y(0)}{L} \quad [59]$$

As an example, consider the solution of the following problem.

$$\frac{d^2 y}{dx^2} + 16 \sin(y) = 0 \quad \text{with : } y = 1 \text{ at } x = 0 \text{ and } y = 0 \text{ at } x = L = 1 \quad [60]$$

To solve this, we first have to create a system of two first-order equations

$$\frac{dy}{dx} = z \quad \text{and} \quad \frac{dz}{dx} = -16 \sin(y) \quad [61]$$

We know that $y(0) = 1$ and we can use the first equation in [59] to get $z^{(1)} = (0 - 1)/1 = -1$. With this initial guess value of the slope at $x = 0$ a fourth-order Runge-Kutta calculation with $h = 0.005$ gives $y^{(1)}(L) = -3.8870$. Thus the error in this first calculation is $E^{(1)} = y^{(1)}(L) - y(L) = -3.8870 - 0 = -3.8870$. We can use the second part of equation [59] to get the next value of the slope at $x = 0$.

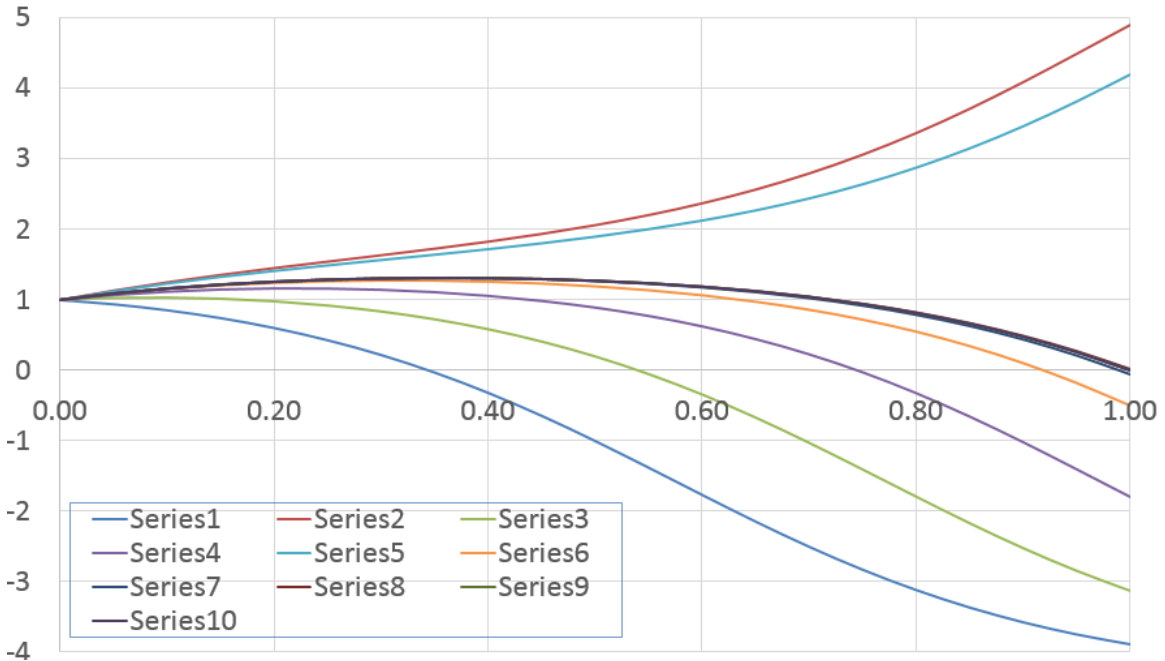
$$z^{(2)}(0) = \frac{2y(L) - y^{(1)}(L) - y(0)}{L} = \frac{2(0) - (-3.8870) - 1}{1} = 2.8870 \quad [62]$$

A second Runge-Kutta calculation gives $y^{(2)}(L) = 4.8974$ with a corresponding error of $4.8974 - 0 = 4.8974$; we can apply equation [58] to get the new value of $z(0)$ for the next iteration.

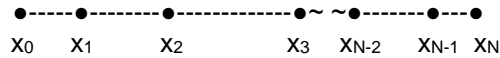
$$z^{(3)}(0) = z^{(2)}(0) - E^{(2)} \frac{z^{(2)}(0) - z^{(1)}(0)}{E^{(2)} - E^{(1)}} = 2.8870 - 4.8974 \frac{2.8870 - (-1)}{4.8894 - (-3.8870)} = 0.71993 \quad [63]$$

The results after ten iterations of the shoot-and-try give a value of the variable $y = 1.23 \times 10^{-8}$ at $x = 0$. This is taken as close enough to the stated boundary condition of $y = 0$ at $x = 1$. A plot of the results of each iteration of the fourth-order Runge-Kutta is shown in the figure below. We see that the iterations give solutions on either side of the eventual final solution labeled Series10 in the figure legend.

Shoot-and-Try Iterations



The **finite-difference method** is an alternative to the shoot-and-try method. In the finite difference method, one uses finite-difference expressions for derivatives to develop set of simultaneous algebraic equations that can be solved for values of the dependent variables in the differential equation at specific points on a grid. An example of such a grid is shown below.



The grid shown here is a non-uniform grid where the spacing between nodes is not the same. Such grids are used when there is an expected strong variation in certain areas of the grid and more nodes are placed in those strong-variation areas to get better resolution there. Uniform grids have a higher order error, and are generally preferred if there are not specific reason to do otherwise. Regardless of whether or not the grid is uniform, the basic approach is the same. A finite-difference expression for each derivative in the differential equation is used to convert the differential equation into a finite-difference equation at each node in the grid. The set of finite-difference equations is then solved to find the values at the nodes. It may also be necessary to use finite-difference expressions for the boundary conditions if they involve gradients.

The use of finite difference equations is best shown by example. Consider the following equation for one-dimensional heat transfer with a heat source that is proportional to the temperature, T , with a proportionality constant a^2 .

$$\frac{d^2T}{dx^2} + a^2T = 0 \tag{64}$$

We create a finite difference equivalent for this differential equation, at the location $x = x_i$, as shown below. In this equation, h is the step size, $\Delta x = (x_N - x_0)/N$.

$$\frac{T_{i+1} + T_{i-1} - 2T_i}{h^2} + O(h^2) + a^2 T_i = 0 \quad [65]$$

The $O(h^2)$ notation lets us know that the finite difference expression for the derivative is second-order accurate; we will drop this information in obtaining an equation to solve for the values of temperatures at all the nodes. Dropping this notation, multiplying by h^2 , and rearranging terms gives us the following finite difference equation.

$$T_{i+1} + T_{i-1} - (2 - h^2 a^2) T_i = 0 \quad [66]$$

Before discussing the solution of this system of algebraic equations we have to discuss the **boundary conditions**. In general, there are three possible kinds of boundary conditions. The first (and easiest) are: fixed value or Dirichlet boundary conditions; these specify values of the dependent variable at the boundary. They simply say that the values at the boundaries of the grid, $T = T_0$ and $T = T_N$ are known. The second set, called Neumann or gradient boundary conditions specify the gradient of the dependent variable at the boundary. In heat transfer the gradient of temperature at the boundary is used to specify a boundary heat flux. The third boundary condition, sometimes called a mixed boundary specifies a relation between the dependent variable and its gradient at the boundary.

All three types of boundary conditions can be expressed by the following equation:

$$a \frac{dT}{dx} + bT = c \quad [67]$$

In this general equation, the case of fixed temperature is determined by the following equations, using the specified temperatures at the left and right boundary, respectively.

$$a = 0, b = 1, T = T_0, c = T_{left} \quad a = 0, b = 1, T = T_N, c = T_{right} \quad [68]$$

In case of a specified gradient the following equations are used in the case of heat transfer, where k is the thermal conductivity and q_{left} and q_{right} are the specified heat fluxes. Note that second-order directional derivatives are used to keep the boundary conditions with the same (second-order) accuracy as the finite-difference equations.

$$-\frac{q_{left}}{k} = \frac{-T_2 + 4T_1 - 3T_0}{2h} \quad -\frac{q_{right}}{k} = \frac{T_{N-2} - 4T_{N-1} + 3T_N}{2h} \quad [69]$$

The third kind of boundary condition would be for a specified convective heat flux.

$$\frac{h_{left}(T_\infty - T_0)}{k} = \frac{-T_2 + 4T_1 - 3T_0}{2h} \quad \frac{h_{right}(T_\infty - T_N)}{k} = \frac{T_{N-2} - 4T_{N-1} + 3T_N}{2h} \quad [70]$$

In these boundary conditions, the values of h_{left} , h_{right} , and T_∞ are specified constants. (The values of T_∞ could be different on the two sides.)

To talk about **solving the system of finite-difference equations**, we will limit ourselves to fixed boundary conditions. We will assume that we are given $T_{left} = T_A = T_0$ and $T_{right} = T_B = T_N$. We will

rewrite our finite-difference equations from equation [66] using the shorthand expression $\alpha = h^2 a^2$, as shown below.

$$T_{i+1} + T_{i-1} - (2 - h^2 a^2) T_i = 0 \Rightarrow T_{i+1} + T_{i-1} - (2 - \alpha) T_i = 0 \quad [71]$$

With fixed boundary conditions, $T_0 = T_A$ and $T_B = T_N$, we will have a set of linear algebraic equations, written as a matrix equation in equation [7e], below, to solve. The solution is done using $N = 10$, $T_A = T_{\text{left}} = 0$, $T_B = T_{\text{right}} = 1$, $a = 2$, and $L = 1$.

For this differential equation, the exact solution is given by the following equation.

$$T = \frac{T_B - T_A \cos(aL)}{\sin(aL)} \sin(ax) + T_A \cos(ax) \quad [72]$$

We see that this set of equations forms a tridiagonal matrix where all terms that are not on the principal diagonal or the diagonals immediately above or below the principal diagonal are all zeros.

$$\begin{bmatrix} -2+\alpha & 1 & 0 & 0 & \cdots & 0 & 0 \\ 1 & -2+\alpha & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & -2+\alpha & 1 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \ddots & & \vdots & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & 0 & \cdots & -2+\alpha & 1 \\ 0 & 0 & 0 & 0 & \cdots & 1 & -2+\alpha \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ \vdots \\ \vdots \\ T_{N-2} \\ T_{N-1} \end{bmatrix} = \begin{bmatrix} -T_A \\ 0 \\ 0 \\ \vdots \\ \vdots \\ 0 \\ -T_B \end{bmatrix} \quad [73]$$

This is a particularly easy set of equations to solve. The process for doing this is discussed in the appendix. The solutions are shown in the table below for $a = 2$, $L = 1$, and $\alpha = aL = 2$.

Results of Finite-Difference Calculation				
i	x_i	T_i	Exact T_i	T_i Error
0	0.0	0	0	0
1	0.1	0.21918	0.21849	0.00070
2	0.2	0.42960	0.42826	0.00134
3	0.3	0.62284	0.62097	0.00187
4	0.4	0.79115	0.78891	0.00224
5	0.5	0.92783	0.92541	0.00242
6	0.6	1.02739	1.02501	0.00238
7	0.7	1.08585	1.08375	0.00211
8	0.8	1.10088	1.09928	0.00160
9	0.9	1.07188	1.07099	0.00089
10	1.0	1.0	0	0

We see that the errors are zero, as expected, at the initial and final nodes where the boundary conditions are specified. We also see that the closer we are to these boundaries, the smaller are the errors. The largest errors are at the nodes near the center where we are furthest from the boundaries. It is useful to have a single measure of the overall error. If we regard the error values as the components of a vector, we can use a vector norm as a measure of the overall error. The simplest error measure is the zero norm, which is the maximum error. For this

calculation that value is 0.00242. A more common measure is the root-mean-square (RMS) error defined by the following equation.

$$|\varepsilon|_{RMS} = \sqrt{\frac{1}{N} \sum_{i=1}^N \varepsilon_i^2} = \sqrt{\frac{1}{N} \sum_{i=1}^N (T_{exact} - T_{numerical})_i^2} \quad [74]$$

We can also compute the temperature gradients at the boundaries from the numerical solution and compare them to the gradients from the exact solution.

$$q_{x=0} = -k \left. \frac{dT}{dx} \right|_{x=0} = -ka \frac{T_B - T_A \cos(aL)}{\sin(aL)} \quad [75]$$

$$q_{x=L} = -k \left. \frac{dT}{dx} \right|_{x=L} = \frac{ka[T_A - T_B \cos(aL)]}{\sin(aL)} \quad [76]$$

For the results in the table, with a step size $h = 0.1$, the maximum error shown in the table is 2.42×10^{-3} and the calculated RMS error is 1.83×10^{-3} . The calculation was repeated with $h = .01$ resulting in a maximum error of 2.41×10^{-5} and an RMS error of 1.73×10^{-5} . We see that decreasing the step size by a factor of 10 reduced the two different error calculations by a factor of 100. This is consistent with the original second-order finite-difference expressions that were used to model the differential equation.

The heat flux at the boundaries can be computed by finite-difference expressions. The second order expressions for these calculations are shown below.

$$q_0 = -k \left. \frac{dT}{dx} \right|_{x=x_0} = -k \frac{-3T_0 + 4T_1 - T_2}{2h} \quad q_N = -k \left. \frac{dT}{dx} \right|_{x=x_N} = -k \frac{3T_N - 4T_{N-1} + T_{N-2}}{2h} \quad [77]$$

The results of these heat-flux calculations (expressed as q/k because k was not specified) are shown in the table below for both sets of calculations.

x	q_{exact}/k	h	q/k	Error
0	-2.1995	.1	-2.2357	.03618
0	-2.1995	.01	-2.1999	.00036
1	.9153	.1	.9332	.01786
1	.9153	.01	.9155	.00021

We see that the exact value of q/k does not change as we change the grid size. Because the differential equation has a heat source term, the $-a^2T$ term implies a heat gain in the region, regardless of the value of a – there is a heat loss on both sides of the one-dimensional region in the calculations. The heat loss at $x = 0$ is larger in magnitude than the heat loss at $x = 1$ because of the larger temperature gradient at that point. We see that the numerical result for the gradient calculation also displays a second order error because the error decreases by a factor of 100 when the step size decreases by a factor of 10.

This simple example of a finite-difference calculation is meant to illustrate the process that is used. More accurate difference methods can be used and more complex boundary conditions can be considered.

Stiff systems of equations

A stiff system of differential equations is a system of differential equations whose solutions have widely different exponential terms. The name arose from structures calculations during the 1940s

where this phenomenon was first observed. To see how this can occur, consider the following differential equation.

$$\frac{d^2 y}{dt^2} + 2 \frac{dy}{dt} + y = 9801e^{-t} \quad \text{with : } y = 101 \text{ and } y' = -200 \text{ at } t_0 = 0 \quad [78]$$

You should be able to show that the following equation satisfies the differential equation and the initial conditions.

$$y = 100e^{-t} + e^{-100t} \quad [79]$$

In the previous section on stability we saw that differential equations whose solutions had exponential terms like $\exp(-at)$ required low values of ah for stability in numerical calculations. How does this work in a solution like that in equation [79] where there are two exponential terms in the solution? After some thought you should realize that if there are two exponential terms, the one with the largest value of a will determine the stability. A numerical solution of equation [78], which has the analytical solution shown in equation [79], it is the value of $a = 100$ that will set the stability limit on ah , not the value of $a = 1$. However, the contribution of the $\exp(-100t)$ term to the solution will soon fade to zero. [At $t = 0.1$, the $\exp(-100t)$ term equals 4.54×10^{-5} , and the $100\exp(-t)$ term equals 90.5; at $t = 1$, the $\exp(-100t)$ term equals 3.72×10^{-44} and the $100\exp(-t)$ term equals 36.8.] In the numerical solution to a problem with behavior like this, stability would require a very small step size to keep the $\exp(-100t)$ term stable, but after a small initial time, this term would not contribute to the overall solution. So here is the problem of stiff systems: parts of the solution, which are exponential terms with large negative exponents, will rapidly go to zero. In principle, these could be ignored and still provide an accurate solution, but, in practice, the demands of stability force these terms to control the step size to a very small value. In other words, the insignificant terms in the solution are governing the required step size for stability!

Gear's Method

An early procedure for solution of stiff equations was due to C. W. Gear. This method uses a detailed computer code which adjusts the order of the algorithm and the step size based on a test of the results during the calculation. It is an implicit method. The basic formula for the method is given by the following equation.

$$y_{n+1} = \gamma \left(\beta h f_{n+1} + \sum_{j=0}^k \alpha_{-j} y_{n-j} \right) \quad [80]$$

The coefficients in this equation depend on the order, k , of the equation that is being used. This order is adjusted during the calculation to give more accurate solutions in regions where the variation is steeper. The coefficients used in this equation are shown in the table on the next page.

The presence of the derivative at the end of the time step, f_{n+1} , in equation [80] makes the Gear algorithm an implicit one. Each step has to be solved iteratively to get the derivative term to match the new value(s) of y_{n+1} .

Coefficients in Gear Equation for Different Orders, k								
k	γ	β	α_0	α_{-1}	α_{-2}	α_{-3}	α_{-4}	α_{-5}
1	1	1	1					
2	1/3	2	4	-1				
3	1/11	6	18	-9	2			
4	1/25	12	48	-36	16	-3		
5	1/137	60	300	-300	200	-75	12	
6	1/147	60	360	-450	400	-225	72	-10

Appendix – Solving Tridiagonal Matrix Equations

A general system of tridiagonal matrix equations may be written in the following format.

$$A_i x_{i-1} + B_i x_i + C_i x_{i+1} = D_i \tag{A-1}$$

This provides not only a representation of the general tridiagonal equation; it also suggests a data structure for storing the array on a computer. Each diagonal of the matrix is stored as a one-dimensional array. For the most general solution of 100 simultaneous equations, we would have $100^2 = 10,000$ coefficients and 100 right-hand-side terms. In the tridiagonal matrix formulation with 100 unknowns, we would have only 400 nonzero terms considering both the coefficients and the right-hand side terms.

In order to maintain the tridiagonal structure, the first and last equation in the set will have only two terms. These equations may be written as shown below. These equations show that neither A_0 nor C_N are defined.

$$B_0 x_0 + C_0 x_1 = D_0 \tag{A-2}$$

$$A_N x_{N-1} + B_N x_N = D_N \tag{A-3}$$

The set of equations represented by equation [73] (on page 23) is particularly simple. In that set of equations, all $A_i = C_i = 1$; all $B_i = (-2 + \alpha)$, and all $D_i = 0$, except for the first and last values. In the general form that we are solving here the coefficients in one equation may all be different, and a given coefficient, say A, may have different values in different equations. To start the solution process, we solve equation [A-2] for x_0 in terms of x_1 as follows.

$$x_0 = [-C_0 / B_0] x_1 + [D_0 / B_0] \tag{A-4}$$

The solution to the tridiagonal matrix set of equations, known as the Thomas algorithm, seeks to find an equation like [A-4] for each other unknown in the set. The general equation that we are seeking will find the value of x_i in terms of x_{i+1} in the general form shown below.

$$x_i = E_i x_{i+1} + F_i \tag{A-5}$$

By comparing equations [A-4] and [A-5], we see that we already know $E_0 = -C_0 / B_0$ and $F_0 = D_0 / B_0$. To get equations for subsequent values of E_i and F_i , we rewrite equation [A-5] to be solved for $x_{i-1} = E_{i-1} x_i + F_{i-1}$, and substitute this into the general equation, [A-1].

$$A_i [E_{i-1} x_i + F_{i-1}] + B_i x_i + C_i x_{i+1} = D_i \tag{A-6}$$

We can rearrange this equation to solve for x_i .

$$x_i = \frac{-C_i}{B_i + A_i E_{i-1}} x_{i+1} + \frac{D_i - A_i F_{i-1}}{B_i + A_i E_{i-1}} \quad [\text{A-7}]$$

By comparing equations [A-5] and [A-6], we see that the general expressions for E_i and F_i are given in terms of the already know equations coefficients, A_i , B_i , C_i ; and D_i , and previously computed values of E and F .

$$E_i = \frac{-C_i}{B_i + A_i E_{i-1}} \quad \text{and} \quad F_i = \frac{D_i - A_i F_{i-1}}{B_i + A_i E_{i-1}} \quad [\text{A-8}]$$

We have to get an equation for the final point, x_N . We will not calculate x_N , until we have completed the process of computing the values of E and F up through equation $N-1$. At that point, we will know the coefficients the following equation:

$$x_{N-1} = E_{N-1} x_N + F_{N-1} \quad [\text{A-9}]$$

We will also know the coefficients A_N , B_N , and D_N in the original matrix equation, given by [A-3]. We can solve equations [A-3] and [A-9] simultaneously for x_N .

$$x_N = \frac{D_N - A_N F_{N-1}}{B_N + A_N E_{N-1}} \quad [\text{A-10}]$$

We see that the right-hand side of this equation is the same as the right-hand side of the equation for F_N in [A-8].

The Thomas algorithm is a simple one to implement in a computer program. The code below provides a C++ function to implement the calculations shown in this appendix. This function uses separate arrays for E_i , F_i , and x_i . However, it is possible to save computer storage by overwriting the input arrays with the results for E_i , F_i , and x_i . This is possible because the input data are not required for the TDMA algorithm after their initial use in the computation of E_i and F_i .

```
void tdma( double *a, double *b, double *c, double *d,
           double *x, int N )
{
    // Generic subroutine to solve a set of simultaneous linear equations that
    // form a tridiagonal matrix. The general form of the equations to be solved is
    // a[i] * x[i-1] + b[i] * x[i] + c[i] * x[i+1] = d[i]
    // The index, i, runs from 0 to N. The values of a[0] and c[N] are not defined
    // The user must define the one-dimensional arrays a, b, c, and d.
    // The user passes these arrays and a value for N to this function.
    // The function returns the resulting values of x to the user.
    // All arrays are declared as pointers in the calling program to allow
    // allocation of the arrays at run time.

    double *e = new double[N+1]; // Allocate storage for working arrays
    double *f = new double[N+1];
    e[0] = -c[0]/b[0]; // Get values of e and f for initial node
    f[0] = d[0]/b[0];
    for ( int i = 1; i < N; i++) // Get values of e and f for nodes 1 to N-1
    {
        e[i] = -c[i] / ( b[i] + a[i] * e[i-1] );
        f[i] = (d[i] - a[i] * f[i-1]) / ( b[i] + a[i] * e[i-1] );
    }
    // All e and f values now found now. Start with calculation of x[N].
    // Then get remaining values by back substitution in a for loop.
    x[N] = (d[N] - a[N] * f[N-1]) / ( b[N] + a[N] * e[N-1] );

    for ( i = N-1; i >= 0; i-- )
    {
        x[i] = e[i] * x[i+1] + f[i];
    }
    delete[] e; // Free memory used for allocated arrays
    delete[] f;
}

```