10
Numerical Solutions of PDEs

There's no sense in being precise when you don't even know what you're talking about.- John von Neumann (1903-1957)

Most of the book has dealt with finding exact solutions to some generic problems. However, most problems of interest cannot be solved exactly. The heat, wave, and Laplace equations are linear partial differential equations and can be solved using separation of variables in geometries in which the Laplacian is separable. However, once we introduce nonlinearities, or complicated non-constant coefficients intro the equations, some of these methods do not work. Even when separation of variables or the method of eigenfunction expansions gave us exact results, the computation of the resulting series had to be done on a computer and inevitably one could only use a finite number of terms of the expansion. So, therefore, it is sometimes useful to be able to solve differential equations numerically.

In this chapter we will introduce the idea of numerical solutions of partial differential equations. However, we will first begin with a discussion of the solution of ordinary differential equations in order to get a feel for some common problems in the solution of differential equations and the notion of convergence rates of numerical schemes. Then, we turn to the finite difference method and the ideas of stability. Other common approaches may be added later.

10.1 Ordinary Differential Equations

10.1.1 Euler's Method

In this section we will look at the simplest method for solving first order equations, Euler's Method. While it is not the most efficient method, it does provide us with a picture of how one proceeds and can be improved by introducing better techniques, which are typically covered in a numerical analysis text.

Let's consider the class of first order initial value problems of the form

$$\frac{dy}{dx} = f(x,y), \quad y(x_0) = y_0. \tag{10.1}$$

We are interested in finding the solution $y(x)$ of this equation which passes through the initial point (x_0, y_0) in the xy-plane for values of x in the interval $[a, b]$, where $a = x_0$. We will seek approximations of the solution at N points, labeled x_n for $n = 1, \ldots, N$. For equally spaced points we have $\Delta x = x_1 - x_0 = x_2 - x_1$, etc. We can write these as

$$x_n = x_0 + n\Delta x.$$

In Figure 10.1 we show three such points on the x-axis.

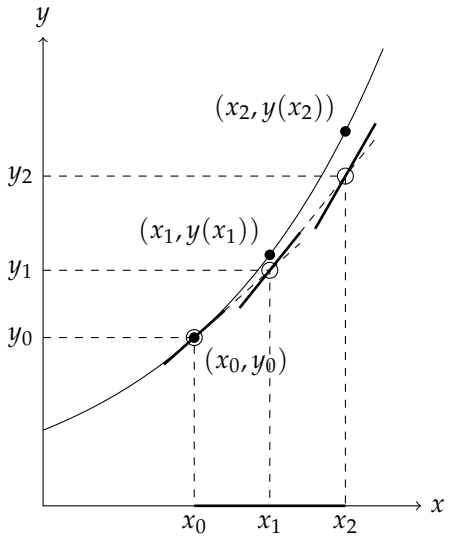

Figure 10.1: The basics of Euler's Method are shown. An interval of the x axis is broken into N subintervals. The approximations to the solutions are found using the slope of the tangent to the solution, given by $f(x, y)$. Knowing previous approximations at (x_{n-1}, y_{n-1}), one can determine the next approximation, y_n.

The first step of Euler's Method is to use the initial condition. We represent this as a point on the solution curve, $(x_0, y(x_0)) = (x_0, y_0)$, as shown in Figure 10.1. The next step is to develop a method for obtaining approximations to the solution for the other x_n's.

We first note that the differential equation gives the slope of the tangent line at $(x, y(x))$ of the solution curve since the slope is the derivative, $y'(x)'$ From the differential equation the slope is $f(x, y(x))$. Referring to Figure 10.1, we see the tangent line drawn at (x_0, y_0). We look now at $x = x_1$. The vertical line $x = x_1$ intersects both the solution curve and the tangent line passing through (x_0, y_0). This is shown by a heavy dashed line.

While we do not know the solution at $x = x_1$, we can determine the tangent line and find the intersection point that it makes with the vertical. As seen in the figure, this intersection point is in theory close to the point on the solution curve. So, we will designate y_1 as the approximation of the solution $y(x_1)$. We just need to determine y_1.

The idea is simple. We approximate the derivative in the differential equation by its difference quotient:

$$\frac{dy}{dx} \approx \frac{y_1 - y_0}{x_1 - x_0} = \frac{y_1 - y_0}{\Delta x}. \tag{10.2}$$

Since the slope of the tangent to the curve at (x_0, y_0) is $y'(x_0) = f(x_0, y_0)$,

we can write

$$\frac{y_1 - y_0}{\Delta x} \approx f(x_0, y_0). \tag{10.3}$$

Solving this equation for y_1, we obtain

$$y_1 = y_0 + \Delta x f(x_0, y_0). \tag{10.4}$$

This gives y_1 in terms of quantities that we know.

We now proceed to approximate $y(x_2)$. Referring to Figure 10.1, we see that this can be done by using the slope of the solution curve at (x_1, y_1). The corresponding tangent line is shown passing though (x_1, y_1) and we can then get the value of y_2 from the intersection of the tangent line with a vertical line, $x = x_2$. Following the previous arguments, we find that

$$y_2 = y_1 + \Delta x f(x_1, y_1). \tag{10.5}$$

Continuing this procedure for all x_n, $n = 1, \ldots N$, we arrive at the following scheme for determining a numerical solution to the initial value problem:

$$\begin{aligned} y_0 &= y(x_0), \\ y_n &= y_{n-1} + \Delta x f(x_{n-1}, y_{n-1}), \quad n = 1, \ldots, N. \end{aligned} \tag{10.6}$$

This is referred to as Euler's Method.

Example 10.1. *Use Euler's Method to solve the initial value problem* $\frac{dy}{dx} = x + y, \quad y(0) = 1$ *and obtain an approximation for* $y(1)$.

First, we will do this by hand. We break up the interval $[0, 1]$*, since we want the solution at* $x = 1$ *and the initial value is at* $x = 0$*. Let* $\Delta x = 0.50$*. Then,* $x_0 = 0$*,* $x_1 = 0.5$ *and* $x_2 = 1.0$*. Note that there are* $N = \frac{b-a}{\Delta x} = 2$ *subintervals and thus three points.*

We next carry out Euler's Method systematically by setting up a table for the needed values. Such a table is shown in Table 10.1. Note how the table is set up. There is a column for each x_n *and* y_n*. The first row is the initial condition. We also made use of the function* $f(x, y)$ *in computing the* y_n*'s from (10.6). This sometimes makes the computation easier. As a result, we find that the desired approximation is given as* $y_2 = 2.5$*.*

n	x_n	$y_n = y_{n-1} + \Delta x f(x_{n-1}, y_{n-1} = 0.5x_{n-1} + 1.5y_{n-1}$
0	0	1
1	0.5	$0.5(0) + 1.5(1.0) = 1.5$
2	1.0	$0.5(0.5) + 1.5(1.5) = 2.5$

Table 10.1: Application of Euler's Method for $y' = x + y$, $y(0) = 1$ and $\Delta x = 0.5$.

Is this a good result? Well, we could make the spatial increments smaller. Let's repeat the procedure for $\Delta x = 0.2$*, or* $N = 5$*. The results are in Table 10.2.*

Now we see that the approximation is $y_1 = 2.97664$*. So, it looks like the value is near 3, but we cannot say much more. Decreasing* Δx *more shows that we are beginning to converge to a solution. We see this in Table 10.3.*

Table 10.2: Application of Euler's Method for $y' = x + y$, $y(0) = 1$ and $\Delta x = 0.2$.

n	x_n	$y_n = 0.2x_{n-1} + 1.2y_{n-1}$
0	0	1
1	0.2	$0.2(0) + 1.2(1.0) = 1.2$
2	0.4	$0.2(0.2) + 1.2(1.2) = 1.48$
3	0.6	$0.2(0.4) + 1.2(1.48) = 1.856$
4	0.8	$0.2(0.6) + 1.2(1.856) = 2.3472$
5	1.0	$0.2(0.8) + 1.2(2.3472) = 2.97664$

Table 10.3: Results of Euler's Method for $y' = x + y$, $y(0) = 1$ and varying Δx

Δx	$y_N \approx y(1)$
0.5	2.5
0.2	2.97664
0.1	3.187484920
0.01	3.409627659
0.001	3.433847864
0.0001	3.436291854

Of course, these values were not done by hand. The last computation would have taken 1000 lines in the table, or at least 40 pages! One could use a computer to do this. A simple code in Maple would look like the following:

```
> restart:
> f:=(x,y)->y+x;
> a:=0:  b:=1:  N:=100:  h:=(b-a)/N;
> x[0]:=0: y[0]:=1:
  for i from 1 to N do
  y[i]:=y[i-1]+h*f(x[i-1],y[i-1]):
  x[i]:=x[0]+h*(i):
  od:
  evalf(y[N]);
```

In this case we could simply use the exact solution. The exact solution is easily found as

$$y(x) = 2e^x - x - 1.$$

(The reader can verify this.) So, the value we are seeking is

$$y(1) = 2e - 2 = 3.4365636\ldots.$$

Thus, even the last numerical solution was off by about 0.00027.

Adding a few extra lines for plotting, we can visually see how well the approximations compare to the exact solution. The Maple code for doing such a plot is given below.

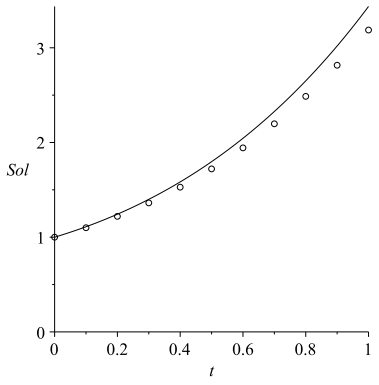

Figure 10.2: A comparison of the results Euler's Method to the exact solution for $y' = x + y$, $y(0) = 1$ and $N = 10$.

```
> with(plots):
> Data:=[seq([x[i],y[i]],i=0..N)]:
> P1:=pointplot(Data,symbol=DIAMOND):
> Sol:=t->-t-1+2*exp(t);
```

```
> P2:=plot(Sol(t),t=a..b,Sol=0..Sol(b)):
> display({P1,P2});
```

We show in Figures 10.2-10.3 the results for $N = 10$ and $N = 100$. In Figure 10.2 we can see how quickly the numerical solution diverges from the exact solution. In Figure 10.3 we can see that visually the solutions agree, but we note that from Table 10.3 that for $\Delta x = 0.01$, the solution is still off in the second decimal place with a relative error of about 0.8%.

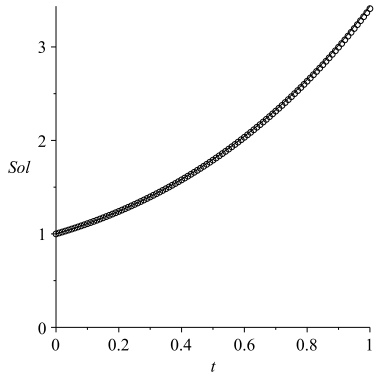

Figure 10.3: A comparison of the results Euler's Method to the exact solution for $y' = x + y$, $y(0) = 1$ and $N = 100$.

WHY WOULD WE USE A NUMERICAL METHOD when we have the exact solution? Exact solutions can serve as test cases for our methods. We can make sure our code works before applying them to problems whose solution is not known.

There are many other methods for solving first order equations. One commonly used method is the fourth order Runge-Kutta method. This method has smaller errors at each step as compared to Euler's Method. It is well suited for programming and comes built-in in many packages like Maple and MATLAB. Typically, it is set up to handle systems of first order equations.

In fact, it is well known that nth order equations can be written as a system of n first order equations. Consider the simple second order equation

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

This is a larger class of equations than the second order constant coefficient equation. We can turn this into a system of two first order differential equations by letting $u = y$ and $v = y' = u'$. Then, $v' = y'' = f(x, u)$. So, we have the first order system

$$\begin{aligned} u' &= v, \\ v' &= f(x, u). \end{aligned} \tag{10.7}$$

We will not go further into the Runge-Kutta Method here. You can find more about it in a numerical analysis text. However, we will see that systems of differential equations do arise naturally in physics. Such systems are often coupled equations and lead to interesting behaviors.

10.1.2 *Higher Order Taylor Methods*

EULER'S METHOD FOR SOLVING DIFFERENTIAL EQUATIONS is easy to understand but is not efficient in the sense that it is what is called a first order method. The error at each step, the local truncation error, is of order Δx, for x the independent variable. The accumulation of the local truncation errors results in what is called the global error. In order to generalize Euler's Method, we need to rederive it. Also, since these methods are typically used for initial value problems, we will cast the problem to be solved as

$$\frac{dy}{dt} = f(t, y), \quad y(a) = y_0, \quad t \in [a, b]. \tag{10.8}$$

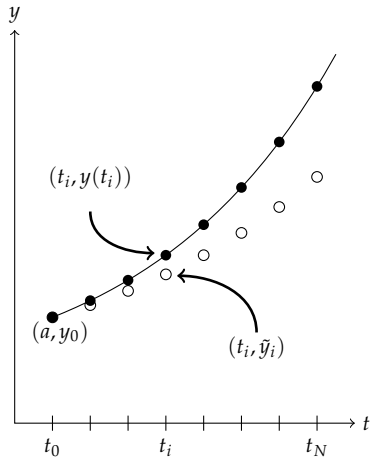

Figure 10.4: The interval $[a, b]$ is divided into N equally spaced subintervals. The exact solution $y(t_i)$ is shown with the numerical solution, $\tilde{y}_i$ with $t_i = a + ih$, $i = 0, 1, \ldots, N$.

The first step towards obtaining a numerical approximation to the solution of this problem is to divide the t-interval, $[a, b]$, into N subintervals,

$$t_i = a + ih, \quad i = 0, 1, \ldots, N, \quad t_0 = a, \quad t_N = b,$$

where

$$h = \frac{b-a}{N}.$$

We then seek the numerical solutions

$$\tilde{y}_i \approx y(t_i), \quad i = 1, 2, \ldots, N,$$

with $\tilde{y}_0 = y(t_0) = y_0$. Figure 10.4 graphically shows how these quantities are related.

Euler's Method can be derived using the Taylor series expansion of of the solution $y(t_i + h)$ about $t = t_i$ for $i = 1, 2, \ldots, N$. This is given by

$$\begin{aligned} y(t_{i+1}) &= y(t_i + h) \\ &= y(t_i) + y'(t_i)h + \frac{h^2}{2}y''(\xi_i), \quad \xi_i \in (t_i, t_{i+1}). \end{aligned} \tag{10.9}$$

Here the term $\frac{h^2}{2}y''(\xi_i)$ captures all of the higher order terms and represents the error made using a linear approximation to $y(t_i + h)$.

Dropping the remainder term, noting that $y'(t) = f(t, y)$, and defining the resulting numerical approximations by $\tilde{y}_i \approx y(t_i)$, we have

$$\begin{aligned} \tilde{y}_{i+1} &= \tilde{y}_i + hf(t_i, \tilde{y}_i), \quad i = 0, 1, \ldots, N-1, \\ \tilde{y}_0 &= y(a) = y_0. \end{aligned} \tag{10.10}$$

This is Euler's Method.

Euler's Method is not used in practice since the error is of order h. However, it is simple enough for understanding the idea of solving differential equations numerically. Also, it is easy to study the numerical error, which we will show next.

The error that results for a single step of the method is called the local truncation error, which is defined by

$$\tau_{i+1}(h) = \frac{y(t_{i+1}) - \tilde{y}_i}{h} - f(t_i, y_i).$$

A simple computation gives

$$\tau_{i+1}(h) = \frac{h}{2}y''(\xi_i), \quad \xi_i \in (t_i, t_{i+1}).$$

Since the local truncation error is of order h, this scheme is said to be of order one. More generally, for a numerical scheme of the form

$$\begin{aligned} \tilde{y}_{i+1} &= \tilde{y}_i + hF(t_i, \tilde{y}_i), \quad i = 0, 1, \ldots, N-1, \\ \tilde{y}_0 &= y(a) = y_0, \end{aligned} \tag{10.11}$$

The local truncation error.

the local truncation error is defined by

$$\tau_{i+1}(h) = \frac{y(t_{i+1}) - \tilde{y}_i}{h} - F(t_i, y_i).$$

The accumulation of these errors leads to the global error. In fact, one can show that if f is continuous, satisfies the Lipschitz condition,

$$|f(t, y_2) - f(t, y_1)| \leq L|y_2 - y_1|$$

for a particular domain $D \subset R^2$, and

$$|y''(t)| \leq M, \quad t \in [a, b],$$

then

$$|y(t_i) - \tilde{y}_|| \leq \frac{hM}{2L}\left(e^{L(t_i - a)} - 1\right), \quad i = 0, 1, \ldots, N.$$

Furthermore, if one introduces round-off errors, bounded by δ, in both the initial condition and at each step, the global error is modified as

$$|y(t_i) - \tilde{y}_|| \leq \frac{1}{L}\left(\frac{hM}{2} + \frac{\delta}{h}\right)\left(e^{L(t_i - a)} - 1\right) + |\delta_0|e^{L(t_i - a)}, \quad i = 0, 1, \ldots, N.$$

Then for small enough steps h, there is a point when the round-off error will dominate the error. [See Burden and Faires, *Numerical Analysis* for the details.]

Can we improve upon Euler's Method? The natural next step towards finding a better scheme would be to keep more terms in the Taylor series expansion. This leads to Taylor series methods of order n.

Taylor series methods of order n take the form

$$\begin{aligned} \tilde{y}_{i+1} &= \tilde{y}_i + hT^{(n)}(t_i, \tilde{y}_i), \quad i = 0, 1, \ldots, N-1, \\ \tilde{y}_0 &= y_0, \end{aligned} \tag{10.12}$$

where we have defined

$$T^{(n)}(t, y) = y'(t) + \frac{h}{2}y''(t) + \cdots + \frac{h^{(n-1)}}{n!}y^{(n)}(t).$$

However, since $y'(t) = f(t, y)$, we can write

$$T^{(n)}(t, y) = f(t, y) + \frac{h}{2}f'(t, y) + \cdots + \frac{h^{(n-1)}}{n!}f^{(n-1)}(t, y).$$

We note that for $n = 1$, we retrieve Euler's Method as a special case. We demonstrate a third order Taylor's Method in the next example.

Example 10.2. *Apply the third order Taylor's Method to*

$$\frac{dy}{dt} = t + y, \quad y(0) = 1$$

and obtain an approximation for $y(1)$ for $h = 0.1$.

The third order Taylor's Method takes the form

$$\begin{aligned} \tilde{y}_{i+1} &= \tilde{y}_i + hT^{(3)}(t_i, \tilde{y}_i), \quad i = 0, 1, \ldots, N-1, \\ \tilde{y}_0 &= y_0, \end{aligned} \tag{10.13}$$

where

$$T^{(3)}(t,y) = f(t,y) + \frac{h}{2}f'(t,y) + \frac{h^2}{3!}f''(t,y)$$

and $f(t,y) = t + y(t)$.

In order to set up the scheme, we need the first and second derivative of $f(t,y)$:

$$\begin{aligned} f'(t,y) &= \frac{d}{dt}(t+y) \\ &= 1 + y' \\ &= 1 + t + y \end{aligned} \tag{10.14}$$

$$\begin{aligned} f''(t,y) &= \frac{d}{dt}(1+t+y) \\ &= 1 + y' \\ &= 1 + t + y \end{aligned} \tag{10.15}$$

Inserting these expressions into the scheme, we have

$$\begin{aligned} \tilde{y}_{i+1} &= \tilde{y}_i + h\left[(t_i + y_i) + \frac{h}{2}(1 + t_i + y_i) + \frac{h^2}{3!}(1 + t_i + y_i)\right], \\ &= \tilde{y}_i + h(t_i + y_i) + h^2(\frac{1}{2} + \frac{h}{6})(1 + t_i + y_i), \\ \tilde{y}_0 &= y_0, \end{aligned} \tag{10.16}$$

for $i = 0, 1, \ldots, N-1$.

In Figure 10.2 we show the results comparing Euler's Method, the 3rd Order Taylor's Method, and the exact solution for $N = 10$. *In Table 10.4 we provide are the numerical values. The relative error in Euler's method is about 7% and that of the 3rd Order Taylor's Method is about 0.006%. Thus, the 3rd Order Taylor's Method is significantly better than Euler's Method.*

Table 10.4: Numerical values for Euler's Method, 3rd Order Taylor's Method, and exact solution for solving Example 10.2 with $N = 10$..

Euler	Taylor	Exact
1.0000	1.0000	1.0000
1.1000	1.1103	1.1103
1.2200	1.2428	1.2428
1.3620	1.3997	1.3997
1.5282	1.5836	1.5836
1.7210	1.7974	1.7974
1.9431	2.0442	2.0442
2.1974	2.3274	2.3275
2.4872	2.6509	2.6511
2.8159	3.0190	3.0192
3.1875	3.4364	3.4366

In the last section we provided some Maple code for performing Euler's method. A similar code in MATLAB looks like the following:

```
a=0;
b=1;
```

```
N=10;
h=(b-a)/N;

% Slope function
  f = inline('t+y','t','y');
  sol = inline('2*exp(t)-t-1','t');

% Initial Condition
  t(1)=0;
  y(1)=1;

% Euler's Method
  for i=2:N+1
      y(i)=y(i-1)+h*f(t(i-1),y(i-1));
      t(i)=t(i-1)+h;
  end
```

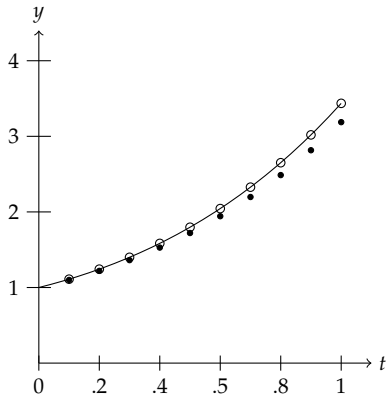

Figure 10.5: Numerical results for Euler's Method (filled circle) and 3rd Order Taylor's Method (open circle) for solving Example 10.2 as compared to exact solution (solid line).

A simple modification can be made for the 3rd Order Taylor's Method by replacing the Euler's method part of the preceding code by

```
% Taylor's Method, Order 3
  y(1)=1;
  h3 = h^2*(1/2+h/6);
  for i=2:N+1
      y(i)=y(i-1)+h*f(t(i-1),y(i-1))+h3*(1+t(i-1)+y(i-1));
      t(i)=t(i-1)+h;
  end
```

While the accuracy in the last example seemed sufficient, we have to remember that we only stopped at one unit of time. How can we be confident that the scheme would work as well if we carried out the computation for much longer times. For example, if the time unit were only a second, then one would need 86,400 times longer to predict a day forward. Of course, the scale matters. But, often we need to carry out numerical schemes for long times and we hope that the scheme not only converges to a solution, but that it converges to the solution to the given problem. Also, the previous example was relatively easy to program because we could provide a relatively simple form for $T^{(3)}(t,y)$ with a quick computation of the derivatives of $f(t,y)$. This is not always the case and higher order Taylor methods in this form are not typically used. Instead, one can approximate $T^{(n)}(t,y)$ by evaluating the known function $f(t,y)$ at selected values of t and y, leading to Runge-Kutta methods.

10.1.3 Runge-Kutta Methods

As we had seen in the last section, we can use higher order Taylor methods to derive numerical schemes for solving

$$\frac{dy}{dt} = f(t,y), \quad y(a) = y_0, \quad t \in [a,b], \tag{10.17}$$

using a scheme of the form

$$\begin{aligned} \tilde{y}_{i+1} &= \tilde{y}_i + hT^{(n)}(t_i, \tilde{y}_i), \quad i = 0, 1, \ldots, N-1, \\ \tilde{y}_0 &= y_0, \end{aligned} \tag{10.18}$$

where we have defined

$$T^{(n)}(t,y) = y'(t) + \frac{h}{2}y''(t) + \cdots + \frac{h^{(n-1)}}{n!}y^{(n)}(t).$$

In this section we will find approximations of $T^{(n)}(t,y)$ which avoid the need for computing the derivatives.

For example, we could approximate

$$T^{(2)}(t,y) = f(t,y) + \frac{h}{2}fracdfdt(t,y)$$

by

$$T^{(2)}(t,y) \approx af(t+\alpha, y+\beta)$$

for selected values of a, α, and β. This requires use of a generalization of Taylor's series to functions of two variables. In particular, for small α and β we have

$$\begin{aligned} af(t+\alpha, y+\beta) &= a\left[f(t,y) + \frac{\partial f}{\partial t}(t,y)\alpha + \frac{\partial f}{\partial y}(t,y)\beta \right. \\ &\quad \left. + \frac{1}{2}\left(\frac{\partial^2 f}{\partial t^2}(t,y)\alpha^2 + 2\frac{\partial^2 f}{\partial t \partial y}(t,y)\alpha\beta + \frac{\partial^2 f}{\partial y^2}(t,y)\beta^2\right)\right] \\ &\quad + \quad \text{higher order terms.} \end{aligned} \tag{10.19}$$

Furthermore, we need $\frac{df}{dt}(t,y)$. Since $y = y(t)$, this can be found using a generalization of the Chain Rule from Calculus III:

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

Thus,

$$T^{(2)}(t,y) = f(t,y) + \frac{h}{2}\left[\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}\frac{dy}{dt}\right].$$

Comparing this expression to the linear (Taylor series) approximation of $af(t+\alpha, y+\beta)$, we have

$$\begin{aligned} T^{(2)} &\approx af(t+\alpha, y+\beta) \\ f + \frac{h}{2}\frac{\partial f}{\partial t} + \frac{h}{2}f\frac{\partial f}{\partial y} &\approx af + a\alpha\frac{\partial f}{\partial t} + \beta\frac{\partial f}{\partial y}. \end{aligned} \tag{10.20}$$

We see that we can choose

$$a = 1, \quad \alpha = \frac{h}{2}, \quad \beta = \frac{h}{2} f.$$

This leads to the numerical scheme

$$\begin{aligned} \tilde{y}_{i+1} &= \tilde{y}_i + hf\left(t_i + \frac{h}{2}, \tilde{y}_i + \frac{h}{2} f(t_i, \tilde{y}_i)\right), \quad i = 0, 1, \ldots, N-1, \\ \tilde{y}_0 &= y_0, \end{aligned} \tag{10.21}$$

This Runge-Kutta scheme is called the Midpoint Method, or Second Order Runge-Kutta Method, and it has order 2 if all second order derivatives of $f(t, y)$ are bounded.

The Midpoint or Second Order Runge-Kutta Method.

Often, in implementing Runge-Kutta schemes, one computes the arguments separately as shown in the following MATLAB code snippet. (This code snippet could replace the Euler's Method section in the code in the last section.)

```
% Midpoint Method
  y(1)=1;
  for i=2:N+1
      k1=h/2*f(t(i-1),y(i-1));
      k2=h*f(t(i-1)+h/2,y(i-1)+k1);
      y(i)=y(i-1)+k2;
      t(i)=t(i-1)+h;
  end
```

Example 10.3. *Compare the Midpoint Method with the 2nd Order Taylor's Method for the problem*

$$y' = t^2 + y, \quad y(0) = 1, \quad t \in [0, 1]. \tag{10.22}$$

The solution to this problem is $y(t) = 3e^t - 2 - 2t - t^2$. *In order to implement the 2nd Order Taylor's Method, we need*

$$\begin{aligned} T^{(2)} &= f(t, y) + \frac{h}{2} f'(t, y) \\ &= t^2 + y + \frac{h}{2}(2t + t^2 + y). \end{aligned} \tag{10.23}$$

The results of the implementation are shown in Table 10.3.

There are other way to approximate higher order Taylor polynomials. For example, we can approximate $T^{(3)}(t, y)$ using four parameters by

$$T^{(3)}(t, y) \approx af(t, y) + bf(t + \alpha, y + \beta f(t, y).$$

Expanding this approximation and using

$$T^{(3)}(t, y) \approx f(t, y) + \frac{h}{2} \frac{df}{dt}(t, y) + \frac{h^2}{6} \frac{df}{dt}(t, y),$$

we find that we cannot get rid of $O(h^2)$ terms. Thus, the best we can do is derive second order schemes. In fact, following a procedure similar to the derivation of the Midpoint Method, we find that

$$a + b = 1, \quad , \alpha b = \frac{h}{2}, \beta = \alpha.$$

Table 10.5: Numerical values for 2nd Order Taylor's Method, Midpoint Method, exact solution, and errors for solving Example 10.3 with $N = 10$..

Exact	Taylor	Error	Midpoint	Error
1.0000	1.0000	0.0000	1.0000	0.0000
1.1055	1.1050	0.0005	1.1053	0.0003
1.2242	1.2231	0.0011	1.2236	0.0006
1.3596	1.3577	0.0019	1.3585	0.0010
1.5155	1.5127	0.0028	1.5139	0.0016
1.6962	1.6923	0.0038	1.6939	0.0023
1.9064	1.9013	0.0051	1.9032	0.0031
2.1513	2.1447	0.0065	2.1471	0.0041
2.4366	2.4284	0.0083	2.4313	0.0053
2.7688	2.7585	0.0103	2.7620	0.0068
3.1548	3.1422	0.0126	3.1463	0.0085

There are three equations and four unknowns. Therefore there are many second order methods. Two classic methods are given by the modified Euler method ($a = b = \frac{1}{2}$, $\alpha = \beta = h$) and Huen's method ($a = \frac{1}{4}$, $b = \frac{3}{4}$, $\alpha = \beta = \frac{2}{3}h$).

The Fourth Order Runge-Kutta.

The Fourth Order Runge-Kutta Method, which is most often used, is given by the scheme

$$\begin{aligned}
\tilde{y}_0 &= y_0, \\
k_1 &= hf(t_i, \tilde{y}_i), \\
k_2 &= hf(t_i + \frac{h}{2}, \tilde{y}_i + \frac{1}{2}k_1), \\
k_3 &= hf(t_i + \frac{h}{2}, \tilde{y}_i + \frac{1}{2}k_2), \\
k_4 &= hf(t_i + h, \tilde{y}_i + k_3), \\
\tilde{y}_{i+1} &= \tilde{y}_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \quad i = 0, 1, \ldots, N-1. \quad (10.24)
\end{aligned}$$

Again, we can test this on Example 10.3 with $N = 10$. The MATLAB implementation is given by

```
% Runge-Kutta 4th Order to solve dy/dt = f(t,y), y(a)=y0, on [a,b]
  clear

  a=0;
  b=1;
  N=10;
  h=(b-a)/N;

% Slope function
  f = inline('t^2+y','t','y');
  sol = inline('-2-2*t-t^2+3*exp(t)','t');

% Initial Condition
  t(1)=0;
```

```
  y(1)=1;

% RK4 Method
  y1(1)=1;
  for i=2:N+1
      k1=h*f(t(i-1),y1(i-1));
      k2=h*f(t(i-1)+h/2,y1(i-1)+k1/2);
      k3=h*f(t(i-1)+h/2,y1(i-1)+k2/2);
      k4=h*f(t(i-1)+h,y1(i-1)+k3);
      y1(i)=y1(i-1)+(k1+2*k2+2*k3+k4)/6;
      t(i)=t(i-1)+h;
  end
```

MATLAB has built-in ODE solvers, as do other software packages, like Maple and Mathematica. You should also note that there are currently open source packages, such as Python based NumPy and Matplotlib, or Octave, of which some packages are contained within the Sage Project.

MATLAB has built-in ODE solvers, such as **ode45** for a fourth order Runge-Kutta method. Its implementation is given by

```
[t,y]=ode45(f,[0 1],1);
```

In this case f is given by an inline function like in the above RK4 code. The time interval is entered as $[0, 1]$ and the 1 is the initial condition, $y(0) = 1$.

However, **ode45** is not a straight forward RK4 implementation. It is a hybrid method in which a combination of 4th and 5th order methods are combined allowing for adaptive methods to handled subintervals of the integration region which need more care. In this case, it implements a fourth order Runge-Kutta-Fehlberg method. Running this code for the above example actually results in values for $N = 41$ and not $N = 10$. If we wanted to have the routine output numerical solutions at specific times, then one could use the following form

```
tspan=0:h:1;
[t,y]=ode45(f,tspan,1);
```

In Table 10.6 we show the solutions which results for Example 10.3 comparing the RK4 snippet above with **ode45**. As you can see RK4 is much better than the previous implementation of the second order RK (Midpoint) Method. However, the MATLAB routine is two orders of magnitude better that RK4.

There are many ODE solvers in MATLAB. These are typically useful if RK4 is having difficulty solving particular problems. For the most part, one is fine using RK4, especially as a starting point. For example, there is **ode23**, which is similar to **ode45** but combining a second and third order scheme. Applying the results to Example 10.3 we obtain the results in Table 10.6. We compare these to the second order Runge-Kutta method. The code snippets are shown below.

```
% Second Order RK Method
  y1(1)=1;
```

Table 10.6: Numerical values for Fourth Order Runge-Kutta Method, rk45, exact solution, and errors for solving Example 10.3 with $N = 10$.

Exact	Taylor	Error	Midpoint	Error
1.0000	1.0000	0.0000	1.0000	0.0000
1.1055	1.1055	4.5894e-08	1.1055	-2.5083e-10
1.2242	1.2242	1.2335e-07	1.2242	-6.0935e-10
1.3596	1.3596	2.3850e-07	1.3596	-1.0954e-09
1.5155	1.5155	3.9843e-07	1.5155	-1.7319e-09
1.6962	1.6962	6.1126e-07	1.6962	-2.5451e-09
1.9064	1.9064	8.8636e-07	1.9064	-3.5651e-09
2.1513	2.1513	1.2345e-06	2.1513	-4.8265e-09
2.4366	2.4366	1.6679e-06	2.4366	-6.3686e-09
2.7688	2.7688	2.2008e-06	2.7688	-8.2366e-09
3.1548	3.1548	2.8492e-06	3.1548	-1.0482e-08

```
for i=2:N+1
    k1=h*f(t(i-1),y1(i-1));
    k2=h*f(t(i-1)+h/2,y1(i-1)+k1/2);
    y1(i)=y1(i-1)+k2;
    t(i)=t(i-1)+h;
end

tspan=0:h:1;
[t,y]=ode23(f,tspan,1);
```

Table 10.7: Numerical values for Second Order Runge-Kutta Method, rk23, exact solution, and errors for solving Example 10.3 with $N = 10$.

Exact	Taylor	Error	Midpoint	Error
1.0000	1.0000	0.0000	1.0000	0.0000
1.1055	1.1053	0.0003	1.1055	2.7409e-06
1.2242	1.2236	0.0006	1.2242	8.7114e-06
1.3596	1.3585	0.0010	1.3596	1.6792e-05
1.5155	1.5139	0.0016	1.5154	2.7361e-05
1.6962	1.6939	0.0023	1.6961	4.0853e-05
1.9064	1.9032	0.0031	1.9063	5.7764e-05
2.1513	2.1471	0.0041	2.1512	7.8665e-05
2.4366	2.4313	0.0053	2.4365	0.0001
2.7688	2.7620	0.0068	2.7687	0.0001
3.1548	3.1463	0.0085	3.1547	0.0002

We have seen several numerical schemes for solving initial value problems. There are other methods, or combinations of methods, which aim to refine the numerical approximations efficiently as if the step size in the current methods were taken to be much smaller. Some methods extrapolate solutions to obtain information outside of the solution interval. Others use one scheme to get a guess to the solution while refining, or correcting, this to obtain better solutions as the iteration through time proceeds. Such methods are described in courses in numerical analysis and in the literature. At this point we will apply these methods to several physics problems before continuing with analytical solutions.

10.2 The Heat Equation

10.2.1 The Finite Difference Method

The heat equation can be solved using separation of variables. However, many partial differential equations cannot be solved exactly and one needs to turn to numerical solutions. The heat equation is a simple test case for using numerical methods. Here we will use the simplest method, finite differences.

Let us consider the heat equation in one dimension,

$$u_t = ku_{xx}.$$

Boundary conditions and an initial condition will be applied later. The starting point is figuring out how to approximate the derivatives in this equation.

Recall that the partial derivative, u_t, is defined by

$$\frac{\partial u}{\partial t} = \lim_{\Delta t \to \infty} \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t}.$$

Therefore, we can use the approximation

$$\frac{\partial u}{\partial t} \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t}. \tag{10.25}$$

This is called a forward difference approximation.

In order to find an approximation to the second derivative, u_{xx}, we start with the forward difference

$$\frac{\partial u}{\partial x} \approx \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x}.$$

Then,

$$\frac{\partial u_x}{\partial x} \approx \frac{u_x(x + \Delta x, t) - u_x(x, t)}{\Delta x}.$$

We need to approximate the terms in the numerator. It is customary to use a backward difference approximation. This is given by letting $\Delta x \to -\Delta x$ in the forward difference form,

$$\frac{\partial u}{\partial x} \approx \frac{u(x, t) - u(x - \Delta x, t)}{\Delta t}. \tag{10.26}$$

Applying this to u_x evaluated at $x = x$ and $x = x + \Delta x$, we have

$$u_x(x, t) \approx \frac{u(x, t) - u(x - \Delta x, t)}{\Delta x},$$

and

$$u_x(x + \Delta x, t) \approx \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x}.$$

Inserting these expressions into the approximation for u_{xx}, we have

$$\begin{aligned}\frac{\partial^2 u}{\partial x^2} &= \frac{\partial u_x}{\partial x} \\ &\approx \frac{u_x(x+\Delta x,t)-u_x(x,t)}{\Delta x} \\ &\approx \frac{\frac{u(x+\Delta x,t)-u(x,t)}{\Delta x}-\frac{u(x,t)-u(x-\Delta x,t)}{\Delta x}}{\Delta x} \\ &= \frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)}{(\Delta x)^2}. \end{aligned} \tag{10.27}$$

This approximation for u_{xx} is called the central difference approximation of u_{xx}.

Combining Equation (10.25) with (10.27) in the heat equation, we have

$$\frac{u(x,t+\Delta t)-u(x,t)}{\Delta t} \approx k\frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)}{(\Delta x)^2}.$$

Solving for $u(x,t+\Delta t)$, we find

$$u(x,t+\Delta t) \approx u(x,t)+\alpha\left[u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)\right], \tag{10.28}$$

where $\alpha = k\frac{\Delta t}{(\Delta x)^2}$.

In this equation we have a way to determine the solution at position x and time $t+\Delta t$ given that we know the solution at three positions, x, $x+\Delta x$, and $x+2\Delta x$ at time t.

$$u(x,t+\Delta t) \approx u(x,t)+\alpha\left[u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)\right]. \tag{10.29}$$

A shorthand notation is usually used to write out finite difference schemes. The domain of the solution is $x \in [a,b]$ and $t \geq 0$. We seek approximate values of $u(x,t)$ at specific positions and times. We first divide the interval $[a,b]$ into N subintervals of width $\Delta x = (b-a)/N$. Then, the endpoints of the subintervals are given by

$$x_i = a + i\Delta x, \quad i = 0,1,\ldots,N.$$

Similarly, we take time steps of Δt, at times

$$t_j = j\Delta t, \quad j = 0,1,2,\ldots.$$

This gives a grid of points (x_i,t_j) in the domain.

At each grid point in the domain we seek an approximate solution to the heat equation, $u_{i,j} \approx u(x_i,t_j)$. Equation (10.29) becomes

$$u_{i,j+1} \approx u_{i,j}+\alpha\left[u_{i+1,j}-2u_{i,j}+u_{i-1,j}\right]. \tag{10.30}$$

Equation (10.31) is the finite difference scheme for solving the heat equation. This equation is represented by the stencil shown in Figure 10.6. The black circles represent the four terms in the equation, $u_{i,j}$ $u_{i-1,j}$ $u_{i+1,j}$ and $u_{i,j+1}$.

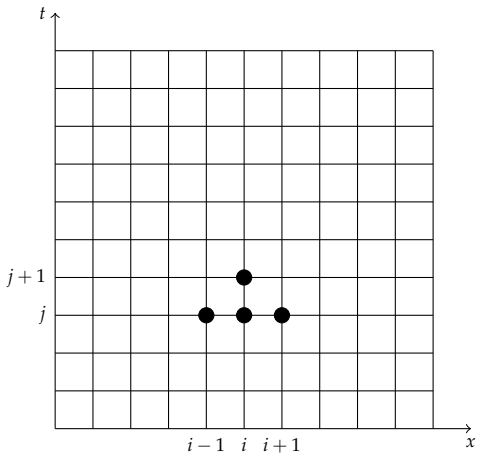

Figure 10.6: This *stencil* indicates the four types of terms in the finite difference scheme in Equation (10.31). The black circles represent the four terms in the equation, $u_{i,j}$ $u_{i-1,j}$ $u_{i+1,j}$ and $u_{i,j+1}$.

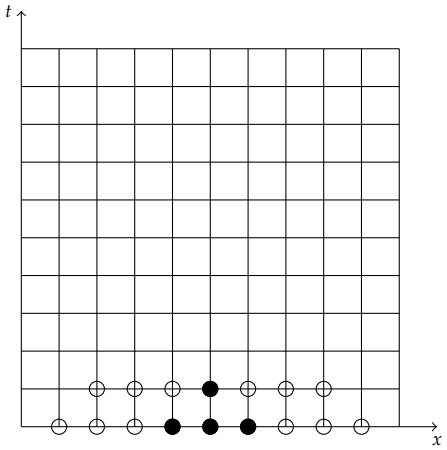

Figure 10.7: Applying the stencil to the row of initial values gives the solution at the next time step.

Let's assume that the initial condition is given by

$$u(x,0) = f(x).$$

Then, we have $u_{i,0} = f(x_i)$. Knowing these values, denoted by the open circles in Figure 10.7, we apply the stencil to generate the solution on the $j = 1$ row. This is shown in Figure 10.7.

Further rows are generated by successively applying the stencil on each row, using the known approximations of $u_{i,j}$ at each level. This gives the values of the solution at the open circles shown in Figure 10.8. We notice that the solution can only be obtained at a finite number of points on the grid.

In order to obtain the missing values, we need to impose boundary conditions. For example, if we have Dirichlet conditions at $x = a$,

$$u(a,t) = 0,$$

or $u_{0,j} = 0$ for $j = 0, 1, \ldots$, then we can fill in some of the missing data points as seen in Figure 10.9.

The process continues until we again go as far as we can. This is shown in Figure 10.10.

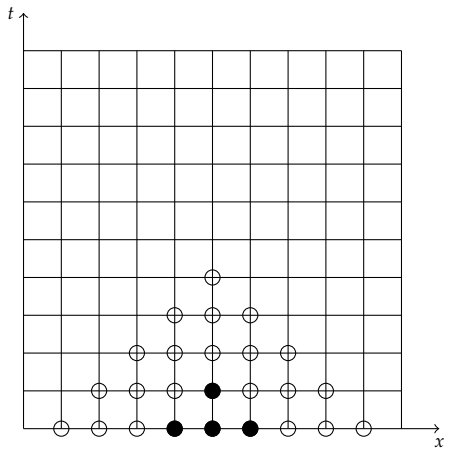

Figure 10.8: Continuation of the process provides solutions at the indicated points.

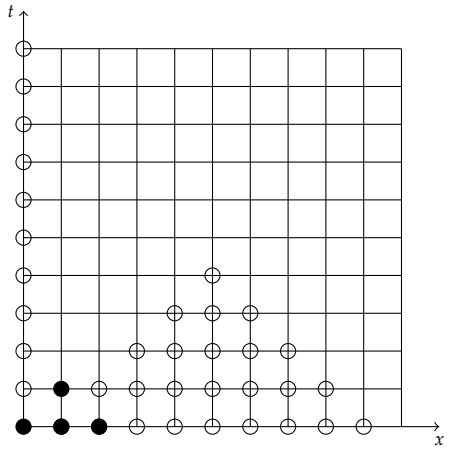

Figure 10.9: Knowing the values of the solution at $x = a$, we can fill in more of the grid.

We can fill in the rest of the grid using a boundary condition at $x = b$. For Dirichlet conditions at $x = b$,

$$u(b, t) = 0,$$

or $u_{N,j} = 0$ for $j = 0, 1, \ldots$, then we can fill in the rest of the missing data points as seen in Figure 10.11.

We could also use Neumann conditions. For example, let

$$u_x(a, t) = 0.$$

The approximation to the derivative gives

$$\frac{\partial u}{\partial x}\Big|_{x=a} \approx \frac{u(a + \Delta x, t) - u(a, t)}{\Delta x} = 0.$$

Then,

$$u(a + \Delta x, t) - u(a, t),$$

or $u_{0,j} = u_{1,j}$, for $j = 0, 1, \ldots$. Thus, we know the values at the boundary and can generate the solutions at the grid points as before.

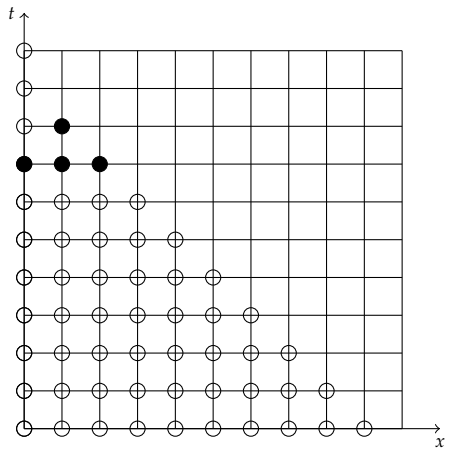

Figure 10.10: Knowing the values of the solution at $x = a$, we can fill in more of the grid until we stop.

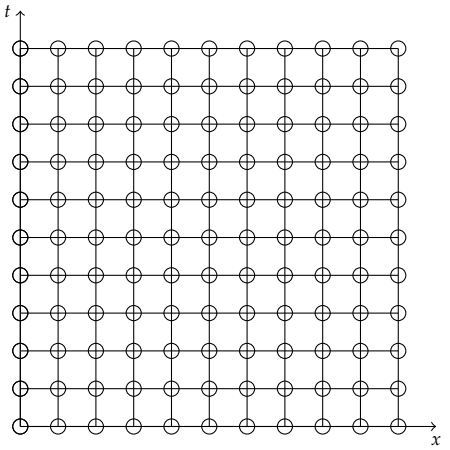

Figure 10.11: Using boundary conditions and the initial condition, the grid can be fill in through any time level.

We now have to code this using software. We can use MATLAB to do this. An example of the code is given below. In this example we specify the length of the rod, $L = 1$, and the heat constant, $k = 1$. The code is run for $t \in [0, 0.1]$.

The grid is created using $N = 10$ subintervals in space and $M = 50$ time steps. This gives dx = Δx and dt = Δt. Using these values, we find the numerical scheme constant $\alpha = k\Delta t/(\Delta x)^2$.

Nest, we define $x_i = i * dx$, $i = 0, 1, \ldots, N$. However, in MATLAB, we cannot have an index of 0. We need to start with $i = 1$. Thus, $x_i = (i-1) * dx$, $i = 1, 2, \ldots, N+1$.

Next, we establish the initial condition. We take a simple condition of

$$u(x, 0) = \sin \pi x.$$

We have enough information to begin the numerical scheme as developed earlier. Namely, we cycle through the time steps using the scheme. There is one loop for each time step. We will generate the new time step from the last time step in the form

$$u_i^{new} = u_i^{old} + \alpha \left[u_{i+1}^{old} - 2u_i^{old} + u_{i-1}^{old} \right]. \tag{10.31}$$

This is done suing $u0(i) = u_i^{new}$ and $u1(i) = u_i^{old}$.

At the end of each time loop we update the boundary points so that the grid can be filled in as discussed. When done, we can plot the final solution. If we want to show solutions at intermediate steps, we can plot the solution earlier.

```
% Solution of the Heat Equation Using a Forward Difference Scheme

% Initialize Data
%     Length of Rod, Time Interval
%     Number of Points in Space, Number of Time Steps

L=1;
T=0.1;
k=1;
N=10;
M=50;
dx=L/N;
dt=T/M;
alpha=k*dt/dx^2;

% Position

for i=1:N+1
    x(i)=(i-1)*dx;
end

% Initial Condition

for i=1:N+1
    u0(i)=sin(pi*x(i));
end

% Partial Difference Equation (Numerical Scheme)

for j=1:M
   for i=2:N
      u1(i)=u0(i)+alpha*(u0(i+1)-2*u0(i)+u0(i-1));
   end
   u1(1)=0;
   u1(N+1)=0;
   u0=u1;
end

% Plot solution
plot(x, u1);
```

10.3 Truncation Error

In the previous section we found a finite difference scheme for numerically solving the one dimensional heat equation. We have from Equations (10.29) and (10.31),

$$u(x,t+\Delta t) \approx u(x,t) + \alpha\left[u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)\right] \quad (10.32)$$

$$u_{i,j+1} \approx u_{i,j} + \alpha\left[u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right], \quad (10.33)$$

where $\alpha = k\Delta t/(\Delta x)^2$. For points $x \in [a,b]$ and $t \geq 0$, we use the scheme to find approximate values of $u(x_i,t_i) = u_{i,j}$ at positions $x_i = a + i\Delta x, \quad i = 0,1,\ldots,N$, and times $t_j = j\Delta t, \quad j = 0,1,2,\ldots$.

In implementing the scheme we have found that there are errors introduced just like when using Euler's Method for ordinary differential equations. These truncations errors can be found by applying Taylor approximations just like we had for ordinary differential equations. In the schemes (10.32) and (10.33), we have not use equality. In order to replace the approximation by an equality, we need t estimate the order of the terms neglected in a Taylor series approximation of the time and space derivatives we have approximated.

We begin with the time derivative approximation. We used the forward difference approximation (10.25),

$$\frac{\partial u}{\partial t} \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t}. \quad (10.34)$$

This can be derived from the Tayloer series expansion of $u(x,t+\Delta t)$ about $\Delta t = 0$,

$$u(x,t+\Delta t) = u(x,t) + \frac{\partial u}{\partial t}(x,t)\Delta t + \frac{1}{2!}\frac{\partial^2 u}{\partial t^2}(x,t)(\Delta t)^2 + O((\Delta t)^3).$$

Solving for $\frac{\partial u}{\partial t}(x,t)$, we obtain

$$\frac{\partial u}{\partial t}(x,t) = \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} - \frac{1}{2!}\frac{\partial^2 u}{\partial t^2}(x,t)\Delta t + O((\Delta t)^2).$$

We see that we have obtained the forward difference approximation (10.25) with the added benefit of knowing something about the error terms introduced in the approximation. Namely, when we approximate u_t with the forward difference approximation (10.25), we are making an error of

$$E(x,t,\Delta t) = -\frac{1}{2!}\frac{\partial^2 u}{\partial t^2}(x,t)\Delta t + O((\Delta t)^2).$$

We have truncated the Taylor series to obtain this approximation and we say that

$$\frac{\partial u}{\partial t} = \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} + O(\Delta t) \quad (10.35)$$

is a first order approximation in Δt.

In a similar manor, we can obtain the truncation error for the $u_x x$- term. However, instead of starting with the approximation we used in Equation **??**uxx), we will derive a term using the Taylor series expansion of $u(x + \Delta x, t)$ about $\Delta x = 0$. Namely, we begin with the expansion

$$\begin{aligned} u(x+\Delta x,t) &= u(x,t) + u_x(x,t)\Delta x + \frac{1}{2!}u_{xx}(x,t)(\Delta x)^2 + \frac{1}{3!}u_{xxx}(x,t)(\Delta x)^3 \\ &\quad + \frac{1}{4!}u_{xxxx}(x,t)(\Delta x)^4 + \ldots. \end{aligned} \tag{10.36}$$

We want to solve this equation for u_{xx}. However, there are some obstructions, like needing to know the u_x term. So, we seek a way to eliminate lower order terms. On way is to note that replacing Δx by $-\Delta x$ gives

$$\begin{aligned} u(x-\Delta x,t) &= u(x,t) - u_x(x,t)\Delta x + \frac{1}{2!}u_{xx}(x,t)(\Delta x)^2 - \frac{1}{3!}u_{xxx}(x,t)(\Delta x)^3 \\ &\quad + \frac{1}{4!}u_{xxxx}(x,t)(\Delta x)^4 + \ldots. \end{aligned} \tag{10.37}$$

Adding these Taylor series, we have

$$\begin{aligned} u(x+\Delta x,t) + u(x+\Delta x,t) &= 2u(x,t) + u_{xx}(x,t)(\Delta x)^2 \\ &\quad + \frac{2}{4!}u_{xxxx}(x,t)(\Delta x)^4 + O((\Delta x)^6). \end{aligned} \tag{10.38}$$

We can now solve for u_{xx} to find

$$\begin{aligned} u_{xx}(x,t) &= \frac{u(x+\Delta x,t) - 2u(x,t) + u(x+\Delta x,t)}{(\Delta x)^2} \\ &\quad + \frac{2}{4!}u_{xxxx}(x,t)(\Delta x)^2 + O((\Delta x)^4). \end{aligned} \tag{10.39}$$

Thus, we have that

$$u_{xx}(x,t) = \frac{u(x+\Delta x,t) - 2u(x,t) + u(x+\Delta x,t)}{(\Delta x)^2} + O((\Delta x)^2)$$

is the second order in Δx approximation of u_{xx}.

Combining these results, we find that the heat equation is approximated by

$$\frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x+\Delta x,t) - 2u(x,t) + u(x+\Delta x,t)}{(\Delta x)^2} + O\left((\Delta x)^2, \Delta t\right).$$

This has local truncation error that is first order in time and second order in space.

10.4 Stability

ANOTHER CONSIDERATION FOR NUMERICAL SCHEMES for the heat equation is the stability of the scheme. In implementing the finite difference scheme,

$$u_{m,j+1} = u_{m,j} + \alpha\left[u_{m+1,j} - 2u_{m,j} + u_{m-1,j}\right], \tag{10.40}$$

$\alpha = k\Delta t/(\Delta x)^2$, one finds that the solution goes crazy when α is too big. In other words, if you try to push the individual time steps too far into the future, then something goes haywire. We can determine the onset of instability by looking at the solution of this equation for $u_{m,j}$. [Note: We changed index i to m to avoid confusion later in this section.]

The scheme is actually what is called a partial difference equation for $u_{m,j}$. We could write it in terms of difference, such as $u_{m+1,j} - u_{m,j}$ and $u_{m,j+1} - u_{m,j}$. The furthest apart the time steps are are one unit and the spatial points are two units apart. We can see this in the stencils in Figure 10.6. So, this is a second order partial difference equation similar to the idea that the heat equation is a second order partial differential equation. The heat equation can be solved using the method of separation of variables. The difference scheme can also be solved in a similar fashion. We will show how this can lead to product solutions.

We begin by assuming that $u_{mj} = X_m T_j$, a product of functions of the indices m and j. Inserting this guess into the finite difference equation, we have

$$\begin{aligned} u_{m,j+1} &= u_{m,j} + \alpha \left[u_{m+1,j} - 2u_{m,j} + u_{m-1,j}\right], \\ X_m T_{j+1} &= X_m T_j + \alpha \left[X_{m+1} - 2X_m + X_{m-1}\right] T_j, \\ \frac{T_{j+1}}{T_j} &= \frac{\alpha X_{m+1} + (1-2\alpha)X_m + \alpha X_{m-1}}{X_m}. \end{aligned} \tag{10.41}$$

Noting that we have a function of j equal to a function of m, then we can set each of these to a constant, λ. Then, we obtain two ordinary difference equations:

$$T_{j+1} = \lambda T_j, \tag{10.42}$$

$$\alpha X_{m+1} + (1-2\alpha)X_m + \alpha X_{m-1} = \lambda X_m. \tag{10.43}$$

The first equation is a simple first order difference equation and can be solved by iteration:

$$\begin{aligned} T_{j+1} &= \lambda T_j, \\ &= \lambda(\lambda T_{j-1}) = \lambda^2 T_{j-1}, \\ &= \lambda^3 T_{j-2}, \\ &= \lambda^{j+1} T_0, \end{aligned} \tag{10.44}$$

The second difference equation can be solved by making a guess in the same spirit as solving a second order constant coefficient differential equation.Namely, let $X_m = \xi^m$ for some number ξ. This gives

$$\begin{aligned} \alpha X_{m+1} + (1-2\alpha)X_m + \alpha X_{m-1} &= \lambda X_m, \\ \xi^{m-1}\left[\alpha\xi^2 + (1-2\alpha)\xi + \alpha\right] &= \lambda\xi^m \\ \alpha\xi^2 + (1-2\alpha-\lambda)\xi + \alpha &= 0. \end{aligned} \tag{10.45}$$

This is an equation foe ξ in terms of α and λ. Due to the boundary conditions, we expect to have oscillatory solutions. So, we can guess that

$\xi = |\xi| e^{i\theta}$, where i here is the imaginary unit. We assume that $|\xi| = 1$, and thus $\xi = e^{i\theta}$ and $X_m = \xi^m = e^{im\theta}$. Since $x_m = m\Delta x$, we have $X_m = e^{ix_m\theta/\Delta x}$. We define $\beta = theta/\Delta$, to give $X_m = e^{i\beta x_m}$ and $\xi = e^{i\beta\Delta x}$.

Inserting this value for ξ into the quadratic equation for ξ, we have

$$\begin{aligned} 0 &= \alpha\xi^2 + (1 - 2\alpha - \lambda)\xi + \alpha \\ &= \alpha e^{2i\beta\Delta x} + (1 - 2\alpha - \lambda)e^{i\beta\Delta x} + \alpha \\ &= e^{i\beta\Delta x}\left[\alpha(e^{i\beta\Delta x} + e^{-i\beta\Delta x}) + (1 - 2\alpha - \lambda)\right] \\ &= e^{i\beta\Delta x}\left[2\alpha\cos(\beta\Delta x) + (1 - 2\alpha - \lambda)\right] \\ \lambda &= 2\alpha\cos(\beta\Delta x) + 1 - 2\alpha. \end{aligned} \tag{10.46}$$

So, we have found that

$$u_{mj} = X_m T_j = \lambda^m(a\cos\alpha x_m + b\sin\alpha x_m), \quad a^2 + b^2 = h_0^2,$$

and

$$\lambda = 2\alpha\cos(\beta\Delta x) + 1 - 2\alpha.$$

For the solution to remain bounded, or stable, we need $|\lambda| \leq 1$.

Therefore, we have the inequality

$$-1 \leq 2\alpha\cos(\beta\Delta x) + 1 - 2\alpha \leq 1.$$

Since $\cos(\beta\Delta x) \leq 1$, the upper bound is obviously satisfied. Since $-1 \leq \cos(\beta\Delta x)$, the lower bound is satisfied for $-1 \leq -2s + 1 - 2s$, or $s \leq \frac{1}{2}$. Therefore, the stability criterion is satisfied when

$$\alpha = k\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}.$$

10.5 Matrix Formulation