Numerical Methods for Differential Equations

Ya Yan Lu
Department of Mathematics
City University of Hong Kong
Kowloon, Hong Kong

Contents

1 ODE IVP: Explicit One-step Methods **4**
1.1 Introduction 4
1.2 Euler and Runge-Kutta Methods 5
1.3 Local truncation error and order 10
1.4 Embedded Runge-Kutta methods 13

2 ODE IVP: Implicit One-step Methods **17**
2.1 Stiff equations 17
2.2 Implicit one-step methods 21

3 ODE IVP: Multi-step Methods **23**
3.1 Explicit multi-step methods 23
3.2 Implicit multi-step methods 26

4 ODE IVP: Stability Concepts **29**
4.1 Zero stability 29
4.2 Absolute stability 31

5 ODE Boundary Value Problems **34**
5.1 The shooting method 34
5.2 Finite difference methods 37
5.3 The finite element method 39

6 Finite Difference Methods for Parabolic PDEs **42**
6.1 Introduction 42
6.2 Classical explicit method 43
6.3 Crank-Nicolson method 44
6.4 Stability analysis 45
6.5 Alternating direction implicit method 48

7 Finite Difference Methods for Hyperbolic PDEs **52**
7.1 First order hyperbolic equations 52
7.2 Explicit methods for wave equation 55
7.3 Maxwell's equations 57

8 Finite Difference Methods for Elliptic PDEs **60**
8.1 Finite difference method for Poisson equation 60
8.2 Fast Poisson solver based on FFT 61
8.3 Classical iterative methods 62
8.4 Conjugate gradient method 64
8.4.1 1-D optimization problem 64
8.4.2 Subspace minimization problem 65
8.4.3 Orthogonal residual 66
8.4.4 The next conjugate direction 66
8.4.5 The method 67
8.4.6 Rate of convergence 67

Chapter 1

ODE IVP: Explicit One-step Methods

1.1 Introduction

In this chapter, we study numerical methods for initial value problems (IVP) of ordinary differential equations (ODE). The first step is to re-formulate your ODE as a system of first order ODEs:

$$\frac{dy}{dt} = f(t,y) \quad \text{for} \quad t > t_0 \tag{1.1}$$

with the initial condition

$$y(t_0) = y_0 \tag{1.2}$$

where t is the independent variable, $y = y(t)$ is the unknown function of t, y_0 is the given initial condition, and f is a given function of t and y which describes the differential equation. High order differential equations can also be written as a first order system by introducing the derivatives as new functions. Our numerical methods can be used to solve any ordinary differential equations. We only need to specify the function f.

The variable t is discretized, say t_j for $j = 0,1,2,...$, then we determine $y_j \approx y(t_j)$ for $j = 1,2,3,...$. The first class of methods (Runge-Kutta methods) involve one-step. If y_j is calculated, then we construct y_{j+1} from y_j. Previous values such as y_{j-1} are not needed. Since this is an IVP and for the first step, we have y_0 only at t_0, then we can find y_1, y_2, ..., in a sequence. The one-step methods are vary natural. A higher order method gives a more accurate numerical solution than a lower order method for a fixed step size. But a higher order one-step method requires more evaluations of the f function. For example, the first order Euler's method requires only one evaluation of f, i.e., $f(t_j, y_j)$, but a fourth order Runge-Kutta method requires four evaluations of f.

For a large scale problem, the computation of f could be time consuming. Thus, it is desirable to have high order methods that require only one evaluation of f in each step. This is not possible in a one-step method. But it is possible in a multi-step method. Therefore, the main advantage of the multi-step method is that they are efficient. However, they are more difficult to use.

For one-step methods, we will introduce implicit methods. These are methods designed for the so-called "stiff" ODEs. If an explicit method is used for a stiff ODE and the step size is not small enough, the error (between the exact and the numerical solution) may grow very fast. For these stiff ODEs, the

implicit methods are useful. The situation is the same for multi-step methods. We also need implicit multi-step methods for stiff ODEs.

We will also introduce the embedded Runge-Kutta methods. These are methods that combine two methods together, so that the step size can be automatically chosen for a desired accuracy. There are also multi-step methods that allow automatic selection of the step size. But they are more complicated and we will not cover them.

Consider the following example. We have the following differential equation for $u = u(t)$:

$$u''' + \sin(t)\sqrt{1+(u'')^2}u' + \frac{u}{1+e^{-t}} = t^2 \tag{1.3}$$

for $t > 0$, with the initial conditions:

$$u(0) = 1, \quad u'(0) = 2, \quad u''(0) = 3. \tag{1.4}$$

We can introduce a vector y

$$y(t) = \begin{bmatrix} u(t) \\ u'(t) \\ u''(t) \end{bmatrix}$$

and write down the equation for y as

$$y' = f(t,y) = \begin{bmatrix} u' \\ u'' \\ -\sin(t)\sqrt{1+(u'')^2}\, u' - u/(1+e^{-t}) + t^2 \end{bmatrix}$$

The initial condition is $y(0) = [1,2,3]$. Here is a simple MATLAB program for the above function f.

```
function k = f(t, y)
%  remember y is a column vector of three components.
k = zeros(3,1);
k(1) = y(2);
k(2) = y(3);
k(3) = -sin(t) * sqrt(1+y(3)^2) * y(2) - y(1)/(1 + exp(-t)) + t^2;
```

In the MATLAB program, `y(1), y(2), y(3)` are the three components of the vector y. They are $u(t)$, $u'(t)$ and $u''(t)$, respectively. They are different from $y(1)$, $y(2)$ and $y(3)$ which are the vectors y evaluated at $t = 1$, $t = 2$ and $t = 3$. Notice that we also have $y(0)$, which is the initial value of y. But we do not have `y(0)`. Anyway, the components of y are only used inside the MATLAB programs.

A numerical method is usually given for the general system (1.1-1.2). We specify the system of ODEs by writing a program for the function f, then the same numerical method can be easily used for solving many different differential equations.

1.2 Euler and Runge-Kutta Methods

Numerical methods start with a discretization of t by t_0, t_1, t_2, ..., say

$$t_j = t_0 + jh$$

where h is the **step size**. Numerical methods are formulas for y_1, y_2, y_3, ..., where y_j is the approximate solution at t_j. We use $y(t_j)$ to denote the (unknown) exact solution, thus

$$y_j \approx y(t_j).$$

Please notice that when y is a vector, y_1, y_2, ..., are also vectors. In particular, y_1 is not the first component of y vector, y_2 is not the 2nd component of the y vector. The components of y are only explicitly given inside the MATLAB programs as `y(1)`, `y(2)`, etc.

Euler's method:

$$y_{j+1} = y_j + hf(t_j, y_j). \tag{1.5}$$

Since y_0 is the known initial condition, the above formula allows us o find y_1, y_2, etc, in a sequence. The Euler's method can be easily derived as follows. First, we assume h is small and consider the Taylor expansion:

$$y(t_1) = y(t_0 + h) = y(t_0) + hy'(t_0) + ...$$

Now, we know that $y'(t_0) = f(t_0, y(t_0))$. If we keep only the first two terms of the Taylor series, we obtain the first step of Euler's method:

$$y_1 = y_0 + hf(t_0, y_0),$$

where $y(t_1)$ is replaced by the "numerical solution" y_1, etc. The general step from t_j to t_{j+1} is similar.

Here is a MATLAB program for the Euler's method:

```
function y1 = eulerstep(h, t0, y0)
% This is one step of the Euler's method. It is
% given for the first step, but any other step
% is just the same. You need the MATLAB function
% f to specify the system of ODEs.
y1 = y0 + h* f(t0, y0)
```

Now, let us solve (1.3-1.4) from $t = 0$ to $t = 1$ with the step size $h = 0.01$. For this purpose, we need to write a main program. In the main program, we specify the initial conditions, initial time t_0, final time and the total number of steps. The step size can then be calculated. Here is the MATLAB program.

```
%  The main program to solve (1.3)-(1.4) from t=0 to
%  t = 1 by Euler's method.

% initial time
t0 = 0;
% final time
tfinal = 1;
% number of steps
nsteps =  100;
% step size
```

```
h = (tfinal - t0)/ nsteps;
% initial conditions
y = [1, 2, 3]';
% set the variable t.
t = t0
% go through the steps.
for j= 1 : nsteps
  y = eulerstep(h, t, y)
  t = t + h
  % saved output for u(t) only, i.e. the first component of y.
  tout(j) = t;
  u(j) = y(1);
end
% draw a figure for the solution u.
plot(tout, u)
```

Now, insider MATLAB, in a folder containing the three programs: `f.m`, `eulerstep.m`, `eulermain.m`, if we type `eulermain`, we will see a solution curve. That is the solid curve in Fig. 1.1. This is for the

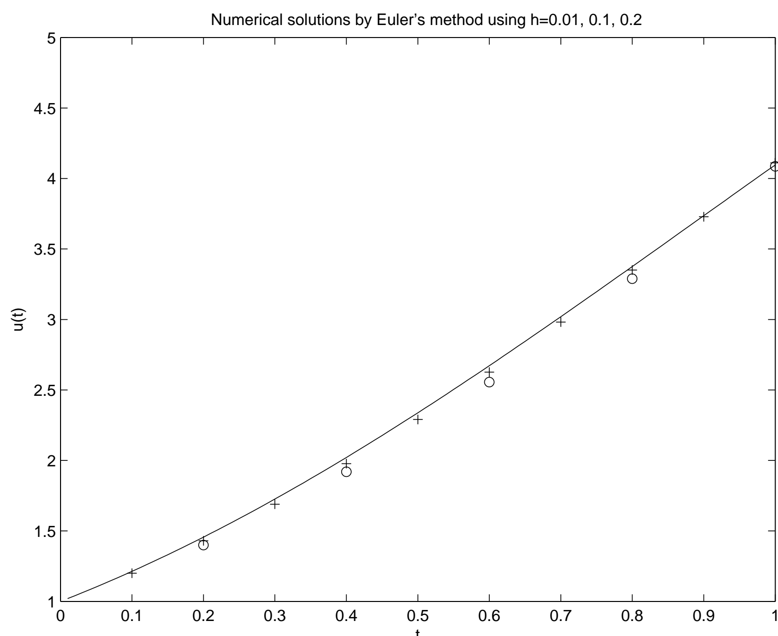

Figure 1.1: Numerical solutions of (1.3) and (1.4) by Euler's method. The solid curve is for $h = 0.01$. The "$+$" is for $h = 0.1$ and the "o" is for $h = 0.2$.

case of $h = 0.01$. We also want to see what happens if h is 0.2 and 0.1. For this purpose, we change `nsteps` to 5 and 10, then use `plot(tout, u, 'o')` and `plot(tout, u, '+')` to show the results. All three plots are shown in the Fig. 1.1.

The Euler's method is not very accurate. To obtain a numerical solution with an acceptable accuracy, we have to use a very small step size h. A small step size h implies a larger number of steps, thus more

computing time. It is desirable to develop methods that are more accurate than Euler's method. If we look at the Taylor series again, we have

$$y(t_1) = y(t_0 + h) = y(t_0) + hy'(t_0) + \frac{h^2}{2}y''(t_0) + \frac{h^3}{6}y'''(t_0) + \dots$$

This can be written as

$$\frac{y(t_1) - y(t_0)}{h} = y'(t_0) + \frac{h}{2}y''(t_0) + \frac{h^2}{6}y'''(t_0) + \dots \tag{1.6}$$

Actually, the right hand side is a more accurate approximation for $y'(t_0 + h/2)$, since

$$y'(t_0 + \frac{h}{2}) = y'(t_0) + \frac{h}{2}y''(t_0) + \frac{h^2}{8}y'''(t_0) + \dots$$

The first two terms on the right hand sides of the above two equations are identical, although the third terms involving $y'''(t_0)$ are different. Thus,

$$\frac{y(t_1) - y(t_0)}{h} \approx y'(t_0 + \frac{h}{2}) = f\left(t_0 + \frac{h}{2}, y(t_0 + \frac{h}{2})\right)$$

The right hand side now involves $y(t_0 + h/2)$. Of course, this is now known, because we only have $y(t_0)$. The idea is that we can use Euler's method (with half step size $h/2$) to get an approximate $y(t_0 + h/2)$, then use the above to get an approximation of $y(t_1)$. The Euler approximation for $y(t_0 + h/2)$ is $y(t_0) + h/2 f(t_0, y_0)$. Therefore, we have

$$k_1 = f(t_0, y_0) \tag{1.7}$$

$$k_2 = f(t_0 + \frac{h}{2}, y_0 + \frac{h}{2}k_1) \tag{1.8}$$

$$y_1 = y_0 + hk_2. \tag{1.9}$$

This is the first step of the so-called **midpoint** method. The general step is obtained by simply replacing t_0, y_0 and y_1 by t_j, y_j and y_{j+1}, respectively.

The right hand side of (1.6) can also be approximated by $(y'(t_0) + y'(t_1))/2$, because

$$\frac{y'(t_0) + y'(t_1)}{2} = y'(t_0) + \frac{h}{2}y''(t_0) + \frac{h^2}{4}y'''(t_0) + \dots$$

Therefore, we have

$$\frac{y(t_1) - y(t_0)}{h} \approx \frac{y'(t_0) + y'(t_1)}{2}.$$

We can replace $y'(t_0)$ and $y'(t_1)$ by $f(t_0, y(t_0))$ and $f(t_1, y(t_1))$, but of course, we do not know $y(t_1)$, because that is what we are trying to solve. But we can use Euler's method to get the first approximation of $y(t_1)$ and use it in $f(t_1, y(t_1))$, then use the above to get the second (and better) approximation of $y(t_1)$. This can be summarized as

$$k_1 = f(t_0, y_0) \tag{1.10}$$

$$k_2 = f(t_0 + h, y_0 + hk_1) \tag{1.11}$$

$$y_1 = y_0 + \frac{h}{2}(k_1 + k_2). \tag{1.12}$$

This is the first step of the so-called **modified Euler's** method. The general step from t_j to t_{j+1} is easily obtained by replacing the subscripts 0 and 1 by j and $j+1$, respectively.

Similarly, the right hand side of (1.6) can be approximated by

$$Ay'(t_0) + By'(t_0 + \alpha h),$$

where α is a given constant, $0 < \alpha \leq 1$, the coefficients A and B can be determined, such that the above matches the first two terms of the right hand side of (1.6). We obtain

$$A = 1 - \frac{1}{2\alpha}, \quad B = \frac{1}{2\alpha}.$$

Then $y'(t_0 + \alpha h) = f(t_0 + \alpha h, y(t_0 + \alpha h))$ and we use Euler's method to approximate $y(t_0 + \alpha h)$. That is

$$y(t_0 + \alpha h) \approx y(t_0) + \alpha h f(t_0, y(t_0)).$$

Finally, we obtain the following general **2nd order Runge-Kutta Methods:**

$$k_1 = f(t_j, y_j) \tag{1.13}$$
$$k_2 = f(t_j + \alpha h, y_j + \alpha h k_1) \tag{1.14}$$
$$y_{j+1} = y_j + h\left[\left(1 - \frac{1}{2\alpha}\right)k_1 + \frac{1}{2\alpha}k_2\right] \tag{1.15}$$

Since α is an arbitrary parameter, there are infinitely many 2nd order Runge-Kutta methods. The midpoint method and the modified Euler's method correspond to $\alpha = 1/2$ and $\alpha = 1$, respectively. In this formula, k_1 and k_2 are temporary variables, they are different for different steps.

There are many other Runge-Kutta methods (3rd order, 4th order and higher order). The following **classical 4th order Runge-Kutta** method is widely used, because it is quite easy to remember.

$$k_1 = f(t_j, y_j) \tag{1.16}$$
$$k_2 = f(t_j + \frac{h}{2}, y_j + \frac{h}{2}k_1) \tag{1.17}$$
$$k_3 = f(t_j + \frac{h}{2}, y_j + \frac{h}{2}k_2) \tag{1.18}$$
$$k_4 = f(t_j + h, y_j + hk_3) \tag{1.19}$$
$$y_{j+1} = y_j + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{1.20}$$

We have mentioned the **order** of a method above. This concept will be explained in the next section.

Next, we consider a MATLAB implementation of the midpoint method. For this purpose, we write the following function called `midptstep` which is saved in the file called `midptstep.m`.

```
function y1 = midptstep(h, t0, y0)
  % This is midpoint method (one of the second order Runge-Kutta methods).
  % It is given for the first step, but any other step is just the same.
  % You need the MATLAB function f to specify the system of ODEs.
    k1 = f(t0, y0);
    k2 = f(t0+h/2, y0 + (h/2)*k1)
    y1 = y0 + h* k2;
```

To solve the same differential equation (1.3-1.4), we need the earlier MATLAB function `f` and a main program. We can write a main program by copying the main program `eulermain` for Euler's method. The new main program `midptmain` is different from `eulermain` only in one line. The line `y = eulerstep(h, t, y)` is now replaced by

```
y = midptstep(h, t, y)
```

You can see that writing a program for a new method is very easy, since we have separated the differential equation (in `f.m`) and the numerical method (in `eulerstep.m` or `midptstep.m`) from the main program. In Fig. 1.2, we show the numerical solution $u(t)$ for (1.3-1.4) calculated by the midpoint

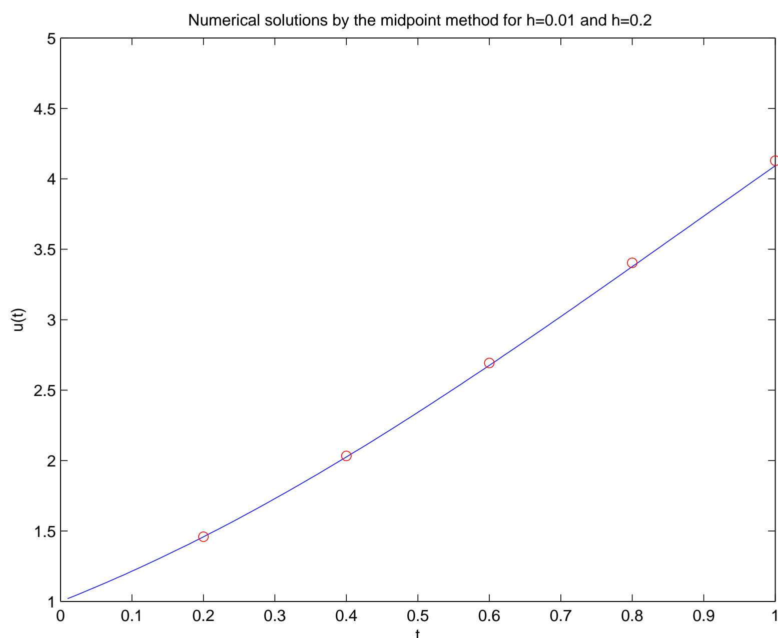

Figure 1.2: Numerical solutions by the midpoint method. The solid curve is for $h = 0.01$. The "o" is for $h = 0.2$.

method with $h = 0.01$ and $h = 0.2$. You can see that the midpoint solution obtained with $h = 0.2$ is much more accurate than the Euler's solution with the same h.

1.3 Local truncation error and order

When a numerical method is used to solve a differential equation, we want to know how accurate is the numerical solution. We will denote the exact solution as $y(t)$, thus $y(t_j)$ is the exact solution at t_j. The numerical solution at t_j is denoted as y_j, therefore, we are interested in the following error:

$$e_j = |y(t_j) - y_j|.$$

We do not expect to be able to know e_j exactly, because we do not have the exact solution in general. Therefore, we will be happy to have some estimates (such as approximate formulas or inequalities) for e_j. However, even this is not so easy. The reason is that the error accumulates. Let us look at the steps.

We start with $y_0 = y(t_0)$ which is exact, then we calculate y_1 which approximates $y(t_1)$, then we calculate y_2 which approximates $y(t_2)$, etc. Notice that when we calculate y_2, we use y_1, not $y(t_1)$. The numerical solution y_1 has some error, this error will influence y_2. Therefore, the error e_2 depends on e_1. Similarly, the error at the third step, i.e., e_3, depends on the error at step 2, etc. As a result, it is rather difficult to estimate e_j.

The numerical methods given in the previous sections can be written in the following general form:

$$y_{j+1} = \phi(t_j, h, y_j), \tag{1.21}$$

where ϕ is some function related to the function f which defines the differential equation. For example, the Euler's method is

$$\phi(t_j, h, y_j) = y_j + hf(t_j, y_j).$$

The midpoint method is

$$\phi(t_j, h, y_j) = y_j + hf\left(t_j + \frac{h}{2}, y_j + \frac{h}{2} f(t_j, y_j)\right).$$

If we have the exact solution, we can put the exact solution $y(t)$ into (1.21). That is, we replace y_j and y_{j+1} by $y(t_j)$ and $y(t_{j+1})$ in (1.21). When this is done, the two sides of (1.21) will not equal, so we should consider

$$T_{j+1} = y(t_{j+1}) - \phi(t_j, h, y(t_j)). \tag{1.22}$$

The above T_{j+1} is the so-called **local truncation error**. If we know the exact solution $y(t)$, then we can calculate T_j. In reality, we do not know the exact solution, but we can understand how T_{j+1} depends on step size h by studying the Taylor series of T_{j+1}. We are interested in the local truncation error because it can be estimated and it gives information on the true error. Therefore, we will try to do a Taylor series for T_{j+1} at t_j, assuming h is small. In fact, we only need to calculate the first non-zero term of the Taylor series:

$$T_{j+1} = Ch^{p+1} + \ldots$$

where the integer p is the **order** of the method, C is a coefficient that depends on t_j, $y(t_j)$, $y'(t_j)$, $f(t_j, y(t_j))$, etc. But C does not depend on the step size h. The above formula for T_{j+1} gives us information on how T_{j+1} varies with the step size h. Because h is supposed to be small, we notice that a larger p implies that $|T_{j+1}|$ will be smaller. Therefore, the method will be more accurate if p is larger.

We notice that $|T_1| = e_1$, because $y_0 = y(t_0)$, thus $y_1 = \phi(t_0, h, y_0) = \phi(t_0, h, y(t_0))$. However, it is clear that $|T_j| \neq e_j$ for $j > 1$.

When we try to work out the first non-zero term of the Taylor series of T_{j+1}, we work on the general equation (1.1). This is for the local truncation error at t_{j+1}. But the general case at t_{j+1} has no real difference with the special case at t_1. If we work out the Taylor series for T_1, we automatically know the result at T_{j+1}. The integer p (that is the order of the method) should be the same. In the coefficient C, we just need to replace t_0, $y(t_0)$, $f(t_0, y(t_0))$, ... by t_j, $y(t_j)$, $f(t_j, y(t_j))$, ...

Now, let us work out the local truncation error for Euler's method. The method is $y_{j+1} = y_j + hf(t_j, y_j) = \phi(t_j, h, y_j)$. Thus,

$$T_1 = y(t_1) - \phi(t_0, h, y(t_0)) = y(t_1) - y_1.$$

We have a Taylor expansion for $y(t_1)$ at t_0:

$$y(t_1) = y(t_0) + hy'(t_0) + \frac{h^2}{2}y''(t_0) + \ldots$$

Notice that $y'(t_0) = f(t_0, y(t_0))$. Therefore,

$$T_1 = \frac{h^2}{2}y''(t_0) + \ldots$$

The power of h is $p+1$ for $p = 1$. Therefore, the Euler's method is a first order method.

We can show that the local truncation error of the general 2nd order Runge-Kutta methods is

$$T_1 = \frac{h^3}{4}\left[\left(\frac{2}{3} - \alpha\right)y''' + \alpha y''\frac{\partial f}{\partial y}\right]_{t=t_0} + \ldots$$

As an example, we prove the result for the midpoint method ($\alpha = 1/2$). The local truncation error is

$$T_1 = h^3\left[\frac{1}{24}y''' + \frac{1}{8}y''\frac{\partial f}{\partial y}\right]_{t=t_0} + O(h^4)$$

Proof: First, since the differential equation is $y' = f(t,y)$. We use the **chain rule** and obtain:

$$\begin{aligned}
y'' &= f_t + f_y y' = f_t + f f_y \\
y''' &= f_{tt} + f_{ty}y' + [f]'f_y + f[f_y]' = f_{tt} + f f_{ty} + [f_t + f_y y']f_y + f[f_{ty} + f_{yy}y'] \\
&= f_{tt} + 2f f_{ty} + f^2 f_{yy} + [f_t + f f_y]f_y = f_{tt} + 2f f_{ty} + f^2 f_{yy} + y'' f_y
\end{aligned}$$

Now for y_1 using the midpoint method, we have

$$\begin{aligned}
k_1 &= f(t_0, y_0) = y'(t_0) \\
k_2 &= f(t_0 + \frac{h}{2}, y_0 + \frac{h}{2}k_1) = f(t_0 + \frac{h}{2}, y_0 + \frac{h}{2}y'(t_0)).
\end{aligned}$$

Now, we need **Taylor expansion** for functions of two variables. In general, we have

$$\begin{aligned}
f(t+\delta, y+\Delta) &= f(t,y) + \delta f_t(t,y) + \Delta f_y(t,y) \\
&\quad + \frac{\delta^2}{2}f_{tt}(t,y) + \delta\Delta f_{ty}(t,y) + \frac{\Delta^2}{2}f_{yy}(t,y) + \ldots
\end{aligned}$$

Now, for k_2, apply the above Taylor formula and use f to denote $f(t_0, y_0) = y'(t_0)$, we have

$$\begin{aligned}
k_2 &= f + \frac{h}{2}f_t + \frac{h}{2}y' f_y + \frac{h^2}{8}f_{tt} + \frac{h^2 y'}{4}f_{ty} + \frac{h^2(y')^2}{8}f_{yy} + O(h^3) \\
&= y' + \frac{h}{2}y'' + \frac{h^2}{8}[y''' - y'' f_y] + O(h^3).
\end{aligned}$$

Here y, f and their derivatives are all evaluated at t_0. Notice that $y(t_0) = y_0$. Therefore,

$$y_1 = y + hk_2 = y + hy' + \frac{h^2}{2}y'' + \frac{h^3}{8}[y''' - y'' f_y] + O(h^4)$$

Use the Taylor expansion

$$y(t_1) = y(t_0 + h) = y + hy' + \frac{h^2}{2}y'' + \frac{h^3}{6}y''' + O(h^4)$$

and the definition for T_1, we have

$$T_1 = \frac{h^3}{6}y''' - \frac{h^3}{8}[y''' - y'' f_y] + O(h^4) = h^3\left[\frac{1}{24}y''' + \frac{1}{8}y'' f_y\right] + O(h^4).$$

1.4 Embedded Runge-Kutta methods

Some differential equations may have solutions that change rapidly in some time intervals and change relatively slowly in other time intervals. As an example, we consider the Van der Pol equation:

$$u'' + u = \mu(1-u^2)u', \quad t > 0.$$

For $\mu = 6$ and the initial conditions $u(0) = 1$, $u'(0) = 0$, we use the midpoint method with $h = 0.04$ and solve the equation from $t = 0$ to $t = 40$. The solution is given in Fig. 1.3. It appears that we should not

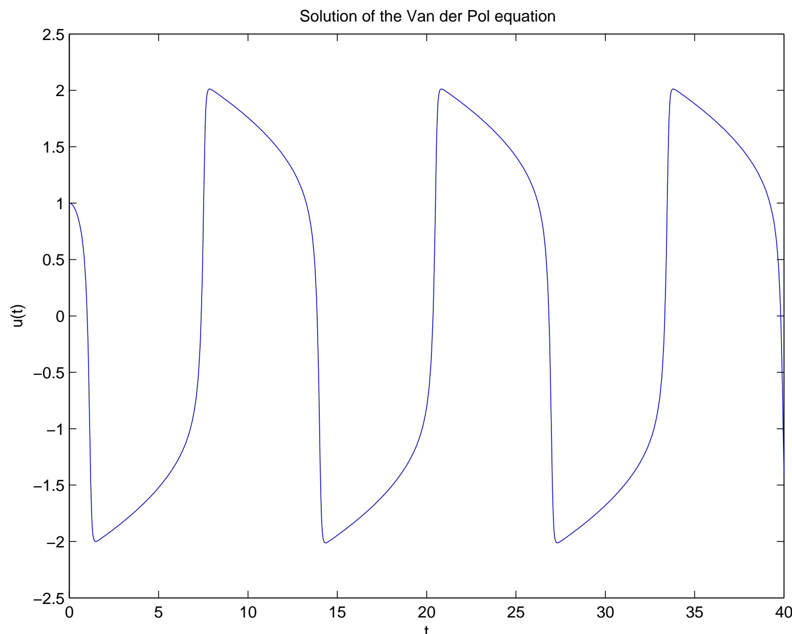

Figure 1.3: A solution of the Van der Pol equation for $\mu = 6$ and $u(0) = 1$, $u'(0) = 0$.

keep the step size h as a constant. Rather, we should only use a small h when the solution changes with time rapidly. A numerical method that automatically selects the step size in each step is an adaptive method.

A class of adaptive method for solving differential equations is the so-called embedded Runge-Kutta methods. An embedded Runge-Kutta method uses two ordinary Runge-Kutta methods for comparing the numerical solutions and selecting the step size. Moreover, the two methods in an embedded method typically share the evaluation of f (we are solving $y' = f(t,y)$). Therefore, the required computation effort is minimized.

Here is a 3rd order Runge-Kutta method

$$k_1 = f(t_j, y_j) \tag{1.23}$$

$$k_2 = f(t_j + h, y_j + hk_1) \tag{1.24}$$

$$k_3 = f(t_j + \frac{h}{2}, y_j + \frac{h}{4}(k_1 + k_2)) \tag{1.25}$$

$$y_{j+1} = y_j + \frac{h}{6}(k_1 + 4k_3 + k_2) \tag{1.26}$$

The cost of this method is mainly related to the calculation of k_1, k_2 and k_3. That is, three evaluations of f. With the above k_1 and k_2, we can use the 2nd order Runge-Kutta method ($\alpha = 1$, the modified Euler's method) to get a less accurate solution at t_{j+1}:

$$y^*_{j+1} = y_j + \frac{h}{2}(k_1 + k_2). \tag{1.27}$$

Although we are not going to use y^*_{j+1} as the numerical solution at t_{j+1}, we can still use y^*_{j+1} to compare with the 3rd order solution y_{j+1}. If their difference is too large, we reject the solution and use a smaller stepsize h to repeat the calculation. If their difference is small enough, we will accept y_{j+1}. But we also use this information to suggest a step size for the next step. A user must specify a small number ε (called the error tolerance) to control the error for selecting the step size. The difference between y_{j+1} and y^*_{j+1} is

$$e = ||y_{j+1} - y^*_{j+1}|| = \frac{h}{3}||k_1 - 2k_3 + k_2||. \tag{1.28}$$

Since y may be a vector, we have used a vector norm above.

To understand the formula for changing the step size, we consider the first step and the exact solution $y(t_1)$ at t_1. The local truncation errors give us

$$\begin{aligned} y(t_1) - y^*_1 &= C_1 h^3 + ... \\ y(t_1) - y_1 &= C_2 h^4 + ... \end{aligned}$$

for some C_1 and C_2 related to the solution at t_0, its derivatives, the function f and its partial derivatives at t_1. Thus, we have

$$y_1 - y^*_1 = C_1 h^3 + ...$$

Therefore,

$$e \approx ||C_1|| h^3. \tag{1.29}$$

Although we do not know C_1, the above relationship allows us to design a stepsize selection method based on the user specified error tolerance ε. If $e \le \varepsilon$, we accept y_1, otherwise, we reject y_1 and repeat this step. The current step size used for calculating y_1 and y^*_1 is h, how should we choose a new step size? We have

$$e_{new} \approx ||C_1|| h^3_{new} < \varepsilon.$$

Compare this with (1.29), we have

$$\frac{||C_1||\, h^3_{new}}{||C_1||\, h^3} < \frac{\varepsilon}{e}$$

or

$$h_{new} < h \left(\frac{\varepsilon}{e}\right)^{1/3}$$

To satisfy the above inequality, we use

$$h := 0.9h \left(\frac{\varepsilon}{e}\right)^{1/3} \tag{1.30}$$

to reset the stepsize. Now, if $e \le \varepsilon$, we accept $t_1 = t_0 + h$ and y_1, but we also use formula (1.30) to reset the stepsize. This gives rise to the possibility to increase the stepsize when the original h is too small (so that e is much smaller than ε).

Algorithm: to solve $y' = f(t, y)$ from t_0 to t_{end} with error tolerance ε and initial condition $y(t_0) = y_0$,

initialize $t = t_0$, $y = y_0$, ε, h (initial step size)
while $t < t_{end}$
    $k_1 = f(t, y)$
    $k_2 = f(t + h, y + hk_1)$
    $k_3 = f(t + h/2, y + \frac{h}{4}(k_1 + k_2))$
    $e = \frac{h}{3}||k_1 - 2k_3 + k_2||$
    if $e \le \varepsilon$, then
        $y = y + \frac{h}{6}(k_1 + 4k_3 + k_2)$
        $t = t + h$
        output t, y
    end if
    $h = 0.9h(\varepsilon/e)^{1/3}$
end

Notice that the formula for resetting h is outside the "if...end if" loop. That is, whether the calculation is accepted or not, h will always be changed.

As an example, we consider

$$y' = y - ty^2, \quad t > 0, \tag{1.31}$$

with initial condition $y(0) = 1$. If we use $\varepsilon = 10^{-5}$ and the initial step size $h = 0.5$, we get

$$k_1 = 1, \;\; k_2 = 0.375, \;\; k_3 \approx 0.8286, \;\; e \approx 0.047.$$

Since $e > \varepsilon$, this step is rejected. We have the new step size $h \approx 0.0269$ and

$$k_2 \approx 0.9985, \;\; k_3 \approx 0.9996, \;\; e \approx 6.4194 \times 10^{-5}.$$

Thus, the new step is accepted and

$$t_1 \approx 0.0269, \;\; y_1 \approx 1.0268.$$

The numerical solution of this differential equation is shown in Fig. 1.4. A MATLAB program (`erk23.m`) for the embedded Runge-Kutta method is given below.

```
function [tout, yout] = erk23(t0, tfinal, y0, tiny, h0)
% This is the embedded Runge-Kutta method using a 3rd order
% Runge-Kutta method and a 2nd order Runge-Kutta method.
% We are solving y' = f(t, y), where y is a column vector
% of functions of t.
% Input: t0, the initial time
%        tfinal, the final time
%        y0, a column vector of the initial conditions, i.e., y(t0) = y0.
%        tiny, the small parameter for error tolerance
%        h0, initial time step
% Output: tout, a row vector for the discrete time steps
```

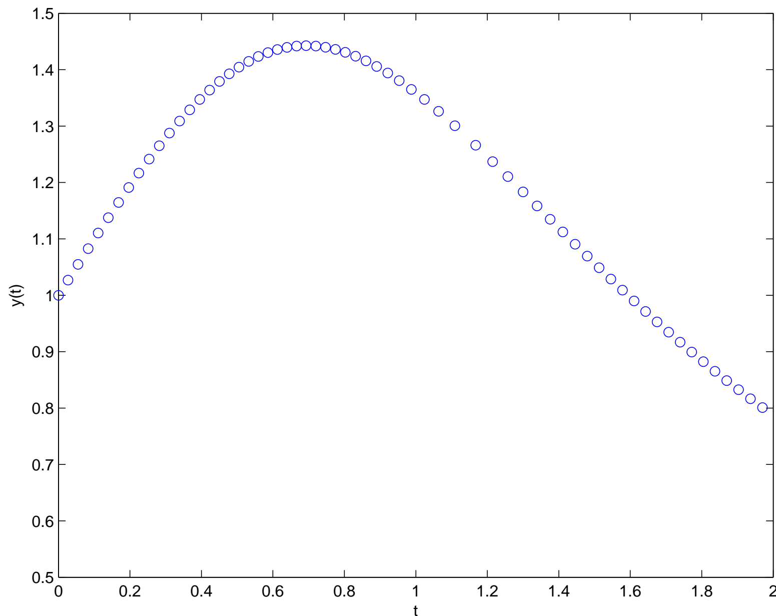

Figure 1.4: Numerical solution of (1.31) by embedded Runge-Kutta method.

```
%          yout, a matrix for solutions of y at various various time.
  t = t0;
  y = y0;
  h = h0;
  tout = t0;
  yout = y0;
  while t < tfinal
    k1 = f(t, y);
    k2 = f(t+h, y + h*k1);
    k3 = f(t+h/2, y + h*(k1+k2)/4);
    E = (h/3)*norm(k1-2*k3 +k2);
    if E <= tiny
      y = y + (h/6)*(k1 + 4*k3 + k2);
      t = t + h;
      tout = [tout, t];
      yout = [yout, y];
    end
    h = 0.9 * h * (tiny/E)^(1/3);
  end
```

This program requires `f.m` which specifies the differential equation.

Chapter 2

ODE IVP: Implicit One-step Methods

2.1 Stiff equations

The Euler's method and the Runge-Kutta methods in previous sections are **explicit** methods. For the step from t_j to t_{j+1}, the numerical solution y_{j+1} has an **explicit** formula. In the section on local truncation errors, we write down such a formula by $y_{j+1} = \phi(t_j, h, y_j)$. It turns out that the explicit methods have some difficulties for some differential equations. These are the so-called **stiff** differential equations.

First, we consider the following example:

$$y' + \sin(t) = -200(y - \cos(t)), \tag{2.1}$$

with initial condition $y(0) = 0$. The exact solution is

$$y(t) = \cos(t) - e^{-200t}.$$

As t increases, the exact solution converges to $\cos(t)$ rapidly. Let us use the Euler's method for this equation. The numerical solutions are obtained with $h = 0.008$ and $h = 1/99$ in Fig. 2.1. We observe that the numerical solution looks reasonable for large t if $h = 0.008$. There are large errors at the first a few steps, then the error decrease rapidly. In fact, this is true for $h < 0.01$. If $h = 1/99$, we can see that the error oscillates and grows exponentially in t. If we replace the Euler's method by a higher order Runge-Kutta (explicit) method, we still have similar difficulties.

While the above example appears to be a toy problem, we also have more realistic examples. Let us consider the heat equation

$$u_t = u_{xx}, \quad 0 < x < L, \quad t > 0. \tag{2.2}$$

This is one of the simplest partial differential equations (PDEs) and it will be studied again in Chapter 6. This equation is usually solved with two boundary conditions and one initial condition. For example, we have

$$u(0,t) = a \quad \text{and} \quad u(L,t) = b, \quad t \geq 0,$$
$$u(x,0) = f(x), \quad 0 < x < L,$$

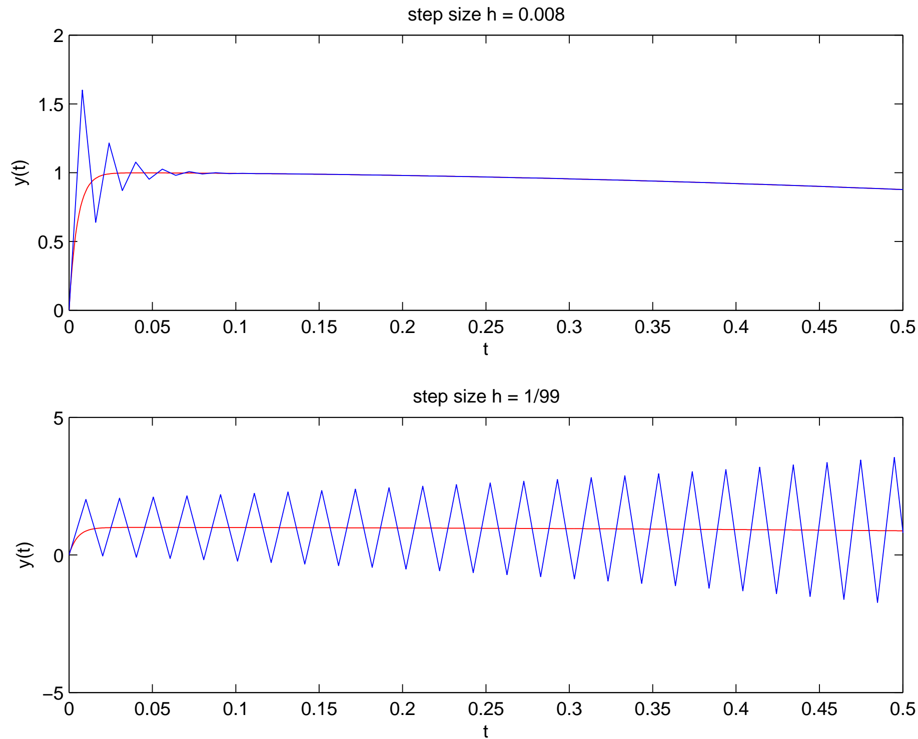

Figure 2.1: Comparison of the exact and the numerical solutions of equation (2.1). The numerical solutions are obtained by the Euler's method

where f is a given function of x. We can solve this *initial and boundary value problem* by separation of variables. We have

$$u(x,t) = u_\infty(x) + \sum_{j=1}^{\infty} \hat{g}_j \sin\left(\frac{j\pi x}{L}\right) e^{-(j\pi/L)^2 t},$$

where

$$u_\infty(x) = a + (b-a)\frac{x}{L}, \quad \hat{g}_j = \frac{2}{L}\int_0^L [f(x) - u_\infty(x)] \sin\left(\frac{j\pi x}{L}\right) dx.$$

Notice that the solution converges rapidly to the time-independent (steady) solution u_∞ as $t \to \infty$. The steady solution is determined by the boundary conditions only and it is a linear function of x. In Chapter 6, we will study a number of numerical methods for this equation. For the moment, we will use a simple method to discretize the variable x and approximate this PDE by a system of ODEs. We discretize x by

$$x_i = i\Delta x \quad \text{for} \quad i = 0, 1, 2, ..., m+1 \quad \text{and} \quad \Delta x = \frac{L}{m+1},$$

denote $u_i = u(x_i, t)$ and approximate $u_{xx}(x_i, t)$ by

$$u_{xx}(x_i, t) \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{(\Delta x)^2},$$

then the heat equation is approximated by the following system of ODEs:

$$\frac{d}{dt}\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{m-1} \\ u_m \end{bmatrix} = \frac{1}{(\Delta x)^2}\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{m-1} \\ u_m \end{bmatrix} + \frac{1}{(\Delta x)^2}\begin{bmatrix} a \\ 0 \\ \vdots \\ 0 \\ b \end{bmatrix}. \tag{2.3}$$

Since only x is discretized, we call the above approximation a **semi-discretization**. Originally, the heat equation is defined on the two-dimensional domain $\{(x,t)|0<x<L, t>0\}$, now we are approximating u only on the lines: $x = x_i$ for $t > 0$. We call such a process that turns PDE to a system of ODEs the **method of lines**. In the following, we let $L = 1$, $a = 1$, $b = 2$ and $f(x) = 0$ for $0 < x < L$, and solve the above system by the 4th order classical Runge-Kutta method. The right hand side of the ODE system is given in `f.m`:

```
function k=f(t,u)
% ODE system for semi-discretized heat equation.
% u_t = u_xx, 0<x<L, t>0,
% u=a at x=0, u=b at x=L.
global L a b
m = length(u);
dx = L/(m+1);
s = 1/(dx)^2;
k(1)=s*(a-2*u(1)+u(2));
k(m)=s*(u(m-1)-2*u(m)+b);
k(2:m-1)=s*(u(1:m-2)-2*u(2:m-1)+u(3:m));
```

The 4th order classical Runge-Kutta method is given in `rk4step.m`:

```
function y1 = rk4step(h,t0,y0);
k1 = f(t0,y0);
k2 = f(t0+h/2, y0 + (h/2)*k1);
k3 = f(t0+h/2, y0 + (h/2)*k2);
k4 = f(t0+h, y0+ h*k3);
y1 = y0 + (h/6)*(k1+2*k2+2*k3+k4);
```

The main program is given below.

```
global L a b
L=1; a=1; b=2;
% discretizing x by m points in (0,L)
m = 99;
dx = L/(m+1);
x = dx*(1:m);
% simple initial condition u = 0.
```

```
u = zeros(1,m);
% solve from t=0 to t=0.05 with nsteps
tzero = 0; tfinal = 0.05;
nsteps = 718;   % try 717, 716
h = (tfinal - tzero)/nsteps
for j=1:nsteps
    t = (j-1)*h;
    u = rk4step(h,t,u);
end
% draw the solution at t=tfinal
plot([0,x,L],[a,u,b])
```

We have tried two steps $h = 0.05/718 \approx 6.964 \times 10^{-5}$ and $h = 0.05/716 \approx 6.983 \times 10^{-5}$. The smaller h gives a satisfactory solution, while the larger h gives an incorrect solution with wild oscillations. Compared with the grid size $\Delta x = 0.01$, the time step size $h = 6.964 \times 10^{-5}$ appears to be extremely

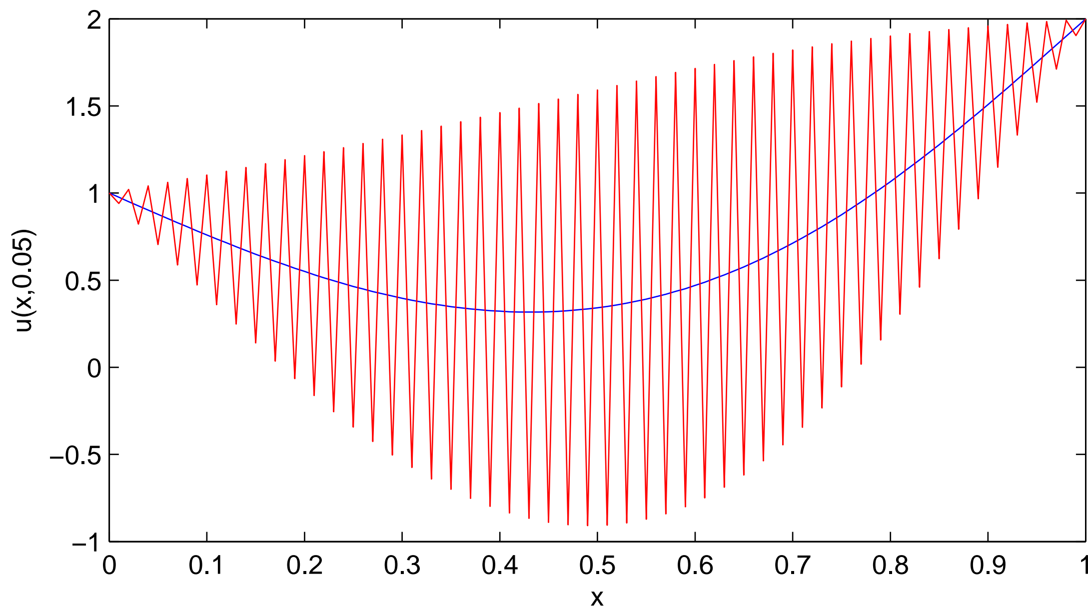

Figure 2.2: Numerical solutions of the heat equation at $t = 0.05$ by a 4th order Runge-Kutta method with step size $h = 0.05/718$ and $h = 0.05/716$.

small.

The concept of "stiff" differential equation is not rigorously defined. Suppose we have an exact solution $y(t)$ of a differential equation. Fix a time t_*, the differential equation has infinitely many other solutions for $t > t_*$. If these other solutions converge to $y(t)$ rapidly for $t > t_*$, then we may say that this differential equation is stiff at t_*. If a nearby solution, say $\tilde{y}(t)$, converges to $y(t)$ rapidly, the derivative of $\tilde{y}$ can be large (in absoluate value). Numerically, this can be difficult to catch. For explicit method, the error can decrease if the step size is sufficiently small. But the error may increase exponentially, if the step size is not small enough.

2.2 Implicit one-step methods

For stiff differential equations, we need the "implicit" methods. One step implicit methods can be written as

$$y_{j+1} = \phi(t_j, h, y_j, y_{j+1}). \tag{2.4}$$

Notice that y_{j+1} is what we want to calculate, but it also appears in the right hand side. To be more precise, the y_{j+1} in the right hand side only appears inside the function f. Remember that we are trying to solve $y' = f(t, y)$. The method is called implicit, because we do not have an explicit formula for y_{j+1}. Instead, we have to solve an equation to find y_{j+1}. If the differential equation is complicated, an implicit method can be very difficult to use.

When applied to stiff differential equations, the implicit methods behave better. We can use a large step size. Of course, if the step size is large, the numerical solution may be not very accurate. But at least the error is under control. Next, we list some one step implicit methods.

Backward Euler's method:

$$y_{j+1} = y_j + hf(t_{j+1}, y_{j+1}). \tag{2.5}$$

This is a first order implicit method. Notice that y_{j+1} also appears in the right hand side in f.

Trapezoid method:

$$y_{j+1} = y_j + \frac{h}{2}\left[f(t_j, y_j) + f(t_{j+1}, y_{j+1})\right] \tag{2.6}$$

This is one of the most widely used implicit method. It is a 2nd order method.

Implicit midpoint method:

$$y_{j+1} = y_j + hf\left(t_j + \frac{h}{2}, \frac{1}{2}(y_j + y_{j+1})\right) \tag{2.7}$$

Again, this is a 2nd order implicit method. Notice that y_{j+1} also appears in the right hand side. The implicit midpoint method is equavelent to the so-called **2nd order Gauss method**:

$$k_1 = f\left(t_j + \frac{h}{2}, y_j + \frac{h}{2}k_1\right) \tag{2.8}$$

$$y_{j+1} = y_j + hk_1. \tag{2.9}$$

This time, k_1 is implicitly given in the first equation. If we eliminate k_1, the method can still be written as (2.10).

Now, let us solve the differential equation (2.1) by the implicit midpoint method. Using the step size $h = 0.02$, we obtain the numerical results shown as the little circles in Fig. 2.3. Besides the first a few steps, the numerical solutions are pretty accurate.

The implicit methods given in this section are one-step method, since y_{j+1} depends on y_j only (does not depend on earlier solutions, such as y_{j-1}). The local truncation error and the order of the method can be defined as before. For the general implicit method (2.10), we first calculate $\tilde{y}_{j+1}$ by changing y_j to the exact solution $y(t_j)$, i.e.,

$$\tilde{y}_{j+1} = \phi(t_j, h, y(t_j), \tilde{y}_{j+1}),$$

then the local truncation error is

$$T_{j+1} = y(t_{j+1}) - \tilde{y}_{j+1}.$$

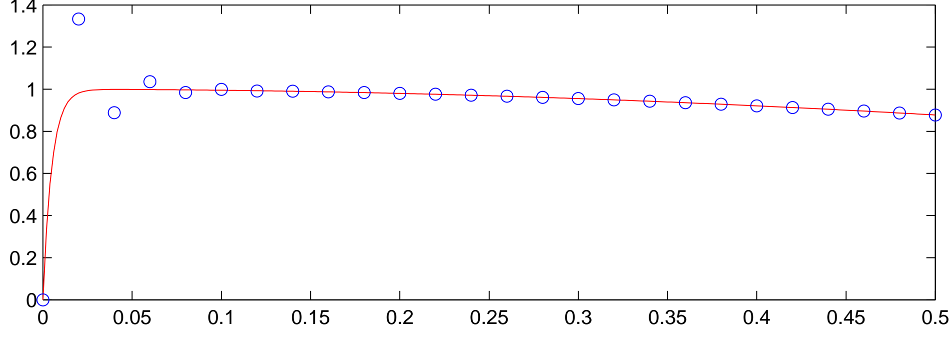

Figure 2.3: Comparision of the exact solution and the numerical solution by the implicit midpoint method with $h = 0.02$ for (2.1).

This definition is somewhat complicated, since $\tilde{y}_{j+1}$ must be solved from an equation. Notice that a one-step implicit method can be written as

$$y_{j+1} = \phi(t_j, h, y_j, y_{j+1}) = y_j + h\Phi(t_j, y_j, y_{j+1}), \tag{2.10}$$

where Φ is related to f. We can approximate $\tilde{y}_{j+1}$ by

$$\hat{y}_{j+1} = \phi(t_j, h, y(t_j), y(t_{j+1})) = y(t_j) + h\Phi(t_j, y(t_j), y(t_{j+1})).$$

Notice that $\hat{y}_{j+1}$ is given explicitly. This gives rise to the modified definition of local truncation error:

$$\hat{T}_{j+1} = y(t_{j+1}) - \hat{y}_{j+1} = y(t_{j+1}) - \phi(t_j, h, y(t_j), y(t_{j+1})).$$

It appears that we just need to insert the exact solution into the numerical formula to get the local truncation error. It can be proved that the original and the modified definitions give the same first non-zero term in their Taylor expansions. That is, if we assume the time step h is small and work out the first a few terms of the Taylor expansions, then

$$\begin{aligned} T_{j+1} &= Ch^{p+1} + Dh^{p+2} + \dots \\ \hat{T}_{j+1} &= Ch^{p+1} + \hat{D}h^{p+2} + \dots \end{aligned}$$

Since we are only interested in the first non-zero term of the local truncation error, we can use $\hat{T}_{j+1}$ to replace the original T_{j+1}. As before, the method has order p, if the first non-zero term of the Taylor series of the local truncation error is proportional to h^{p+1}. For example, the local truncation error of the backward Euler's method is

$$\hat{T}_{j+1} = y(t_{j+1}) - y(t_j) - hf(t_{j+1}, y(t_{j+1})) = y(t_{j+1}) - y(t_j) - hy'(t_{j+1}) = -\frac{h^2}{2}y''(t_j) + \dots$$

Therefore, the backward Euler's method is a first order method.

Chapter 3

ODE IVP: Multi-step Methods

3.1 Explicit multi-step methods

In Runge-Kutta methods, the solution at t_{j+1} is based on the solution at t_j. That is y_{j+1} is calculated from y_j. In multi-step methods, y_{j+1} is calculated from the solutions y_j, y_{j-1}, etc. Multi-step methods are more difficult to program, but they are more efficient than the Runge-Kutta methods. To present the multi-step methods, we need the following notation:

$$f_k = f(t_k, y_k)$$

for any integer k. The Adams-Bashforth methods are the most widely used explicit multi-step methods:
AB2 (2nd order)

$$y_{j+1} = y_j + h\left[\frac{3}{2}f_j - \frac{1}{2}f_{j-1}\right] \tag{3.1}$$

AB3 (3rd order)

$$y_{j+1} = y_j + h\left[\frac{23}{12}f_j - \frac{16}{12}f_{j-1} + \frac{5}{12}f_{j-2}\right]. \tag{3.2}$$

AB4 (4th order)

$$y_{j+1} = y_j + h\left[\frac{55}{24}f_j - \frac{59}{24}f_{j-1} + \frac{37}{24}f_{j-2} - \frac{9}{24}f_{j-3}\right] \tag{3.3}$$

The Adams-Bashforth methods are derived in the following steps.

1. The following formula is exact

$$\int_{t_j}^{t_{j+1}} y'(t)\, dt = y(t_{j+1}) - y(t_j) = \int_{t_j}^{t_{j+1}} f(t, y(t))\, dt$$

2. Find a polynomial interpolation for f based on the points

$$(t_j, f_j),\ \ (t_{j-1}, f_{j-1}), \ldots$$

3. Replace f by its polynomial aproximation in step 2, then integrate.

The method AB2 is a 2-step method, and it is also a second order method. The concept of order is related to the concept of local truncation error (LTE). If we write an explicit multi-step method as

$$y_{j+1} = \phi(t_j, h, y_j, y_{j-1}, ...)$$

for some function ϕ related to the differential equation $y' = f(t, y)$, the LTE is defined as

$$T_{j+1} = y(t_{j+1}) - \phi(t_j, h, y(t_j), y(t_{j-1}), ...)$$

where $y(t)$ is the exact solution of differential equation. As an example, we consider the method AB2. We have

$$\begin{aligned}\phi(t_j, h, y(t_j), y(t_{j-1})) &= y(t_j) + h\left[\frac{3}{2}f(t_j, y(t_j)) - \frac{1}{2}f(t_{j-1}, y(t_{j-1}))\right] \\ &= y(t_j) + h\left[\frac{3}{2}y'(t_j) - \frac{1}{2}y'(t_{j-1})\right]\end{aligned}$$

To find the order, we need the first non-zero term of the Taylor series of the LTE. Since

$$y'(t_{j-1}) = y'(t_j - h) = y'(t_j) - hy''(t_j) + \frac{h^2}{2}y'''(t_j) + ...$$

we obtain

$$\phi(t_j, h, y(t_j), y(t_{j-1})) = y(t_j) + hy'(t_j) + \frac{h^2}{2}y''(t_j) - \frac{h^3}{4}y'''(t_j) + ...$$

On the other hand,

$$y(t_{j+1}) = y(t_j + h) = y(t_j) + hy'(t_j) + \frac{h^2}{2}y''(t_j) + \frac{h^3}{6}y'''(t_j) + ...$$

Therefore,

$$T_{j+1} = \frac{5h^3}{12}y'''(t_j) + ...$$

If the LTE is related to h as

$$T_{j+1} = Ch^{p+1} + ...$$

then the order of the numerical method is p. For AB2, the order is 2. In other words, AB2 is a second order method.

Let us consider the method AB4 (which is a fourth order method). To evaluate y_{j+1}, we need f_j, f_{j-1}, f_{j-2} and f_{j-3}. However, we only need to calculate f_j in this step, because the other three values of f (i.e. f_{j-1}, f_{j-2} and f_{j-3}) are already calculated in the previous steps. In comparision, in each step of the 4th order Runge-Kutta method, we need to evaluate f four times (i.e., k_1, k_2, k_3 and k_4). For large scale problems, usually the most expensive part of the calculation is to evaluate the function f. Therefore, AB4 is roughly four times faster than a 4th order Runge-Kutta method. A MATLAB program for the method AB4 is given below:

```
%  We implement the 4th order Adams-Bashforth's method here. A
%  constant step size h is used. The differential equation is y' = f(t,y),
%  where f is the name  (a string) of the function f(t,y). Notice that y
```

```
%  and f are supposed to be column vectors.
%  Input:
%       t0 --- the intial time
%       y0 --- the initial values (a column vector)
%       tfinal --- the final time
%       steps --- the total number of steps.
%  Output:
%       t --- a row vector for the discretized time
%       y --- a matrix for solutions at various time

function [t, y] = myab4(t0, y0, tfinal, f, steps)
%  setup the step size.
h = (tfinal  - t0)/steps;
%  setup the vector for output.
n = length(y0);
t = t0 : h: tfinal;
y = zeros(n, steps+1);
y(:,1) = y0;
% first 3 steps by the classical 4th order Runge-Kutta method.
[y(:,2), f1]   = myrk4a(f, h, y(:,1), t(1));
[y(:,3), f2]   = myrk4a(f, h, y(:,2), t(2));
[y(:,4), f3]   = myrk4a(f, h, y(:,3), t(3));
%  calculate the remaining steps by AB4
for j=4:steps
   f4 = feval(f, t(j), y(:,j));
   y(:,j+1) = y(:,j) + (h/24)*(-9*f1 + 37*f2-59*f3+55*f4);
   f1 = f2;
   f2 = f3;
   f3 = f4;
end

% The 4th order classical Runge-Kutta method
function [y1, k1] = myrk4a(f, h, y, t)
     k1 = feval(f, t, y);
     k2 = feval(f, t+0.5*h, y+0.5*h*k1);
     k3 = feval(f, t+0.5*h, y+0.5*h*k2);
     k4 = feval(f, t+h, y+h*k3);
     y1 = y + (h/6) * (k1 + 2*k2 + 2*k3 + k4);
```

Next, we use AB4 to solve the following Lorenz system:

$$y_1' = 10(y_2 - y_1)$$

$$y_2' = -y_1y_3 + 28y_1 - y_2$$
$$y_3' = y_1y_2 - \frac{8}{3}y_3.$$

This is implemented in the following MATLAB program `lorenz.m`:

```
% The Lorenz system
function k = lorenz(t,y)
  k = zeros(3,1);
  k(1) = 10 * (y(2) - y(1));
  k(2) = - y(1)*y(3) + 28*y(1) - y(2);
  k(3) = y(1) * y(2) - (8/3) * y(3);
```

Now, we solve the Lorenz system from $t = 0$ to $t = 40$ with the following main program.

```
% the main program to solve the Lorenz system by AB4.
% initial time
t0 = 0;
% final time
tfinal = 40;
% initial conditions (column vector):
y0 = [-11.3360, -16.0335, 24.4450]' ;
% total number of steps
steps = 2000;
% call the function myab4
[t, y] = myab4(t0, y0, tfinal, 'lorenz', steps);
```

The solutions are plotted in Fig. 3.1. The Lorenz system exhibits **chaos**. If you think about the solution as a trajectory in the 3-D space of y_1, y_2 and y_3, then the trajectory does not approach a fixed point or a closed loop. If the trajectory approaches a fixed point, the solutions (as functions of t) tend to constants. If the trajectory approaches a closed loop, then the solutions become periodic as $t \to \infty$. But the solutions of the Lorenz equation is non-periodic as $t \to \infty$.

3.2 Implicit multi-step methods

The Adams-Bashforth methods, like the explicit Runge-Kutta methods, have difficulties for stiff differential equations. Some one-step implicit methods are introduced in Chapter 2. In the following, we develop some implicit multi-step methods.

The Adams-Moulton methods are implicit multi-steps methods and they are derived similarly as the Adams-Bashforth methods. We start with

$$y(t_{j+1}) - y(t_j) = \int_{t_j}^{t_{j+1}} f(t, y(t))\, dt$$

and use polynomial interpolation for f. For Adams-Moulton methods, we include (t_{j+1}, f_{j+1}) as an interpolation point. If we only use two points: (t_j, f_j) and (t_{j+1}, f_{j+1}) for approximating f, then we get

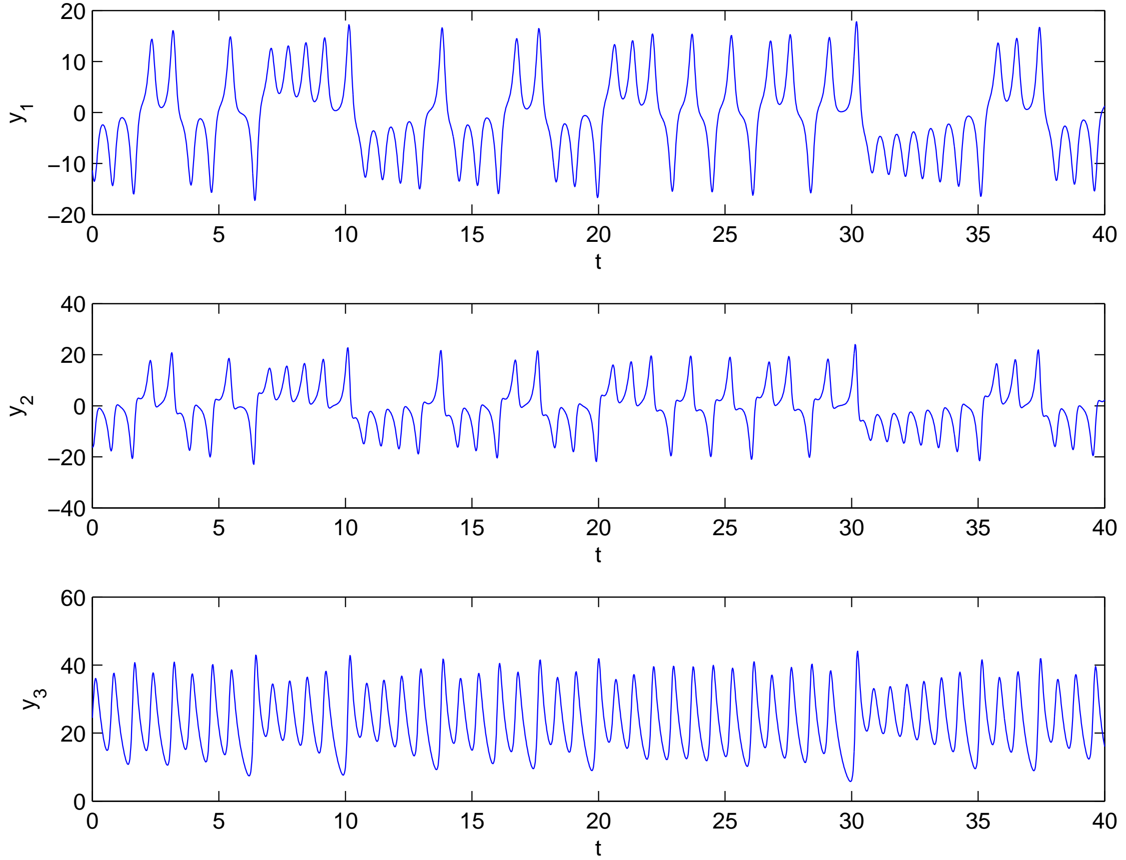

Figure 3.1: Solutions of the Lorenz system by AB4.

a single-step implicit method. This is the Trapezoid method given in section 2.2 (which can be regarded as AM1).

If f is approximated by its polynomial interpolation using the three points:

$$(t_{j+1}, f_{j+1}),\ (t_j, f_j),\ (t_{j-1}, f_{j-1})$$

we get the following 2-step Adams-Moulton method (**AM2**):

$$y_{j+1} = y_j + h\left[\frac{5}{12}f_{j+1} + \frac{8}{12}f_j - \frac{1}{12}f_{j-1}\right]. \tag{3.4}$$

The above is a 3rd order method. The 3-step Adams-Moulton method (**AM3**) has a fourth order of accuracy. It is given as follows:

$$y_{j+1} = y_j + h\left[\frac{9}{24}f_{j+1} + \frac{19}{24}f_j - \frac{5}{24}f_{j-1} + \frac{1}{24}f_{j-2}\right].$$

The Admas-Moulton methods are useful when they are used together with the Admas-Bashforth methods as the so-called **Predictor-Corrector** methods. In this a method, an explicit method is used to calculate a solution at t_{j+1}, say $\tilde{y}_{j+1}$, then it is improved by an implicit method. In the implicit method,

f_{j+1} is replaced by $f(t_{j+1}, \tilde{y}_{j+1})$. Here is the 3rd order Adams predictor-corrector method

$$\tilde{y}_{j+1} = y_j + h\left[\frac{23}{12}f_j - \frac{16}{12}f_{j-1} + \frac{5}{12}f_{j-2}\right] \tag{3.5}$$

$$y_{j+1} = y_j + h\left[\frac{5}{12}f(t_{j+1}, \tilde{y}_{j+1}) + \frac{8}{12}f_j - \frac{1}{12}f_{j-1}\right]. \tag{3.6}$$

Overall, this is still an explicit method. Notice that the two methods involed in the above predictor-corrector method both are 3rd order and the resulting method is also 3rd order.

A class of useful implicit multi-step method is the **Backward Differentiation Formulas (BDF)**. The derivation is as follows:

1. Write down a polynomial $Q(t)$ that interpolates

$$(t_{j+1}, y_{j+1}),\ (t_j, y_j),\ (t_{j-1}, y_{j-1}), \ldots$$

   BDF2 is a 2-step method, so Q is based on the above three points. BDF3 is a 3-step method, so (t_{j-2}, y_{j-2}) is also needed.

2. Replace $y' = f(t, y)$ at t_{j+1} by

$$Q'(t_{j+1}) = f(t_{j+1}, y_{j+1})$$

Consider BDF2, we have

$$Q(t) = y_{j+1}\frac{(t-t_j)(t-t_{j-1})}{(t_{j+1}-t_j)(t_{j+1}-t_{j-1})} + y_j\frac{(t-t_{j+1})(t-t_{j-1})}{(t_j-t_{j+1})(t_j-t_{j-1})} + y_{j-1}\frac{(t-t_j)(t-t_{j+1})}{(t_{j-1}-t_j)(t_{j-1}-t_{j+1})}$$

Take a derivative and set $t = t_{j+1}$, we get

$$\frac{3}{2h}y_{j+1} - \frac{2}{h}y_j + \frac{1}{2h}y_{j-1} = f(t_{j+1}, y_{j+1})$$

or

$$y_{j+1} - \frac{4}{3}y_j + \frac{1}{3}y_{j-1} = \frac{2h}{3}f(t_{j+1}, y_{j+1}).$$

The method **BDF3** can be similarly derived. We have

$$y_{j+1} - \frac{18}{11}y_j + \frac{9}{11}y_{j-1} - \frac{2}{11}y_{j-2} = \frac{6}{11}hf(t_{j+1}, y_{j+1}).$$

For an implicit multi-step method, given as

$$y_{j+1} = \phi(t_j, h, y_{j+1}, y_j, y_{j-1}, \ldots),$$

the local truncation error is defined as

$$T_{j+1} = y(t_{j+1}) - \phi(t_j, h, y(t_{j+1}), y(t_j), y(t_{j-1}), \ldots).$$

Going through a Taylor series, we may find

$$T_{j+1} = Ch^{p+1} + \ldots$$

then the order of the method is p.

Chapter 4

ODE IVP: Stability Concepts

4.1 Zero stability

There are many other multi-step methods. Some of them have higher order than the methods in the previous chapter (for the same number of steps). The following explicit multi-step method

$$y_{j+1} + 4y_j - 5y_{j-1} = h[4f_j + 2f_{j-1}] \tag{4.1}$$

is a third order 2-step method. This can be verified by calculating its local truncation error. We have

$$\begin{aligned} T_{j+1} &= y(t_{j+1}) + 4y(t_j) - 5y(t_{j-1}) - h\left[4f(t_j, y(t_j)) + 2f(t_{j-1}, y(t_{j-1}))\right] \\ &= y(t_{j+1}) + 4y(t_j) - 5y(t_{j-1}) - h\left[4y'(t_j) + 2y'(t_{j-1})\right]. \end{aligned}$$

Now, if we insert the Taylor series of $y(t_{j+1})$, $y(t_{j-1})$ and $y'(t_{j-1})$ at $t = t_j$, we obtain

$$T_{j+1} = \frac{h^4}{6} y^{(4)}(t_j) + \ldots$$

Since the power of h is $4 = 3 + 1$, this is a third order method.

Notice that the 2-step AB2 is only a second order method. It thus seems that the above method would be more useful than AB2. This is not the case, since it can not solve the simplest differential equation

$$y' = 0$$

The solution of the above should be $y = \text{const}$. If the initial condition is $y(t_0) = y_0$, then $y(t) = y_0$ for all t. For the 2-step method (4.1), we must assume that $y_1 \approx y(t_1)$ is also given. We assume that

$$y_1 \approx y_0, \quad \text{but} \quad y_1 \neq y_0$$

This can happen, if we have somehow computed y_1 with a small error. Then (4.1) is simple

$$y_{j+1} + 4y_j - 5y_{j-1} = 0$$

for $j = 1, 2, 3, \ldots$. This linear recurrence relationship can be solved. The general solution is

$$y_j = C_1 \lambda_1^j + C_2 \lambda_2^j,$$

where C_1 and C_2 are constants, λ_1 and λ_2 are the solutions of

$$\lambda^2 + 4\lambda - 5 = 0.$$

Therefore,

$$\lambda_1 = 1, \quad \lambda_2 = -5.$$

Thus the general solution is

$$y_j = C_1 + C_2(-5)^j.$$

Now, we can try to determine C_1 and C_2 from

$$\begin{aligned} y_0 &= C_1 + C_2 \\ y_1 &= C_1 - 5C_2. \end{aligned}$$

We have

$$C_1 = \frac{5y_0 + y_1}{6}, \quad C_2 = \frac{y_0 - y_1}{6}.$$

If $y_0 \neq y_1$, then $C_2 \neq 0$, thus

$$\lim_{j\to\infty} |y_j| = \infty.$$

Therefore, the error grows exponentially fast and the method (4.1) is useless.

Let us write a numerical method for ODE IVPs in the following general form:

$$\sum_{l=0}^{k} \alpha_l y_{j+l} = \alpha_k y_{j+k} + \ldots + \alpha_1 y_{j+1} + \alpha_0 y_j = h\Phi(y_{j+k}, \ldots, y_{j+1}, y_j, t_j; h). \tag{4.2}$$

This is a k-step method. The right hand side is related to the function f, i.e., the right hand side of the ODE. In general, the above method is implicit, since y_{j+k} is also in the right hand side. Furthermore, we may require that $\alpha_k = 1$. Notice that we have shifted the subscripts, so that terms like y_{j-1} do not appear. In fact, method (4.1) is now written as

$$y_{j+2} + 4y_{j+1} - 5y_j = h[4f_{j+1} + 2f_j].$$

In any case, we may ask when the general method (4.2) is zero stable. If the method is applied to $y' = 0$, then we just have the left hand side:

$$\alpha_k y_{j+k} + \ldots + \alpha_1 y_{j+1} + \alpha_0 y_j = 0. \tag{4.3}$$

Consider a special solution $y_j = \zeta^j$, we obtain

$$\rho(\zeta) = \sum_{l=0}^{k} \alpha_l \zeta^l = 0.$$

For zero-stability, we require that all solutions of the linear recurrence (4.3) must be bounded for all j and for all initial conditions. Therefore, the roots of the polynomial $\rho(\zeta)$, i.e. the zeros of the polynomial $\rho(\zeta)$ or the solutions of the above equation, must satisfy

$$|\zeta| \leq 1, \quad \text{and} \quad |\zeta| = 1 \quad \text{only if } \zeta \text{ is a simple root.}$$

4.2 Absolute stability

For stiff differential equations, it is desirable to have **A-stable** numerical methods. A numerical method is called **A-stable** (which means absolutely stable), if *when it is applied to*

$$y' = ay, \quad t > 0, \quad y(0) = y_0,$$

where a is any complex number with Re$(a) < 0$, the numerical solution $y_j \to 0$ as $j \to \infty$, for any step size $h > 0$. Notice that the exact solution of the above equation

$$y(t) = y_0 e^{at} \to 0, \quad \text{as} \quad t \to \infty,$$

since $\text{Re}(a) < 0$. Therefore, the A-stable numerical methods have the correct behavior for large t. An explicit numerical method can never be A-stable. When an explicit method is applied to $y' = ay$, the numerical solution converges to zero if h is small enough, otherwise, the numerical solution diverges exponentially.

The implicit methods presented in Chapter 2 are all A-stable. When applied to $y' = ay$, the backward Euler's method gives

$$y_{j+1} = \frac{1}{1-ah} y_j,$$

or

$$y_j = \left[\frac{1}{1-ah}\right]^j y_0.$$

Since $\text{Re}(a) < 0$, $|1-ah| > 1$, thus, $y_j \to 0$ as $j \to \infty$. For the implicit midpoint method and the Trapzoid method, we get

$$y_{j+1} = \frac{1+ah/2}{1-ah/2}\, y_j.$$

Therefore,

$$y_j = \left[\frac{1+ah/2}{1-ah/2}\right]^j y_0.$$

Since $\text{Re}(a) < 0$, we have

$$\left|1-\frac{ah}{2}\right| > \left|1+\frac{ah}{2}\right|.$$

Therefore, $y_j \to 0$ as $j \to \infty$.

Other implicit methods may or may not be A-stable. In fact, the Adams-Moulton methods are not A-stable. Let us consider the third order method AM2. If we apply the method to $y' = ay$, where a is a complex constant with a negative real part, we get

$$(1-5s)y_{j+1} - (1+8s)y_j + sy_{j-1} = 0,$$

where $s = ah/12$. The general solution of the linear recurrence relationship is

$$y_j = A\lambda_1^j + B\lambda_2^j$$

where λ_1 and λ_2 satisfies

$$(1-5s)\lambda^2 - (1+8s)\lambda + s = 0.$$

If λ_1 or λ_2 satisfies $|\lambda| > 1$ for some s, then $|y_j| \to \infty$ as $j \to \infty$ for the given s (thus the given step size h). If that is the case, the method will not be A-stable. This is indeed the case for real $a < 0$ when $s < -1/2$ or $h > -6/a$. Therefore, the method is not A-stable.

The 2-step BDF formula (i.e., BDF2) is A-stable, but the k-step BDF formulae for $k \geq 3$ are not A-stable. Furthermore, for BDF formulae, we also need to check their zero-stability. It turns out that the BDF formulae are zero-stable only if the number of steps $k \leq 6$. Although the BDF formulae are not A-stable for $3 \leq k \leq 6$, they are $A(0)$**-stable**. A numerical method is called $A(0)$-stable, if *when it is applied to* $y' = ay$ *for any* **real** *and negative a, the numerical solution always satisfies* $y_j \to 0$ *as* $j \to \infty$ *for any step size* $h > 0$. It is clear that $A(0)$-stable is a weaker condition than A-stable. If a method is A-stable, then it is certainly $A(0)$-stable, but the reverse is not true. Notice that the A-stable condition checks the solution for all **complex** a with a real and negative imaginary part, but it includes a real a as a special case.

Actually, we can have more information if we calculate the **region of absolute stability** of a numerical method. This concept is again related to $y' = ay$ for complex a, but it is also related to the step size h. As we see from the earlier calculations, the numerical solution for this equation is closely related to $z = ah$. Therefore, we define the region of absolute stability as a region in the complex z plane, where $z = ah$. It is defined as those values of z such that the numerical solutions of $y' = ay$ satisfy $y_j \to 0$ as $j \to \infty$ for any initial conditions. For the explicit Runge-Kutta methods in Chapter 1, the Adams-Bashforth methods, the Adams-Moulton methods, and the BDF methods, we show the regions of absolute stability in the extra handout. What about the three implicit methods in Chapter 2? The backward Euler's method is identified as BDF1, the trapezoid method is identified as AM1, and the region of absolute stability for implicit midpoint method is identical to that of the trapezoid method. With this concept, we realize that a method is A-stable, if its region of absolute stability includes the left half of the complex z-plane, and a method is $A(0)$-stable, if its region of absolute stability includes the negative half of the real line in the complex z-plane. Furthermore, we can say that one method is more stable than the other method, if the first method has a larger absolute stability region. As a little exercise, we consider the interval of absolute stability on the real axis of z. For $y = ay$, the 4th order Runge-Kutta method gives

$$y_{j+1} = \left(1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24}\right) y_j$$

On the real axis $z = ah$ is real, the interval is thus defined as

$$\left|1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24}\right| < 1.$$

We solve the end points of the interval from

$$1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24} = \pm 1.$$

The case of 1 gives $z = 0$ and $z = -2.7853$, the case -1 has no real roots. Therefore, the interval on the real axis (of the region of absolute stability) is $-2.7853 < z < 0$.

While our numerical methods are designed for the general first order system $y' = f(t, y)$ where y is in general a vector, we only considered the absolute stability concept for $y' = ay$ where y is a scalar and

a is a constant. Therefore, it is natural to ask whether this concept is relevant. First, we consider the linear equations:

$$y' = Ay$$

where A is a square matrix and it is t-independent. In that case, the matrix A has eigenvalues λ_1, λ_2, ..., the corresponding right eigenvectors p_1, p_2, ..., and left eigenvectors w_1^T, w_2^T, ..., where ^{T} denotes the transpose operation. That is

$$Ap_j = \lambda_j p_j, \quad w_j^T A = \lambda_j w_j^T, \quad j = 1, 2, \ldots$$

As y is a column vector of functions, we can multiply the row vector w_j^T and obtain

$$w_j^T y' = w_j^T Ay = \lambda_j w_j^T y,$$

If we define a scalar function $g_j = w_j^T y$, then $g_j' = \lambda_j g_j$. This equation has the same form as the simple equation $y' = ay$ studied earlier. If we assume $\mathrm{Re}(\lambda_j) < 0$ for all j, then the analytic solution satisfies $y \to 0$ as $t \to \infty$. In order to have numerical solutions that converge to zero, we must make sure that $\lambda_j h$, for all j, are in the region of absolute stability. This type of argument goes though for the linear system of ODEs with an inhomogeneous term:

$$y' = Ay + b,$$

where b is a vector. This is exactly the semi-discretized form (2.3) of heat equation discussed in section 2.1. At that time, we did not explain why the method is stable for step size $h = 0.05/718$ and unstable for $h = 0.05/716$. We can explain this, if we calculate the eigenvalues of coefficient matrix in (2.3) and then consider the region of absolute stability of the 4th order Runge-Kutta method. Actually, since the eigenvalues are real, we only need to consider the intersection of the absolute stability region with the real axis. If the eigenvalues are λ_j (all real and negative), then the numerical method is stable if the step size h satisfies

$$|\lambda_j| h < 2.7853.$$

It turns out that the eigenvalues of the coefficient matrix in (2.3) are

$$\lambda_j = -\frac{4}{(\Delta x)^2} \sin^2 \frac{j\pi}{2(m+1)}, \quad j = 1, 2, \ldots, m.$$

The one with the largest absolute value is λ_m. For $m = 99$, $\Delta x = 0.01$, we have

$$\lambda_m \approx -39990.13.$$

Therefore, we need $-39990.13h < 2.7853$ or $h < 6.964968 \times 10^{-5}$. This is satisfied for $h = 0.05/718$ but not $h = 0.05/717$ or $h = 0.05/716$.

For the more general system $y' = f(t, y)$, the absolute stability concept is useful if we think of approximating $f(t, y)$ at any fixed time t_j by a linear system of ODEs using Taylor expansion. But the approximate linear system changes as t_j changes.

Chapter 5

ODE Boundary Value Problems

5.1 The shooting method

Consider a 2nd order ordinary differential equation with two boundary conditions

$$\begin{aligned} y'' &= f(x,y,y'), \quad a < x < b \\ y(a) &= \alpha \\ y(b) &= \beta, \end{aligned}$$

where a, b, α, β are given constants, y is the unknown function of x, f is a given function that specifies the differential equation. This is a two-point boundary value problem. An initial value problem (IVP) would require that the two conditions be given at the same value of x. For example, $y(a) = \alpha$ and $y'(a) = \gamma$. Because the two separate boundary conditions, the above two-point boundary value problem (BVP) is more difficult to solve.

The basic idea of "shooting method" is to replace the above BVP by an IVP. But of course, we do not know the derivative of y at $x = a$. But we can guess and then further improve the guess iteratively. More precisely, we treat $y'(a)$ as the unknown, and use secant method or Newton's method (or other methods for solving nonlinear equations) to determine $y'(a)$.

We introduce a function u, which is a function of x, but it also depends on a parameter t. Namely, $u = u(x;t)$. We use u' and u'' to denote the partial derivative of u, with respect to x. We want u to be exactly y, if t is properly chosen. But u is defined for any t, by

$$\begin{aligned} u'' &= f(x,u,u') \\ u(a;t) &= \alpha \\ u'(a;t) &= t. \end{aligned}$$

If you choose some t, you can then solve the above IVP of u. In general u is not the same as y, since $u'(a) = t \neq y'(a)$. But if t is $y'(a)$, then u is y. Since we do not know $y'(a)$, we determine it from the boundary condition at $x = b$. Namely, we solve t from:

$$\phi(t) = u(b;t) - \beta = 0.$$

If a solution t is found such that $\phi(t) = 0$, that means $u(b;t) = \beta$. Therefore, u satisfies the same two boundary conditions at $x = a$ and $x = b$, as y. In other words, $u = y$. Thus, the solution t of $\phi(t) = 0$ must be $t = y'(a)$.

If we can solve the IVP of u (for arbitrary t) analytically, we can write down a formula for $\phi(t) = u(b;t) - \beta$. Of course, this is not possible in general. However, without an analytic formula, we can still solve $\phi(t) = 0$ numerically. For any t, a numerical method for IVP of u can be used to find an approximate value of $u(b;t)$ (thus $\phi(t)$). The simplest method is to use the secant method.

$$t_{j+1} = t_j - \frac{t_j - t_{j-1}}{\phi(t_j) - \phi(t_{j-1})}\phi(t_j), \quad j = 1,2,3,...$$

For that purpose, we need two initial guesses: t_0 and t_1. We can also use Newton's method:

$$t_{j+1} = t_j - \frac{\phi(t_j)}{\phi'(t_j)}, \quad j = 0,1,2,...$$

We need a method to calculate the derivative $\phi(t)$. Since $\phi(t) = u(b;t) - \beta$, we have

$$\phi'(t) = \frac{\partial u}{\partial t}(b;t) - 0 = \frac{\partial u}{\partial t}(b;t).$$

If we define $v(x;t) = \partial u/\partial t$, we have the following IVP for v:

$$\begin{aligned} v'' &= f_u(x,u,u')\, v + f_{u'}(x,u,u')\, v' \\ v(a;t) &= 0 \\ v'(a;t) &= 1. \end{aligned}$$

Here v' and v'' are the first and 2nd order partial derivatives of v, with respect to x. The above set of equations are obtained from taking partial derivative with respect to x for the system for u. The chain rule is used to obtain the differential equation of v. Now, we have $\phi'(t) = v(b;t)$. Here is the algorithm for the shooting method which involves Newton's method for solving $\phi(t) = 0$:

$t_0 =$ initial guess for $y'(a)$.
for $j = 0,1,2,...$
  solve the following system numerically from $x = a$ to $x = b$
    $u'' = f(x,u,u')$
    $u|_{x=a} = \alpha$
    $u'|_{x=a} = t_j$
    $v'' = f_u(x,u,u')v + f_{u'}(x,u,u')v'$
    $v|_{x=a} = 0$
    $v'|_{x=a} = 1.$
  set
    $t_{j+1} = t_j - \frac{u|_{x=b} - \beta}{v|_{x=b}}.$

If we want to use the methods developed in the previous chapter to solve the above system of two 2nd order equations for u and v, we need to introduce a vector $z = (u,u',v,v')^T$ and write the differential equation as $z' = F(x,z)$ for some vector F. The initial condition is $z(a) = (\alpha,t_j,0,1)^T$.

The shooting method is also applicable to eigenvalue problem:

$$y'' = f(x, y, y', \lambda), \quad a < x < b,$$
$$y(a) = 0, \quad y(b) = 0,$$

where f satisfies the condition $f(x, 0, 0, \lambda) = 0$ and more generally, f is homogeneous in y, i.e.,

$$f(x, cy, cy', \lambda) = c f(x, y, y', \lambda).$$

for any constant c. Notice that $y = 0$ is always a solution of the above boundary value problem. In fact, an eigenvalue problem is a special boundary value problem satisfying (1) $y = 0$ is always a solution, (2) there is a parameter called λ in the equation (or boundary condition). The eigenvalue problem is to determine non-zero solutions which exist only for special values of λ. The solutions of eigenvalue problems are the pairs $\{\lambda, y(x)\}$, where λ is the eigenvalue and y is the eigenfunction. Usually, there are many (may be infinite) eigenvalues and eigenfunctions. To use the shooting method, we consider the initial value problem

$$u'' = f(x, u, u', \lambda), \quad x > a,$$
$$u(a; \lambda) = 0,$$
$$u'(a; \lambda) = 1,$$

where λ is considered as a parameter. Since the solution depends on λ, we use the notation $u = u(x; \lambda)$, but u' and u'' represent the first and second order derivatives with respect to x. For any given λ, we can solve the above initial value problem. Now suppose λ satisfies the condition

$$\phi(\lambda) = u(b; \lambda) = 0,$$

then $y(x) = u(x, \lambda)$ is the eigenfunction we are looking for, and λ is the corresponding eigenvalue. Therefore, we just have to use secant or Newton's method to solve λ from the equation $\phi(\lambda) = 0$. If a secant method is used, we just have to solve initial value problems for different iterates of λ. If Newton's method is used, we must evaluate $\phi'(\lambda)$ for given λ. Therefore, we need

$$v(x; \lambda) = \frac{\partial u}{\partial \lambda}(x; \lambda).$$

We need an initial value problem for v. This can be obtained by taking partial derivative with respect to λ for the initial value problem of u. We have

$$v'' = f_u(x, u, u', \lambda)v + f_{u'}(x, u, u', \lambda)v' + f_\lambda(x, u, u', \lambda), \quad x > a,$$
$$v(a; \lambda) = 0,$$
$$v'(a; \lambda) = 0.$$

Notice that we have been using the chain rule (of Calculus) to get the equation for v. Now, you can solve the initial value problem for v (together with u), then evaluate $\phi'(\lambda)$ for any given λ.

5.2 Finite difference methods

The basic idea of "finite difference method" is to replace the derivatives in a differential equation by "difference approximations".

To approximate the derivative $f'(x_0)$, we can use the left side of the following equations:

1. Forward difference:
$$\frac{f(x_0+h)-f(x_0)}{h} = f'(x_0) + \frac{h}{2}f''(x_0) + \ldots$$
2. Backward difference:
$$\frac{f(x_0)-f(x_0-h)}{h} = f'(x_0) - \frac{h}{2}f''(x_0) + \ldots$$
3. Central difference:
$$\frac{f(x_0+h)-f(x_0-h)}{2h} = f'(x_0) + \frac{h^2}{6}f'''(x_0) + \ldots$$
4. Central difference using half step:
$$\frac{f(x_0+0.5h)-f(x_0-0.5h)}{h} = f'(x_0) + \frac{h^2}{24}f'''(x_0) + \ldots$$
5. Three-point formulas:
$$\begin{aligned}\frac{-f(x_0+2h)+4f(x_0+h)-3f(x_0)}{2h} &= f'(x_0) - \frac{h^2}{3}f'''(x_0) + \ldots \\ \frac{f(x_0-2h)-4f(x_0-h)+3f(x_0)}{2h} &= f'(x_0) - \frac{h^2}{3}f'''(x_0) + \ldots\end{aligned}$$

For the second order derivative, we have:
$$\frac{f(x_0+h)-2f(x_0)+f(x_0-h)}{h^2} = f''(x_0) + \frac{h^2}{12}f^{(4)}(x_0) + \ldots$$

We consider the following two-point BVP of a linear 2nd order ODE:
$$\begin{aligned}&y'' + p(x)y' + q(x)y = r(x), \quad a < x < b \\ &y(a) = \alpha \\ &y(b) = \beta.\end{aligned}$$

Let
$$x_0 = a, \;\; x_j = x_0 + jh, \;\; \text{and} \;\; x_{n+1} = b,$$
we obtain
$$h = \frac{b-a}{n+1}.$$
We are looking for y_j for $j = 1,2,\ldots,n$, where
$$y_j \approx y(x_j).$$

We also let $y_0 = y(x_0) = y(a) = \alpha$ and $y_{n+1} = y(x_{n+1}) = y(b) = \beta$. Thus, y_0 and y_{n+1} are known. The derivatives at x_j can be approximated by

$$\begin{aligned} y'(x_j) &\approx \frac{y(x_{j+1}) - y(x_{j-1})}{2h} \\ y''(x_j) &\approx \frac{y(x_{j-1}) - 2y(x_j) + y(x_{j+1})}{h^2}. \end{aligned}$$

These are the central difference approximations. Therefore, the 2nd order differential equation is discretized by

$$\frac{y_{j-1} - 2y_j + y_{j+1}}{h^2} + p(x_j)\frac{y_{j+1} - y_{j-1}}{2h} + q(x_j)y_j = r(x_j)$$

for $j = 1, 2, ..., n$. This can be written as

$$\left[1 - \frac{h}{2}p(x_j)\right] y_{j-1} + [-2 + h^2 q(x_j)]y_j + \left[1 + \frac{h}{2}p(x_j)\right] y_{j+1} = h^2 r(x_j).$$

We define

$$\begin{aligned} a_j &= -2 + h^2 q(x_j) \\ b_j &= 1 - p(x_j)h/2 \\ c_j &= 1 + p(x_j)h/2 \\ d_j &= h^2 r(x_j) \end{aligned}$$

and obtain

$$\begin{bmatrix} a_1 & c_1 & & & \\ b_2 & a_2 & c_2 & & \\ & b_3 & \ddots & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ & & & b_n & a_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} d_1 - b_1 y(a) \\ d_2 \\ d_3 \\ \vdots \\ d_n - c_n y(b) \end{bmatrix}.$$

If we have the boundary condition

$$y'(b) = \beta$$

then we let

$$x_0 = a, \quad x_j = x_0 + jh, \text{ and } x_n = b.$$

Thus

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

While $y_0 = y(x_0) = y(a) = \alpha$ is known, $y_n = y(x_n) = y(b)$ is not known. Therefore, we have the n unknowns:

$$y_1, y_2, ..., y_n,$$

where $y_j \approx y(x_j)$. Imagine that there is $x_{n+1} = b + h$ and $y_{n+1} \approx y(x_{n+1})$, then we can write down the approximation of the differential equation at $x = b$ as

$$b_n y_{n-1} + a_n y_n + c_n y_{n+1} = d_n$$

where a_j, b_j, c_j and d_j are defined by the same set of formulas (even though x_j here is different). The boundary condition $y'(b) = \beta$ can be discretized as

$$\frac{y_{n+1} - y_{n-1}}{2h} = y'(b) = \beta.$$

We can then solve y_{n+1} and subsititute to the equation obtained earlier. This leads to

$$(b_n + c_n)y_{n-1} + a_n y_n = d_n - 2\beta h c_n$$

Therefore, with this new boundary condition at $x = b$ and the new definition of h and x_j, we have

$$\begin{bmatrix} a_1 & c_1 & & & \\ b_2 & a_2 & c_2 & & \\ & b_3 & \ddots & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ & & & b_n + c_n & a_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} d_1 - b_1 y(a) \\ d_2 \\ d_3 \\ \vdots \\ d_n - 2hc_n y'(b) \end{bmatrix}.$$

5.3 The finite element method

Consider the following linear second order ODE boundary value problem:

$$\begin{aligned} &u'' + p(x)u' + q(x)u = r(x) \ \text{ for } \ a < x < b \\ &u(a) = \alpha, \ \ u(b) = \beta, \end{aligned}$$

where a, b, α and β are given constants, p, q and r are given functions. We start with the discretization:

$$a = x_0 < x_1 < ... < x_{n-1} < x_n < x_{n+1} = b,$$

then try to find numerical solutions on the grid:

$$u_j \approx u(x_j) \quad \text{for} \quad j = 0, 1, ..., n, n+1.$$

From the boundary conditions, we have $u_0 = \alpha$, $u_{n+1} = \beta$. Thus the unknowns are u_1, u_2, ..., u_n. The finite element method provides us an approach for computing these n unknowns.

The finite element method relies on an integral relation derived from the differential equation. Let φ be a differentiable function of x satisfying $\varphi(a) = \varphi(b) = 0$, we can multiply φ to the differential equation of u and integrate. That is

$$\int_a^b \varphi(x)[u'' + p(x)u' + q(x)u - r(x)]\, dx = 0.$$

If we use integration by parts for the term involving u'', we obtain

$$\int_a^b \left[-\varphi' u' + p(x)\varphi u' + q(x)\varphi u - r(x)\varphi\right] dx = 0. \tag{5.1}$$

Now, we consider the basis function $\phi_j(x)$ defined as the *continuous piecewise linear* function satisfying

$$\phi_j(x_j) = 1, \quad \phi_j(x_k) = 0 \ \text{ if } \ k \neq j.$$

More precisely, we have

$$\phi_j(x) = \begin{cases} (x - x_{j-1})/(x_j - x_{j-1}), & x_{j-1} < x \le x_j, \\ (x_{j+1} - x)/(x_{j+1} - x_j), & x_j < x < x_{j+1}, \\ 0, & \text{otherwise.} \end{cases}$$

The derivative of this function is piecewise constant. We have

$$\phi_j'(x) = \begin{cases} 1/(x_j - x_{j-1}), & x_{j-1} < x < x_j, \\ -1/(x_{j+1} - x_j), & x_j < x < x_{j+1}, \\ 0, & \text{otherwise.} \end{cases}$$

The piecewise linear function obtained by connecting (x_j, u_j) by line segments is

$$u^{(n)}(x) = \sum_{j=0}^{n+1} u_j \phi_j(x) \tag{5.2}$$

Obviously, $u^{(n)}$ is an approximation for $u(x)$. If we plug $u^{(n)}$ into the differential equation, we will not get an exact identity. In fact, $u^{(n)}$ does not even have derivative at the the grid points. In the finite element method, we replace u in (5.1) by $u^{(n)}$ and replace φ in (5.1) by ϕ_k, for $k = 1, 2, ..., n$. This gives rise to n equations for the n unknowns u_1, u_2, ..., u_n. These equations can be written as

$$\sum_{j=0}^{n+1} a_{kj} u_j = b_k$$

where

$$a_{kj} = \int_a^b \left[-\phi_k'(x)\phi_j'(x) + p(x)\phi_k(x)\phi_j'(x) + q(x)\phi_k(x)\phi_j(x) \right] \, dx, \quad b_k = \int_a^b \phi_k(x) r(x) \, dx.$$

for $k = 1, 2, ..., n$. If $|j - k| > 1$, we observe that ϕ_k and ϕ_j are non-zero only on intervals that do not overlap. This leads to

$$a_{kj} = 0 \quad \text{if} \quad |j - k| > 1.$$

Therefore, we have

$$\begin{pmatrix} a_{11} & a_{12} & & \\ a_{21} & a_{22} & \ddots & \\ & \ddots & \ddots & a_{n-1,n} \\ & & a_{n,n-1} & a_{nn} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} b_1 - a_{10} u_0 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n - a_{n,n+1} u_{n+1} \end{pmatrix}.$$

This is a tridiagonal system that can be solved in $O(n)$ operations by a special version of Gaussian elimination with partial pivoting. The formula for a_{kj} can be further simplified if we integrate as much as possible and use approximations for $p(x)$ and $q(x)$ when necessary. We have

$$\begin{aligned} a_{kk} &\approx -\frac{1}{h} - \frac{1}{H} + \frac{1}{2} \left[p(x_{k-1/2}) - p(x_{k+1/2}) \right] + \frac{1}{3} \left[h\, q(x_{k-1/2}) + H\, q(x_{k+1/2}) \right] \\ a_{k,k-1} &\approx \frac{1}{h} - \frac{1}{2}\, p(x_{k-1/2}) + \frac{h}{6}\, q(x_{k-1/2}) \\ a_{k,k+1} &\approx \frac{1}{H} + \frac{1}{2}\, p(x_{k+1/2}) + \frac{H}{6}\, q(x_{k+1/2}) \\ b_k &\approx \frac{1}{2} \left[h\, r(x_{k-1/2}) + H\, r(x_{k+1/2}) \right] \end{aligned}$$

where

$$h = x_k - x_{k-1} \quad H = x_{k+1} - x_k \quad x_{k\pm1/2} = \frac{1}{2}(x_k + x_{k\pm1})$$

The above formulae are exact when p and q are constants. For the more general p and q, we have used their midpoint values on each interval. Furthermore, if $p(x) = 0$, the resulting tridiagonal coefficient matrix is symmetric.

Chapter 6

Finite Difference Methods for Parabolic PDEs

6.1 Introduction

For scientitic and engineering applications, it is often necessary to solve partial differential equations. Most partial differential equations for practical problems cannot be solved analytically. Therefore, numerical methods for partial differential equations are extremely important. In this chapter, we study numerical methods for simple parabolic partial differential equations.

The simplest parabolic partial differential equation (PDE) is

$$u_t = a u_{xx}, \tag{6.1}$$

where a is a positive constant. Often, this is called the heat equation, when u represents the temperature of a thin rod. Here, x is the spatial variable along the axis of the rod. We assume that the cross section of the rod is very small and the temperature in the cross section is constant. Then, u is only a function of x and time t. Equation (6.1) is also called the diffusion equation. In this case, we consider a thin tube with water and ink inside. The variable u then represents the density of ink in the tube. As the tube is assumed to have a very small cross section, u is assumed to depend only on x and t. Because of this interprtetation, the coefficient a is called the diffusion coefficient.

Equation (6.1) must be solved with some boundary conditions and an initial condition. Assume that the rod is given by $0 < x < L$ (the length of the rod is L), we solve (6.1) with the following two boundary conditions:

$$u(0,t) = \alpha, \quad u(L,t) = \beta, \tag{6.2}$$

and the following initial condition:

$$u(x,0) = g(x). \tag{6.3}$$

Here, α and β are given constants, g is a given function of x. As $t \to \infty$, the temperature setttles down to a time independent (i.e., steady) solution:

$$\lim_{t\to\infty} u(x,t) = u_\infty(x) = \alpha + \frac{x}{L}(\beta - \alpha).$$

The above solution gives a linear profile that changes from $u = \alpha$ at one end of the rod to $u = \beta$ at the other end of the rod. For $t < \infty$, we have the following time dependent solution:

$$u(x,t) = u_\infty(x) + \sum_{k=1}^{\infty} c_k e^{-a(k\pi/L)^2 t} \sin \frac{k\pi x}{L},$$

where the coefficients $\{c_k\}$ can be determined from the intial condition $u(x,0) = g(x)$.

If the rod is not uniform in the x direction (different part of the rod may be made from differential materials), the coefficient a is no longer a constant. Therefore, we consider the following general parabolic equation:

$$u_t = a(x)u_{xx} + b(x)u_x + c(x)u + d(x), \quad 0 < x < L. \tag{6.4}$$

Here, a, b, c and d are given functions of x, the term $d(x)$ corresponds to some heat source in the rod. We can solve the above equation with the initial condition (6.3), the boundary conditions (6.2) or the following boundary conditions:

$$u_x(0,t) = e_0 u(0,t) + f_0, \quad u_x(L,t) = e_1 u(L,t) + f_1, \tag{6.5}$$

where e_0, f_0, e_1 and f_1 are given constants.

6.2 Classical explicit method

We consider equation (6.4) with the boundary conditions (6.2) and the initial condition (6.3). First, we discretize x and t by

$$x_j = j\Delta x, \quad \Delta x = \frac{L}{n+1}, \quad t_k = k\Delta t$$

for some integer n and some $\Delta t > 0$. We will use the notation u_j^k to represent the numerical solution. That is,

$$u_j^k \approx u(x_j, t_k).$$

From the initial condition (6.3), we have

$$u_j^0 = g(x_j), \quad j = 0, 1, 2, ..., n+1.$$

From the boundary conditions (6.2), we obtain

$$u_0^k = \alpha, \quad u_{n+1}^k = \beta.$$

Our objective is to find u_j^k for $k > 0$ and for $j = 1, 2, ..., n$.

For the derivatives in (6.4), we have the following difference approximantions:

$$\begin{aligned}
u_t(x_j, t_k) &\approx \frac{u(x_j, t_{k+1}) - u(x_j, t_k)}{\Delta t}, \\
u_x(x_j, t_k) &\approx \frac{u(x_{j+1}, t_k) - u(x_{j-1}, t_k)}{2\Delta x}, \\
u_{xx}(x_j, t_k) &\approx \frac{u(x_{j+1}, t_k) - 2u(x_j, t_k) + u(x_{j-1}, t_k)}{(\Delta x)^2}.
\end{aligned}$$

Notice that for the time derivative, we only use the first order forward difference formula. If we insert these difference formulas into the differential equation (6.4) and replace $u(x_j,t_k)$ by u_j^k, etc, we obtain:

$$\frac{1}{\Delta t}(u_j^{k+1}-u_j^k) = \frac{a_j}{(\Delta x)^2}(u_{j+1}^k - 2u_j^k + u_{j-1}^k) + \frac{b_j}{2\Delta x}(u_{j+1}^k - u_{j-1}^k) + c_j u_j^k + d_j \tag{6.6}$$

Here, $a_j = a(x_j)$, $b_j = b(x_j)$, $c_j = c(x_j)$ and $d_j = d(x_j)$. The above is an **explicit** formula for u_j^{k+1}. Numerical implementation of the above is very simple. We need a loop in k, for $k = 0,1,2,....$ Inside this loop, we need another loop for j. It is for $j = 1,2,...,n$.

Since we have used the first order forward difference formula to approximate u_t, we have an $O(\Delta t)$ error from the discretization of t. We have used the second order central difference formulas for u_x and u_{xx}, thus we have an $O((\Delta x)^2)$ error from the discretization of x. If we keep $\Delta t = O((\Delta x)^2)$, then the errors from the discretizations of t and x are roughly the same magnitude. This suggests that the time step should be small. However, there are more serious reasons. Notice that the forward difference approximation corresponds to Euler's method for ODE. It is not suitable for **stiff** differential equations. Here, we have a PDE, but if we discretize x first by the central difference formulas (and keep the original continuous t), we obtain a system of ODEs. If we then discretize the system by Euler's method, we get exactly the same method as (6.6). It turns out that our system of ODEs (obtained with a discretization in x only) is a stiff system. Thus, if Δt is not small, the error will grow **exponentially**. To avoid the use of very small Δt, we need an implicit method.

6.3 Crank-Nicolson method

If we define the half time step:

$$t_{k+1/2} = t_k + \frac{\Delta t}{2},$$

then

$$\frac{u(x_j,t_{k+1}) - u(x_j,t_k)}{\Delta t} \approx u_t(x_j,t_{k+1/2})$$

is a second order formula. To discretize (6.4), we use the same difference formulas for the x-derivatives, but also do an average between the t_k and t_{k+1} time levels. More precisely, we have

$$\begin{aligned}\frac{1}{\Delta t}(u_j^{k+1}-u_j^k) &= \frac{a_j}{2(\Delta x)^2}(u_{j+1}^k - 2u_j^k + u_{j-1}^k + u_{j+1}^{k+1} - 2u_j^{k+1} + u_{j-1}^{k+1}) \\ &+ \frac{b_j}{4\Delta x}(u_{j+1}^k - u_{j-1}^k + u_{j+1}^{k+1} - u_{j-1}^{k+1}) + \frac{c_j}{2}(u_j^k + u_j^{k+1}) + d_j.\end{aligned} \tag{6.7}$$

For boundary conditions (6.2), we have

$$u_0^k = \alpha, \quad u_{n+1}^k = \beta$$

for all k. We can then re-write the above numerical method as

$$A \begin{bmatrix} u_1^{k+1} \\ u_2^{k+1} \\ \vdots \\ u_n^{k+1} \end{bmatrix} = B \begin{bmatrix} u_1^k \\ u_2^k \\ \vdots \\ u_n^k \end{bmatrix} + \vec{p}, \tag{6.8}$$

where A and B are tridiagonal matrices given by

$$a_{jj} = 1 + \frac{a_j \Delta t}{(\Delta x)^2} - \frac{c_j \Delta t}{2}, \tag{6.9}$$

$$a_{j,j-1} = -\frac{a_j \Delta t}{2(\Delta x)^2} + \frac{b_j \Delta t}{4\Delta x}, \tag{6.10}$$

$$a_{j,j+1} = -\frac{a_j \Delta t}{2(\Delta x)^2} - \frac{b_j \Delta t}{4\Delta x}, \tag{6.11}$$

$$b_{jj} = 1 - \frac{a_j \Delta t}{(\Delta x)^2} + \frac{c_j \Delta t}{2} = 2 - a_{jj}, \tag{6.12}$$

$$b_{j,j-1} = \frac{a_j \Delta t}{2(\Delta x)^2} - \frac{b_j \Delta t}{4\Delta x} = -a_{j,j-1}, \tag{6.13}$$

$$b_{j,j+1} = \frac{a_j \Delta t}{2(\Delta x)^2} + \frac{b_j \Delta t}{4\Delta x} = -a_{j,j+1} \tag{6.14}$$

and $\vec{p}$ is the following vector

$$\vec{p} = \begin{bmatrix} d_1 \Delta t - a_{10} u_0^{k+1} + b_{10} u_0^k \\ d_2 \Delta t \\ \vdots \\ d_{n-1} \Delta t \\ d_n \Delta t - a_{n,n+1} u_{n+1}^{k+1} + b_{n,n+1} u_{n+1}^k \end{bmatrix} = \begin{bmatrix} d_1 \Delta t - 2a_{10}\alpha \\ d_2 \Delta t \\ \vdots \\ d_{n-1} \Delta t \\ d_n \Delta t - 2a_{n,n+1}\beta \end{bmatrix}.$$

Since the matrices A and B are tridiagonal, we have

$$a_{jk} = 0, \quad b_{jk} = 0, \quad \text{if} \quad |j-k| \geq 2.$$

Therefore, for each step, we need to solve a linear system with a tridiagonal matrix. This can be done efficiently in $O(n)$ operations.

The Crank-Nicolson method corresponds to the "implicit midpoint" method for ODE IVP. If we discretize the x variable only for (6.4), we obtain a system of ODEs. If we then apply the implicit midpoint method, we get the Crank-Nicolson method.

6.4 Stability analysis

The stability analysis is to find out for what values of Δt and Δx, the numerical method is stable. Let us consider the simple constant coefficient heat equation

$$u_t = a u_{xx},$$

where a is a positive constant. Let x and t be discretized as

$$x_j = j\Delta x, \quad t_k = k\Delta t$$

for integers j and k. Let u_j^k be a numerical solution for $u(x_j, t_k)$. That is

$$u_j^k \approx u(x_j, t_k).$$

The classical explicit method described in section 3.1.1 gives

$$\frac{1}{\Delta t}(u_j^{k+1} - u_j^k) = \frac{a}{(\Delta x)^2}(u_{j+1}^k - 2u_j^k + u_{j-1}^k). \tag{6.15}$$

To understand the stability of this method, we consider special solutions of the following form

$$u_j^k = \rho^k e^{i\beta j \Delta x}, \tag{6.16}$$

where β is an arbitrary constant, ρ is to be determined. If $|\rho| > 1$, then the solution grows exponentially in time, so the numerical method is unstable. Otherwise, it is stable. The purpose of stability analysis is to find out the values of Δt and Δx such that $|\rho| \leq 1$. If we insert the (6.16) into (6.15), we can solve ρ in terms of β, Δx and Δt. We have

$$\frac{1}{\Delta t}(\rho - 1) = \frac{a}{(\Delta x)^2}(e^{i\tilde{\beta}} - 2 + e^{-i\tilde{\beta}}),$$

where $\tilde{\beta} = \beta \Delta x$. This can be simplified to

$$\rho = 1 - 4\frac{a\Delta t}{(\Delta x)^2}\sin^2\frac{\tilde{\beta}}{2}.$$

We can see that $\rho \leq 1$ for any real β. However, it is possible to have $\rho < -1$. For stability, we require that $|\rho| \leq 1$ for all choice of β. This is guaranteed if

$$4\frac{a\Delta t}{(\Delta x)^2} \leq 2.$$

This gives rise to

$$\Delta t \leq \frac{(\Delta x)^2}{2a}.$$

This is the condition on Δt for stability of the numerical method. If the above is not valid, then, we can find a β, such that $\rho < -1$. In that case, the numerical method becomes unstable. Since there is a condition on Δt for the method to be stable, we call such a method **conditionally stable**.

The following MATLAB program illustrates the stability and instability for $\Delta t = 1/20000$ and $\Delta t = 1/19997$ respectively. In this example, we have $a = 1$ and $\Delta x = 0.01$.

```
% we will solve the heat equation u_t = u_{xx} for 0 < x < 1, with
% zero boundary conditions at x=0 and x=1 and the initial condition:
% u(x, 0) = 1- 2 |x - 0.5|.

% we choose dx = 0.01
dx = 0.01;
x = 0: dx: 1;
m = length(x);
u = 1 - 2 * abs(x - 0.5);
u(1) = 0;
u(m) = 0;
```

```
% we solve up to t = 1.
steps = 19997;    % unstable
% steps = 20000;    % stable
dt = 1/steps;
s = dt/(dx*dx);
for k= 0 : steps-1
  b = u(1);
  for j=2:m-1
    a = u(j);
    u(j) = a + s*(u(j+1)-2*a+ b);
    b = a;
  end
end
plot(x, u)
```

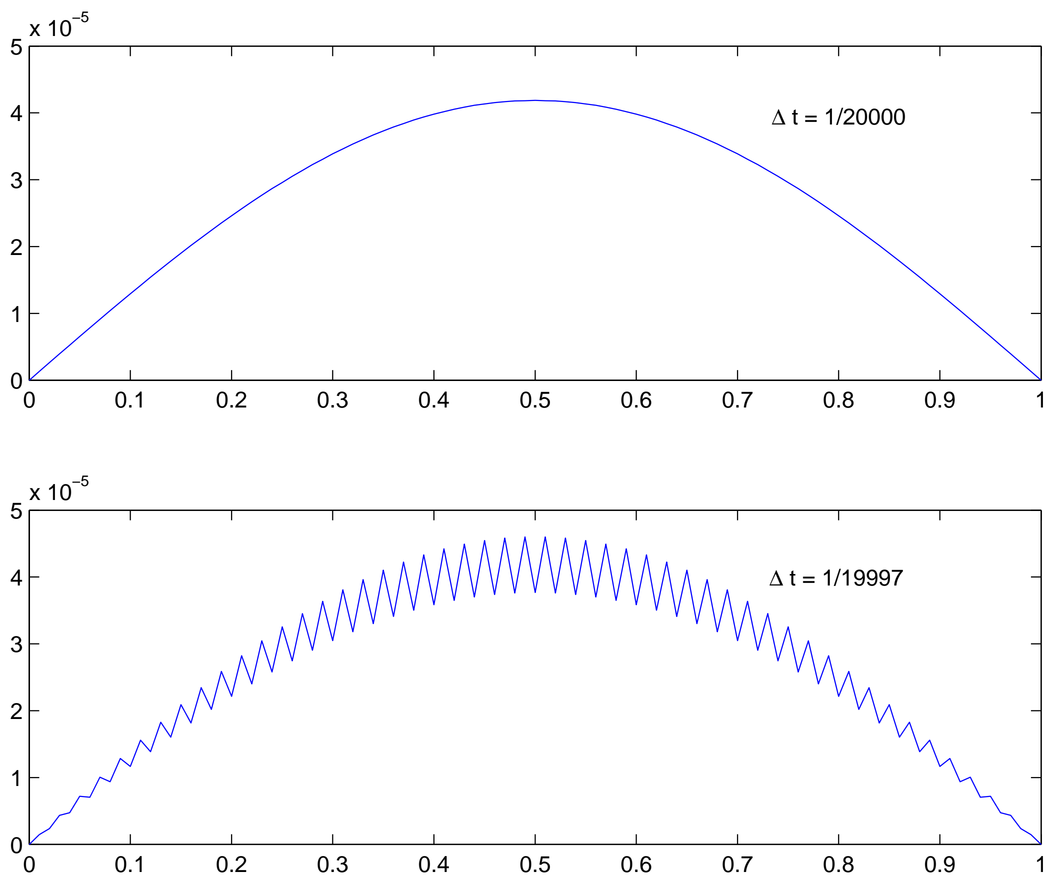

Figure 6.1: Stability of the classical explicit method for the heat equation.

Next, we perform a stability analysis for the Crank-Nicolson method. For $u_t = au_{xx}$, we have the

following discrete formula:

$$\frac{1}{\Delta t}(u_j^{k+1} - u_j^k) = \frac{a}{2(\Delta x)^2}(u_{j+1}^k - 2u_j^k + u_{j-1}^k + u_{j+1}^{k+1} - 2u_j^{k+1} + u_{j-1}^{k+1}). \tag{6.17}$$

For the special solution $u_j^k = \rho^k e^{i\beta j\Delta x}$, we have

$$\frac{1}{\Delta t}(\rho - 1) = \frac{a}{2(\Delta x)^2}[e^{i\tilde{\beta}} - 2 + e^{-i\tilde{\beta}} + \rho(e^{i\tilde{\beta}} - 2 + e^{-i\tilde{\beta}})]$$

This is reduced to

$$\rho = \frac{1-s}{1+s}, \quad s = \frac{2a\Delta t}{(\Delta x)^2}\sin^2\frac{\tilde{\beta}}{2},$$

where $\tilde{\beta} = \beta\Delta x$. Clearly, $s \geq 0$, therefore,

$$|\rho| \leq 1$$

for all choice of β, Δt and Δx. Therefore, Crank-Nicolson is always stable. Since there is no condition for stability, we call such a method **unconditionally stable**.

6.5 Alternating direction implicit method

In this section, we consider the heat equation with two spatial varriables:

$$u_t = a(u_{xx} + u_{yy}), \quad (x,y) \in \Omega, \quad t > 0. \tag{6.18}$$

The initial condition is

$$u(x,y,0) = f(x,y), \quad (x,y) \in \Omega. \tag{6.19}$$

We assume that u satisfy the so-called Dirichlet boundary condition.

$$u(x,y,t) = g(x,y), \quad (x,y) \in \partial\Omega. \tag{6.20}$$

That is, u is given on $\partial\Omega$ — the boundary of Ω. We will consider the case where Ω is the square:

$$\Omega = \{(x,y) \mid 0 < x < L,\ 0 < y < L\}.$$

We can discretize x and y by

$$x_i = ih,\ \ y_j = jh,\ \ h = \frac{L}{n+1},\ \ i,j = 0,1,...,n,n+1$$

and discretize t by $t_k = k\Delta t$ for $k = 0,1,2,....$

As in the previous section, we have a classical explicit method that uses forward difference approximation for u_t and central difference approximation for u_{xx} and u_{yy}. This gives rise to

$$\frac{u_{ij}^{k+1} - u_{ij}^k}{\Delta t} = \frac{a}{h^2}\left(u_{i-1,j}^k + u_{i+1,j}^k + u_{i,j-1}^k + u_{i,j+1}^k - 4u_{ij}^k\right), \tag{6.21}$$

where $u_{ij}^k \approx u(x_i, y_j, t_k)$, etc. The method is very easy to use. Given the numerical solutions at time level t_k, we can simply evaluate u_{ij}^{k+1} for $i = 1,2,...,n$ and $j = 1,2,...,n$ to obtain the numerical solutions at time level t_{k+1}. However, this method is only stable when Δt satisfies

$$\Delta t \leq \frac{h^2}{4a}.$$

The method is unstable if $\Delta t > h^2/(4a)$. In any event, since there is a condition on Δt for stability, we say that this method is "conditionally" stable. This stability condition is derived for constant coefficient a and the boundary condition is ignored. Notice that Δt must be related to the square of h for stability. Therefore, we have to use a very small time step.

Similar to the previous section, we also have the Crank-Nicolson method:

$$\frac{u_{ij}^{k+1}-u_{ij}^{k}}{\Delta t} = \frac{a}{2h^2}\left(u_{i-1,j}^{k}+u_{i+1,j}^{k}+u_{i,j-1}^{k}+u_{i,j+1}^{k}-4u_{ij}^{k}+u_{i-1,j}^{k+1}+u_{i+1,j}^{k+1}+u_{i,j-1}^{k+1}+u_{i,j+1}^{k+1}-4u_{ij}^{k+1}\right).$$

This is an implicit method. Given the solution at t_k, we must solve n^2 unknowns: u_{ij}^{k+1} for $i=1,2,...,n$ and $j=1,2,...,n$. If we put all these n^2 unknowns into one long vector of length n^2, we have a linear system of equations with a coefficient matrix of size $n^2\times n^2$. It is not so easy to solve this large system efficiently. This is particularly the case when a is repplaced by $a(x,y)$ or $a_{ij}=a(x_i,y_j)$ in the discrtized numerical method. As before, Crank-Nicolson method has good stability property. That is, the method is always stable, or unconditionally stable.

Here, we introduce a method which is a lot easier to solve and is also unconditionally stable. This is the Alternating Direction Implicit (ADI) method. The method was originally developed in the 50's. Instead of solving one large linear system of n^2 unknowns, we need to solve n linear systems each with n unknowns. This is achieved by separating the x and y directions. We present the method without discretizing x and y. If t is discretized, the Crank-Nicolson method is

$$\frac{u^{k+1}-u^{k}}{\Delta t} = \frac{a}{2}(\partial_x^2+\partial_y^2)(u^{k+1}+u^k),$$

where u^k is a function of x and y, and $u^k \approx u(x,y,t_k)$. Thus,

$$\left[1-\frac{a\Delta t}{2}(\partial_x^2+\partial_y^2)\right]u^{k+1} = \left[1+\frac{a\Delta t}{2}(\partial_x^2+\partial_y^2)\right]u^{k}. \tag{6.22}$$

The Crank-Nicolson method has a second order error. If we put the exact solution into the above equation, then the error term is $O((\Delta t)^3)$. That is,

$$\left[1-\frac{a\Delta t}{2}(\partial_x^2+\partial_y^2)\right]u(x,y,t_{k+1}) = \left[1+\frac{a\Delta t}{2}(\partial_x^2+\partial_y^2)\right]u(x,y,t_k)+O((\Delta t)^3).$$

Now, we add

$$\frac{a^2(\Delta t)^2}{4}\partial_x^2\partial_y^2 u^{k+1}$$

to the left hand side of (6.22) and add

$$\frac{a^2(\Delta t)^2}{4}\partial_x^2\partial_y^2 u^{k}$$

to the right hand side of (6.22). Then, we can factor both sides of the new equation and obtain

$$\left(1-\frac{a\Delta t}{2}\partial_x^2\right)\left(1-\frac{a\Delta t}{2}\partial_y^2\right)u^{k+1} = \left(1+\frac{a\Delta t}{2}\partial_x^2\right)\left(1+\frac{a\Delta t}{2}\partial_y^2\right)u^{k}. \tag{6.23}$$

Since $u(x,y,t_{k+1}) = u(x,y,t_k)+O(\Delta t)$, we have

$$\frac{a^2(\Delta t)^2}{4}\partial_x^2\partial_y^2 u(x,y,t_{k+1}) - \frac{a^2(\Delta t)^2}{4}\partial_x^2\partial_y^2 u(x,y,t_k)+O((\Delta t)^3).$$

This implies that (6.23) is still a 2nd order method. Let $s = a\Delta t/2$, we have

$$u^{k+1} = \left(1 - s\partial_y^2\right)^{-1} \left(\frac{1+s\partial_x^2}{1-s\partial_x^2}\right) \left(1+s\partial_y^2\right) u^k. \tag{6.24}$$

This gives rise to following procedure for computing u^{k+1}:

1. evaluate v by
$$v = u^k + s\partial_y^2 u^k. \tag{6.25}$$

2. evlaute w by
$$w = v + s\partial_x^2 v. \tag{6.26}$$

3. solve a new v from
$$v - s\partial_x^2 v = w. \tag{6.27}$$

4. solve u^{k+1} from
$$u^{k+1} - s\partial_y^2 u^{k+1} = v. \tag{6.28}$$

Now, let us consider each of these sub-steps. Let us discretize the x and y variables as before. For (6.25), we use central difference approximation for the second derivative in y. This leads to

$$v_{ij} = u_{ij}^k + \frac{s}{h^2}\left(u_{i,j-1}^k - 2u_{ij}^k + u_{i,j+1}^k\right).$$

We can simply evaluate v_{ij} for $i = 1,2,...,n$ and for $j = 1,2,...,n$. Similarly, for w in (6.26), we have

$$w_{ij} = v_{ij} + \frac{s}{h^2}\left(v_{i-1,j} - 2v_{ij} + v_{i+1,j}\right).$$

We can evaluate w_{ij} for $i = 1,2,...,n$ and for $j = 1,2,...,n$. Notice that we need v_{0j} and $v_{n+1,j}$. These are related to the boundary conditions for u. We simply use the boundary condition of u as the boundary condition of v. Now, for the new v satisfying (6.27), we are solving a boundary value problem. Since the y-derivative is not involved. We can solve for each y_j separately. That is, we solve v_{1j}, v_{2j}, ..., v_{nj} from

$$v_{ij} - \frac{s}{h^2}\left(v_{i-1,j} - 2v_{ij} + v_{i+1,j}\right) = w_{ij}, \quad i = 1,2,...,n.$$

This can be writted as a linear system for n unknowns. That is

$$\begin{bmatrix} c & b & & \\ b & c & \ddots & \\ & \ddots & \ddots & b \\ & & b & c \end{bmatrix} \begin{bmatrix} v_{1j} \\ v_{2j} \\ \vdots \\ v_{nj} \end{bmatrix} = \begin{bmatrix} w_{1j} \\ w_{2j} \\ \vdots \\ w_{nj} \end{bmatrix} + \begin{bmatrix} -bv_{0j} \\ 0 \\ \vdots \\ -bv_{n+1,j} \end{bmatrix}$$

where

$$c = 1 + \frac{2s}{h^2}, \quad b = -\frac{s}{h^2}.$$

Furthermore, we need to use the boundary condition for u as the boundary condition for v again. We let

$$v_{0j} = g(0,y_j), \quad v_{n+1,j} = g(L,y_j).$$

For (6.28), we also have a two-point boundary value problem in only one variable. We can discretize (6.28) as

$$u_{ij}^{k+1} - \frac{s}{h^2}\left(u_{i,j-1}^{k+1} - 2u_{ij}^{k+1} + u_{i,j+1}^{k+1}\right) = v_{ij}, \quad j = 1,2,...,n.$$

This can be written as a linear system:

$$\begin{bmatrix} c & b & & \\ b & c & \ddots & \\ & \ddots & \ddots & b \\ & & b & c \end{bmatrix} \begin{bmatrix} u_{i1}^{k+1} \\ u_{i2}^{k+1} \\ \vdots \\ u_{in}^{k+1} \end{bmatrix} = \begin{bmatrix} v_{i1} \\ v_{i2} \\ \vdots \\ v_{in} \end{bmatrix} + \begin{bmatrix} -bu_{i0}^{k+1} \\ 0 \\ \vdots \\ -bu_{i,n+1}^{k+1} \end{bmatrix}$$

Here, u_{i0}^{k+1} and $u_{i,n+1}^{k+1}$ come from the boundary condition:

$$u_{i0}^{k+1} = g(x_i, 0), \quad u_{i,n+1}^{k+1} = g(x_i, L).$$

If we store u_{ij}^{k+1}, v_{ij} as matrices, the n unknowns for a fixed x_i is actually a row vector. Thus, we can replace the above system by its transpose.

$$[u_{i1}^{k+1}, u_{i2}^{k+1}, \ldots, u_{in}^{k+1}] \begin{bmatrix} c & b & & \\ b & c & \ddots & \\ & \ddots & \ddots & b \\ & & b & c \end{bmatrix} = [v_{i1}, v_{i2}, \ldots, v_{in}] + [-bu_{i0}^{k+1}, 0, \ldots, -bu_{i,n+1}^{k+1}]$$

Using matrix notations, let U^{k+1}, V and W be the $n \times n$ matrices whose (i,j) entries are u_{ij}^{k+1}, v_{ij} and w_{ij}, respectively, we have

$$TV = W + B_1, \quad U^{k+1}T = V + B_2,$$

where T is the tridiagonal matrix

$$T = \begin{bmatrix} c & b & & \\ b & c & \ddots & \\ & \ddots & \ddots & b \\ & & b & c \end{bmatrix},$$

B_1 and B_2 are matrices related to the boundary conditions:

$$B_1 = -b \begin{bmatrix} g(0,y_1) & g(0,y_2) & \ldots & g(0,y_n) \\ 0 & & & 0 \\ \vdots & & & \vdots \\ 0 & & & 0 \\ g(L,y_1) & g(L,y_2) & \ldots & g(L,y_n) \end{bmatrix}, \quad B_2 = -b \begin{bmatrix} g(x_1,0) & 0 & \ldots & 0 & g(x_1,L) \\ g(x_2,0) & 0 & \ldots & 0 & g(x_2,L) \\ \vdots & & & & \vdots \\ \vdots & & & & \vdots \\ g(x_n,0) & 0 & \ldots & 0 & g(x_n,L) \end{bmatrix}.$$

It can be proved that the ADI method is still unconditionally stable.

Chapter 7

Finite Difference Methods for Hyperbolic PDEs

7.1 First order hyperbolic equations

In this section, we consider hyperbolic equations given as

$$u_t + a(x,t)u_x = 0, \tag{7.1}$$

and

$$u_t + [f(u)]_x = 0. \tag{7.2}$$

Notice that Eq. (7.1) is a first order linear equation, while Eq. (7.2), where f is given function of u, is a nonlinear equation in general. As an example of the nonlinear hyperbolic equation, we have the Burger's equation

$$u_t + uu_x = 0.$$

This can be written as (7.2) for $f(u) = u^2/2$.

We start with (7.1) assuming that a is a non-zero constant. Let us consider the following three numerical methods

$$\frac{u_j^{k+1} - u_j^k}{\Delta t} + a\frac{u_{j+1}^k - u_{j-1}^k}{2\Delta x} = 0, \tag{7.3}$$

$$\frac{u_j^{k+1} - u_j^k}{\Delta t} + a\frac{u_{j+1}^k - u_j^k}{\Delta x} = 0, \tag{7.4}$$

$$\frac{u_j^{k+1} - u_j^k}{\Delta t} + a\frac{u_j^k - u_{j-1}^k}{\Delta x} = 0. \tag{7.5}$$

In all three methods, the time derivative is approximated by the forward difference scheme. For the partial derivative in x, we have used the central difference approximation in (7.3), the forward difference approximation in (7.4) and the backward difference approximation in (7.5).

To find the stability of these methods, we follow the standard procedure as in the **Von Neumann stability analysis**. We look for a special solution gievn as

$$u_j^k = \rho^k e^{i\beta x_j} = \rho^k e^{ij\tilde{\beta}}, \tag{7.6}$$

where $x_j = j\Delta x$ and $\tilde{\beta} = \beta\Delta x$. If we insert the special solution into a finite difference method, we obtain a relation between ρ and $\tilde{\beta}$. Now for a fixed Δt, if there is at least one $\tilde{\beta}$, such that $|\rho| > 1$, then the numerical method is unstable for that Δt. If $|\rho| \leq 1$ for all $\tilde{\beta}$, then the numerical method is stable for that Δt. Furthermore, if a numerical method is unstable for all $\Delta t > 0$, so the method is completely useless, we call the method **unconditionally unstable**. If the method is stable for small Δt (usually given by an inequality) and unstable for large Δt, then we call the method **conditionally stable**. If the method is stable for all $\Delta t > 0$, then we call the method **unconditionally stable**. For these three methods, (7.3) is unconditionally unstable. If $a > 0$, then (7.4) is unconditionally unstable and (7.5) is conditionally stable. If $a < 0$, then (7.4) is conditionally stable and (7.5) is unconditionally unstable. Here, let us prove that (7.5) is conditionally stable if $a > 0$. For u_j^k given in (7.6), Eq. (7.5) gives

$$\rho - 1 + s(1 - e^{-i\tilde{\beta}}) = 0,$$

where $s = a\Delta t/\Delta x$. That is

$$\rho = 1 - s + s\cos\tilde{\beta} - is\sin\tilde{\beta}.$$

Therefore,

$$|\rho|^2 = 1 - 2s(1-s)(1-\cos\tilde{\beta}).$$

If $0 \leq s \leq 1$, then $|\rho|^2 \leq 1$, then $|\rho| \leq 1$, therefore the numerical method is stable. If $s > 1$, we can choose $\tilde{\beta} = \pi/2$ such that $\cos\tilde{\beta} = 0$, then $|\rho|^2 = 1 - 2s(1-s) = 1 + 2s(s-1) > 1$, therefore the method is unstable. In conclusion, (7.5) is conditionally stable and the stability condition is

$$s = \frac{a\Delta t}{\Delta x} \leq 1.$$

Here, we have already assumed $a > 0$, thus $s > 0$.

As a result of the stability analysis, we have to use (7.4) or (7.5) selectively, depending on the sign of a. For a general $a = a(x,t)$, we have the following **upwind** scheme for (7.1):

$$u_j^{k+1} = u_j^k - s_j^k(u_{j+1}^k - u_j^k), \quad \text{if} \quad a(x_j,t_k) < 0, \tag{7.7}$$

$$u_j^{k+1} = u_j^k - s_j^k(u_j^k - u_{j-1}^k), \quad \text{if} \quad a(x_j,t_k) > 0, \tag{7.8}$$

where $s_j^k = a(x_j,t_k)\Delta t/\Delta x$. For the nonlinear equation (7.2), the upwind scheme is

$$\frac{u_j^{k+1} - u_j^k}{\Delta t} + \frac{f(u_{j+1}^k) - f(u_j^k)}{\Delta x} = 0, \quad \text{if} \quad \frac{f(u_{j+1}^k) - f(u_j^k)}{u_{j+1}^k - u_j^k} < 0, \tag{7.9}$$

$$\frac{u_j^{k+1} - u_j^k}{\Delta t} + \frac{f(u_j^k) - f(u_{j-1}^k)}{\Delta x} = 0, \quad \text{if} \quad \frac{f(u_j^k) - f(u_{j-1}^k)}{u_j^k - u_{j-1}^k} > 0. \tag{7.10}$$

The principle here is

$$\frac{\partial f(u)}{\partial x} = f'(u)\frac{\partial u}{\partial x},$$

therefore, $f'(u) = df/du$ plays the role of a in (7.1). Actually, $f'(u)$ may change sign, so that the two conditions

$$a_{j+\frac{1}{2}}^k = \frac{f(u_{j+1}^k) - f(u_j^k)}{u_{j+1}^k - u_j^k} < 0, \quad a_{j-\frac{1}{2}}^k = \frac{f(u_j^k) - f(u_{j-1}^k)}{u_j^k - u_{j-1}^k} > 0,$$

may be satisfied simultaneously. Therefore, we merge the two equations into one:

$$u_j^{k+1} = u_j^k - \frac{\Delta t}{2\Delta x}\left\{[1-\text{sgn}(a_{j+1/2}^k)][f(u_{j+1}^k)-f(u_j^k)] + (1+\text{sgn}(a_{j-1/2}^k))[f(u_j^k)-f(u_{j-1}^k)]\right\}, \quad (7.11)$$

where

$$\text{sgn}(z) = \begin{cases} 1, & z>0, \\ 0, & z=0, \\ -1, & z<0. \end{cases}$$

The upwind scheme is only a first order numerical method (first order in both t and x). Next, we introduce the second order **Lax-Wendroff** method. The basic idea of the Lax-Wendroff method is to use the Taylor series:

$$u(x,t+\Delta t) = u(x,t) + \Delta t\, u_t(x,t) + \frac{(\Delta t)^2}{2} u_{tt}(x,t) + \ldots$$

but we only keep the first three terms in the Taylor series. Let us start with (7.1) assuming that a is a constant. We have

$$u_t = -au_x, \quad u_{tt} = -au_{tx} = a^2 u_{xx}.$$

Therefore,

$$u(x,t+\Delta t) \approx u(x,t) - a\Delta t\, u_x(x,t) + \frac{(a\Delta t)^2}{2} u_{xx}(x,t).$$

Now, we can use central difference approximations and obtain

$$u_j^{k+1} = u_j^k - \frac{s}{2}(u_{j+1}^k - u_{j-1}^k) + \frac{s^2}{2}(u_{j-1}^k - 2u_j^k + u_{j+1}^k), \quad (7.12)$$

where $s = a\Delta t/\Delta x$. We can carry out a stability analysis for (7.12). Insert u_j^k as in (7.6) into (7.12), we obtain

$$\rho = 1 - is\sin\tilde{\beta} + s^2(\cos\tilde{\beta} - 1),$$

and

$$|\rho|^2 = 1 + 4s^2(s^2-1)\sin^4\frac{\tilde{\beta}}{2}.$$

This leads to the conclusion that Lax-Wendroff method is conditionally stable and the stability condition is $|s| \le 1$ or

$$\frac{|a|\Delta t}{\Delta x} \le 1.$$

Now, let us consider the Lax-Wendroff method for (7.1) where a varies with x and t. From $u_t = -au_x$, we obtain

$$u_{tt} = -(au_x)_t = -a_t u_x + a(au_x)_x.$$

Therefore,

$$\begin{aligned} u(x,t+\Delta t) &\approx u - a\Delta t u_x + \frac{(\Delta t)^2}{2}[-a_t u_x + a(au_x)_x] \\ &= u - \Delta t(a + \frac{\Delta t}{2}a_t)u_x + \frac{(\Delta t)^2}{2}a(au_x)_x \\ &\approx u - \Delta t a(x, t+\Delta t/2)u_x + \frac{(\Delta t)^2}{2}a(au_x)_x, \end{aligned}$$

where $u = u(x,t)$, $u_x = u_x(x,t)$ and $a = a(x,t)$. Now, we can use the central difference scheme and obtain

$$u_j^{k+1} = u_j^k - \frac{\nu}{2} a_j^{k+1/2}(u_{j+1}^k - u_{j-1}^k) + \frac{\nu^2}{2} a_j^k \left[a_{j+1/2}^k (u_{j+1}^k - u_j^k) - a_{j-1/2}^k (u_j^k - u_{j-1}^k) \right], \tag{7.13}$$

where

$$\nu = \frac{\Delta t}{\Delta x}, \quad a_j^k = a(x_j, t_k), \quad a_j^{k+1/2} = a(x_j, t_k + \frac{\Delta t}{2}), \ldots$$

Finally, we consider the Lax-Wendroff for the nonlinear equation (7.2). Let us denote $a = a(u) = f'(u) = df/du$, then from $u_t = -[f(u)]_x$, we have $u_{tt} = -[f(u)]_{tx} = [af_x]_x$, and

$$u(x, t + \Delta t) \approx u - \Delta t\, f_x + \frac{(\Delta t)^2}{2}[a(u) f_x]_x,$$

where $u = u(x,t)$ above. Now, using the centeral difference approximations, we obtain

$$\begin{aligned} u_j^{k+1} \quad = \quad & u_j^k - \frac{\nu}{2}\left[f(u_{j+1}^k) - f(u_{j-1}^k) \right] \\ & + \frac{\nu^2}{2} \left\{ a(u_{j+1/2}^k) \left[f(u_{j+1}^k) - f(u_j^k) \right] - a(u_{j-1/2}^k) \left[f(u_j^k) - f(u_{j-1}^k) \right] \right\}, \end{aligned} \tag{7.14}$$

where $\nu = \Delta t / \Delta x$ and

$$u_{j+1/2}^k = \frac{u_{j+1}^k + u_j^k}{2}, \quad u_{j-1/2}^k = \frac{u_j^k + u_{j-1}^k}{2}.$$

7.2 Explicit methods for wave equation

In this section, we will consider the linear wave equation

$$u_{tt} = c^2(x) u_{xx}. \tag{7.15}$$

As in Chapter 6, we discretize x and t by x_j and t_k, respectively. The time step size is Δt and the spatial grid size is Δx. The following method is based on central difference approximations for both u_{tt} and u_{xx}. Let $u_j^k \approx u(x_j, t_k)$, we have

$$\frac{u_j^{k+1} - 2u_j^k + u_j^{k-1}}{(\Delta t)^2} = c^2(x_j) \frac{u_{j+1}^k - 2u_j^k + u_{j-1}^k}{(\Delta x)^2}.$$

Let

$$s_j = \left[\frac{c(x_j)\Delta t}{\Delta x} \right]^2,$$

then

$$u_j^{k+1} = (2 - 2s_j) u_j^k + s_j (u_{j+1}^k + u_{j-1}^k) - u_j^{k-1}.$$

This is an explicit 2-step (or three time-level) method. It is an explicit method. Sometimes, it is called the leap-frog method.

Equation (7.15) is solved with two initial conditions. If we start the equation at $t = 0$, then the two initial conditions are

$$u(x,0) = f(x), \quad u_t(x,0) = g(x). \tag{7.16}$$

For $t_0 = 0$, we have

$$u_j^0 = u(x_j, t_0) = f(x_j).$$

An approximation at t_1 can be obtained from the first three terms of the Taylor series:

$$u(x,t_1) \approx u(x,t_0) + (\Delta t) u_t(x,t_0) + \frac{(\Delta t)^2}{2} u_{tt}(x,t_0) = f(x) + (\Delta t) g(x) + \frac{(\Delta t)^2}{2} c^2(x) \partial_{xx} f(x).$$

With a central difference approximation, we obtain

$$u_j^1 = f(x_j) + g(x_j)\Delta t + \frac{(\Delta t)^2 c^2(x_j)}{2(\Delta x)^2} \left[f(x_{j-1}) - 2f(x_j) + f(x_{j+1}) \right].$$

Next, we perform a stability analysis assuming that $c(x)$ is a constant. When c is a constatn, we have $s = (c\Delta t/\Delta x)^2$. If we insert

$$u_j^k = \rho^k e^{i\beta x_j} = \rho^k e^{ij\tilde{\beta}}, \quad x_j = j\Delta x, \quad \tilde{\beta} = \beta \Delta x,$$

into the numerical method, we obtain

$$\rho = 2 - 2s + 2s\cos\tilde{\beta} - \frac{1}{\rho}.$$

This gives rise to

$$\rho^2 - 2\gamma\rho + 1 = 0,$$

where

$$\gamma = 1 - 2s\sin^2\frac{\tilde{\beta}}{2}.$$

The two solutions of ρ are

$$\rho = \gamma \pm \sqrt{\gamma^2 - 1}.$$

If $|\gamma| > 1$, then we always have one ρ such that $|\rho| > 1$, thus the method is unstable. If $|\gamma| \leq 1$, then

$$\gamma = \gamma \pm \sqrt{1-\gamma^2} i.$$

We can see that $|\gamma| = 1$ exactly. Therefore, the stability condition is $|\gamma| \leq 1$. Obviously, we have $\gamma \leq 1$, but we also need $\gamma \geq -1$. This must be true for any $\tilde{\beta}$. Thus, we can choose $\tilde{\beta} = \pi$, so that $\sin\frac{\tilde{\beta}}{2} = 1$. Then, the condition $\gamma \geq -1$ implies $s \leq 1$. That is

$$\Delta t \leq \frac{\Delta x}{c}. \tag{7.17}$$

Notice that this stability condition is not so restrictive as the stability condition of the classical explicit method (forward difference in time) for the heat equation. For the heat equation, we need Δt on the order of $(\Delta x)^2$ for stability. Here, we need Δt on the order of Δx for stability. In conclusion, the leap-frog method is **conditionally stable**, with the above stability condition. If c is a function of x, we interprete the above as

$$\Delta t \leq \frac{\Delta x}{\max c(x)}.$$

A generalization of this method for wave equation with two spatial variables is straight forward. Consider the wave equation

$$u_{tt} = c^2(x,y)\left[u_{xx} + u_{yy}\right]. \tag{7.18}$$

If we use the central difference approximation for all second derivatives in the above equation, we obtain

$$\frac{1}{(\Delta t)^2}\left[u_{ij}^{k+1} - 2u_{ij}^{k} + u_{ij}^{k-1}\right] = \frac{c^2(x_i,y_j)}{h^2}\left[u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j+1}^{k} - 4u_{ij}^{k}\right].$$

Here, we assume that $\Delta x = \Delta y = h$.

7.3 Maxwell's equations

The 2-D wave equation (7.18) is a special case of the Maxwell's equations for electromagnetic waves. Under some simplifying conditions, the Maxwell's equations are

$$\nabla \times E = -\mu \frac{\partial H}{\partial t} \tag{7.19}$$

$$\nabla \times H = \varepsilon \frac{\partial E}{\partial t}, \tag{7.20}$$

where E is the electric field, H is the magnetic field, t is the time, μ is the magnetic permeability and ε is the electric permittivity. In general, μ and ε are functions of the spatial variables: x, y and z. If we consider two dimensional case, we assume μ and ε are functions of x and y only, and E and H has no z-dependence (i.e. $\partial_z E = 0$ and $\partial_z H = 0$). In this case, there is a special two solution given by

$$E = \begin{bmatrix} 0 \\ 0 \\ E_z \end{bmatrix}, \quad H = \begin{bmatrix} H_x \\ H_y \\ 0 \end{bmatrix}.$$

That is to say, the electric field E has only one non-zero component in the z direction, the z-component of the magnetic field H is zero and the x and y components of H are non-zero. Here, E_z is not the partial derivative of E with respect to z, it is the z-component of the vector E. Similarly, H_x and H_y are the x- and y-components of H. Now, the Maxwell's equations can be simplified to

$$\mu \frac{\partial H_x}{\partial t} = -\frac{\partial E_z}{\partial y}, \tag{7.21}$$

$$\mu \frac{\partial H_y}{\partial t} = \frac{\partial E_z}{\partial x}, \tag{7.22}$$

$$\varepsilon \frac{\partial E_z}{\partial t} = \frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}. \tag{7.23}$$

We can eliminate H_x and H_y. This leads to

$$\frac{\partial^2 E_z}{\partial t^2} = c^2\left(\frac{\partial^2 E_z}{\partial x^2} + \frac{\partial^2 E_z}{\partial y^2}\right).$$

where $c = 1/\sqrt{\varepsilon\mu}$ is the speed of light in the medium. While the speed of light in vacuum (c_0) is a constant, here c is the speed of light in the medium and it is still a function of x and y.

For the Maxwell's equations, Kane Yee introduced a famuous numerical method based on central difference for first order derivatives in 1966. It is convenient to present this method in the first order system (7.21-7.23). Let us discretize t, x and y by the step size Δt and grid sizes Δx and Δy, respectively. Therefore,

$$t_k = t_0 + k\Delta t, \;\; x_i = x_0 + i\Delta x, \;\; y_j = y_0 + j\Delta y.$$

However, we also need to half steps and half grids. Namely,

$$t_{k+1/2} = t_0 + (k+0.5)\Delta t, \;\; x_{i+1/2} = x_0 + (i+0.5)\Delta x, \;\; y_{j+1/2} = y_0 + (j+0.5)\Delta y.$$

For E_z, we try to calculate its approximation at x_i, y_j and $t_{k+1/2}$. Namely,

$$E_z|_{ij}^{k+1/2} \approx E_z(x_i, y_j, t_{k+1/2}).$$

Similarly, we have

$$H_x|_{i,j+1/2}^{k} \approx H_x(x_i, y_{j+1/2}, t_k), \;\; H_y|_{i+1/2,j}^{k} \approx H_x(x_{i+1/2}, y_j, t_k).$$

The discretization of E_z, H_x and H_y are shown in Fig. (7.1). E_z are evaluated at the grid points marked by "o", H_x are evaluated at the gird points marked by "$\diamond$" and H_y corresponds to "$*$". With this type of

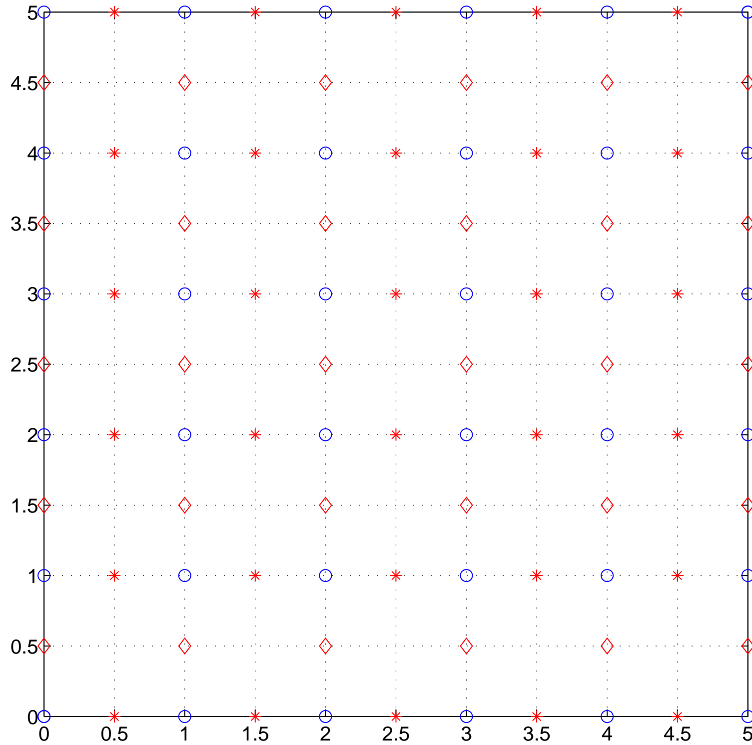

Figure 7.1: Discrete grid points for E_z (marked by "o"), H_x (marked by "$\diamond$") and H_y (marked by "$*$").

staggered grid, we can discretize the Maxwell's equations with second order finite difference method. Yee's finite difference time domain (FDTD) method is

$$\frac{\mu}{\Delta t}\left(H_x|_{i,j+1/2}^{k+1} - H_x|_{i,j+1/2}^{k}\right) = -\frac{1}{\Delta y}\left(E_z|_{i,j+1}^{k+1/2} - E_z|_{i,j}^{k+1/2}\right), \tag{7.24}$$

$$
\begin{aligned}
\frac{\mu}{\Delta t}\left(H_y|_{i+1/2,j}^{k+1} - H_y|_{i+1/2,j}^{k}\right) &= \frac{1}{\Delta x}\left(E_z|_{i+1,j}^{k+1/2} - E_z|_{i,j}^{k+1/2}\right), \qquad (7.25)\\
\frac{\varepsilon}{\Delta t}\left(E_z|_{ij}^{k+1/2} - E_z|_{ij}^{k-1/2}\right) &= \frac{1}{\Delta x}\left(H_y|_{i+1/2,j}^{k} - H_y|_{i-1/2,j}^{k}\right)\\
&\quad - \frac{1}{\Delta y}\left(H_x|_{i,j+1/2}^{k} - H_x|_{i,j-1/2}^{k}\right). \qquad (7.26)
\end{aligned}
$$

This is an explicit method, we can use (7.26) to calculate E_z at time level $t_{k+1/2}$, we can use (7.24) and (7.25) to calculate H_x and H_y at time level t_{k+1}. The method is in fact identical to the earlier method for a single scalar wave equation. However, the more general Yee's method for full Maxwell's equations cannot be written in a single unknown function. But this formulation using H_x, H_y and E_z allows us to treat boundary conditions easily.

Chapter 8

Finite Difference Methods for Elliptic PDEs

8.1 Finite difference method for Poisson equation

Consider the Poisson's equation in the unit square $\Omega = \{(x,y)|0 < x < 1, 0 < y < 1\}$:

$$u_{xx} + u_{yy} = F(x,y) \quad \text{for} \quad (x,y) \in \Omega$$

with some boundary condition $u = g$ for $(x,y) \in \partial\Omega$. Here, $\partial\Omega$ denots the four edges of the unit square and g is a function defined on $\partial\Omega$. We can discretize the problem with a second order finite difference method. Let $\delta = 1/(n+1)$ and $u_{ij} \approx u(ih, jh)$, we have the following discretized formula at (ih, jh):

$$\frac{u_{i-1,j} - 2u_{ij} + u_{i+1,j}}{\delta^2} + \frac{u_{i,j-1} - 2u_{ij} + u_{i,j+1}}{\delta^2} = F_{ij} = F(i\delta, j\delta).$$

or

$$-4u_{ij} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} = f_{ij} \tag{8.1}$$

where $f_{ij} = \delta^2 F_{ij}$. Notice that from the boundary conditions, we have known values for

$$u_{0j} = u(0, j\delta), \quad u_{n+1,j} = u(1, j\delta), \quad u_{i0} = u(i\delta, 0), \quad u_{i,n+1} = u(i\delta, 1).$$

Therefore, we have n^2 unknowns and we need n^2 equations. This is exactly what we have, if we choose $i = 1, 2, ..., n$ and $j = 1, 2, ..., n$ in (8.1). For the n^2 unknowns, we can order them in a large vector. For example, we can define

$$\vec{U} = [u_{11}, u_{21}, ..., u_{n1}, u_{12}, u_{22}, ..., u_{n2}, ..., u_{nn}]^T,$$

then (8.1) can be written as

$$A\vec{U} = \vec{b}, \tag{8.2}$$

where $\vec{b}$ is related to a b_{ij} which is related to F_{ij} and the boundary conditions. In fact,

$$b_{ij} = \delta^2 F_{ij} \quad \text{if} \quad 1 < i < n, \; 1 < j < n.$$

But near the boundary, i.e., $i = 1$ or $i = n$ or $j = 1$ or $j = n$, we need to include some terms from the boundary conditions.

$$\begin{aligned} b_{1j} &= \delta^2 F_{1j} - u_{0j} \quad \text{for} \quad 1 < j < n \\ b_{nj} &= \delta^2 F_{nj} - u_{n+1,j} \quad \text{for} \quad 1 < j < n \\ b_{i1} &= \delta^2 F_{i1} - u_{i0} \quad \text{for} \quad 1 < i < n \\ b_{in} &= \delta^2 F_{in} - u_{i,n+1} \quad \text{for} \quad 1 < i < n. \end{aligned}$$

At the four corners, we have to define b_{ij} to include two nearby points from the boundary:

$$\begin{aligned} b_{11} &= \delta^2 F_{11} - (u_{01} + u_{10}) \\ b_{n1} &= \delta^2 F_{n1} - (u_{n+1,1} + u_{n0}) \\ b_{1n} &= \delta^2 F_{1n} - (u_{0n} + u_{1,n+1}) \\ b_{nn} &= \delta^2 F_{nn} - (u_{n,n+1} + u_{n+1,n}). \end{aligned}$$

The vector $\vec{b}$ can be obtained from b_{ij} in the same way u_{ij} is ordered to give $\vec{U}$.

The coefficient matrix A is an $n^2 \times n^2$ matrix. It cannot be efficiently solved by Gaussian elimination directly, since the standard Gaussian elimination algorithm will require $O((n^2)^3) = O(n^6)$ operations. Actually, the matrix A has at most five non-zeros in each row and A has a bandwidth of $O(n)$. Using Gasssian elimination for banded matrices, the required number of operations is reduced to $O(n^4)$.

8.2 Fast Poisson solver based on FFT

Here, we describe a FFT based fast algorithm which requires only $O(n^2 \log n)$ operations. The method uses the discrete sine transform (which is related to the Discrte Fourier Transform) to obtain a system that can be easily solved.

The discrete sine transform is

$$\begin{aligned} g_j &= \sum_{k=1}^{n} \hat{g}_k \sin \frac{jk\pi}{n+1}, \quad j = 1, 2, ..., n, \\ \hat{g}_k &= \frac{2}{n+1} \sum_{j=1}^{n} g_j \sin \frac{jk\pi}{n+1}, \quad k = 1, 2, ..., n. \end{aligned}$$

If we introduce an $n \times n$ matrix S, whose (j,k) entry is $\sin \frac{jk\pi}{n+1}$, then

$$S^{-1} = \frac{2}{n+1} S.$$

Now, for u_{ij} and b_{ij}, we will fix i, then use discrete sine transform. We have

$$u_{ij} = \sum_{k=1}^{n} \hat{u}_{ik} \sin \frac{jk\pi}{n+1}, \quad b_{ij} = \sum_{k=1}^{n} \hat{b}_{ik} \sin \frac{jk\pi}{n+1}.$$

If we insert these into (8.2), we obtain

$$\begin{aligned} &\sum_{k=1}^{n} \left[-4\hat{u}_{ik} \sin \frac{jk\pi}{n+1} + \hat{u}_{i-1,k} \sin \frac{jk\pi}{n+1} + \hat{u}_{i+1,k} \sin \frac{jk\pi}{n+1} + \hat{u}_{ik} \sin \frac{(j-1)k\pi}{n+1} + \hat{u}_{ik} \sin \frac{(j+1)k\pi}{n+1} \right] \\ &= \sum_{k=1}^{n} \hat{b}_{ik} \sin \frac{jk\pi}{n+1}. \end{aligned}$$

This can be simplified to

$$\sum_{k=1}^{n}\left[(-4+2\cos\frac{k\pi}{n+1})\hat{u}_{ik}+\hat{u}_{i-1,k}+\hat{u}_{i+1,k}\right]\sin\frac{jk\pi}{n+1}=\delta^2\sum_{k=1}^{n}\hat{f}_{ik}\sin\frac{jk\pi}{n+1}.$$

Therefore,

$$(-4+2\cos\frac{k\pi}{n+1})\hat{u}_{ik}+\hat{u}_{i-1,k}+\hat{u}_{i+1,k}=\hat{b}_{ik}$$

For $i=1$ or $i=n$, the above equation should be modified to remove the term $\hat{u}_{i-1,k}$ or $\hat{u}_{i+1,k}$, respectively. Now, if we fix k, we can solve $\hat{u}_{ik}$ (for all i) from the above equation.

$$\begin{bmatrix}\alpha & 1 & & \\ 1 & \alpha & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & \alpha\end{bmatrix}\begin{bmatrix}\hat{u}_{1k}\\ \hat{u}_{2k}\\ \vdots \\ \hat{u}_{nk}\end{bmatrix}=\begin{bmatrix}\hat{b}_{1k}\\ \hat{b}_{2k}\\ \vdots \\ \hat{b}_{nk}\end{bmatrix}$$

for $\alpha=-4+2\cos\frac{k\pi}{n+1}$. This is a tridiagonal system and it can be solved in $O(n)$ operations. Since we have to do this for all k, the total operations required here is $O(n^2)$. But we first need to calculate $\hat{b}_{ik}$ from b_{ij}, based on the discrete sine transform. This can be done in $O(n^2\log n)$ operations. Once we found $\hat{u}_{ik}$, we can use discrte sine transform to find u_{ij}. Again, this requires $O(n^2\log n)$ operations. Since $n^2\log n$ is larger than n^2, the overall operations required to solve the Poisson equation is thus $O(n^2\log n)$. This is the FFT based “fast Poisson solver”.

8.3 Classical iterative methods

Although the FFT-based fast Poisson solver is very efficient, it cannot be generalized to more general equations with variable coefficients. Notice that the matrix A in Eq. (8.2) is sparse, i.e., most of its entries are zero and only a few non-zeros in each row or column. Iterative methods produce a sequence of approximate solutions that converge to the exact solution. Since the matrix A is sparse, it is possible to develop some iterative methods for solving $A\vec{U}=\vec{b}$.

We start by writing the matrix A as three parts: the diagonal D, the strictly lower triangular part $-L$ and the strictly upper triangular part $-R$, such that

$$A=D-L-R.$$

The minus sign in front of L and R are introduced for convenience. Now, $A\vec{U}=\vec{b}$ is identical to $D\vec{U}=(L+R)\vec{U}+\vec{b}$ and

$$\vec{U}=D^{-1}(L+R)\vec{U}+D^{-1}\vec{b},$$

This leads to the **Jacobi iterative method**:

$$\vec{U}^{(j+1)}=D^{-1}(L+R)\vec{U}^{(j)}+D^{-1}\vec{b},\quad j=0,1,2,... \tag{8.3}$$

We have to start with an initial guess $\vec{U}^{(0)}$, after that we can use (8.3) to calculate $\vec{U}^{(1)}$, $\vec{U}^{(2)}$, ... We can prove that for the finite difference approximation of the Poisson equation, i.e., for (8.2), Jacobi iteration converges. To prove the convergence, we need to show that all eigenvalues of $D^{-1}(L+R)$ have

magnitude less than 1. Meanwhile, we can also write (8.2) as $(D-L)\vec{U} = R\vec{U} + \vec{b}$. Therefore, we have the following **Gauss-Seidel iterative method**:

$$\text{solve } \vec{U}^{(j+1)} \text{ from } \quad (D-L)\vec{U}^{(j+1)} = R\vec{U}^{(j)} + \vec{b}. \tag{8.4}$$

Notice that $D-L$ is a lower triangular matrix, therefore, it is easy to solve a linear system with a coefficient matrix $D-L$. Again, for the discrete Poisson equation, we can prove that Gauss-Seidel iterative method converges. For this purpose, it is necessary to show that the eigenvalues of $(D-L)^{-1}R$ has megnitude less than 1. Finally, we can multiply (8.2) by a parameter ω, then add $D\vec{U}$ to both sides:

$$D\vec{U} + \omega(D-L-R)\vec{U} = D\vec{U} + \omega\vec{b}.$$

This can be written as

$$(D-\omega L)\vec{U} = [(1-\omega)D + \omega R]\vec{U} + \omega\vec{b}.$$

This leads to the following **Successive Overrelaxation (SOR) method** developed by Young and Frankel in 1950:

$$\text{solve } \vec{U}^{(j+1)} \text{ from } \quad (D-\omega L)\vec{U}^{(j+1)} = [(1-\omega)D + \omega R]\vec{U}^{(j)} + \omega\vec{b}. \tag{8.5}$$

For the discrete Poisson equation, SOR method converges if $0 < \omega < 2$. The optimal parameter is

$$\omega = \frac{2}{1+\sin(\pi\delta)},$$

where $\delta = 1/(n+1)$ is the grid size as in section 8.1. These three iterative methods are all classical iterative methods.

The **conjugate gradient** method, introduced by Hestenes and Stiefel in 1952, is a modern iterative method with a faster convergence rate. The discrete Poisson equation (8.2) can also be efficiently solved by the **multi-grid** method, where the numerical solutions with larger grid sizes are used to improve the approximation of the numerical solution at the smallest grid size.

8.4 Conjugate gradient method

The conjugate gradient method is a method for solving $Ax = b$, where A is a symmetric positive definite matrix. Here the size of A is large, thus a direct method by Cholesky decomposition (related to the LU decomposition) is expensive. But A is sparse — only very few non-zeros for each row or each column, thus it is efficient to multiply A with any given vector. It is an iterative method that produces the sequence of approximations: $x_1, x_2, x_3, \ldots$. Let A be $m \times m$, define the Krylov space by

$$\mathcal{K}_n = < b, Ab, A^2 b, \ldots, A^{n-1} b >$$

This is the vector space spanned by the vectors b, Ab, ..., $A^{n-1}b$. It is the "column space" of the Krylov matrix

$$K_n = [b, Ab, A^2 b, \ldots, A^{n-1} b].$$

The conjugate gradient method finds $x_n \in \mathcal{K}_n$ which solves the minimization problem

$$\min_{x \in \mathcal{K}_n} (x - x_*)^T A (x - x_*)$$

where $x_* = A^{-1} b$ is the exact solution. However, since

$$(x - x_*)^T A (x - x_*) = 2\phi(x) - b^T A^{-1} b, \quad \text{for} \quad \phi(x) = \frac{1}{2} x^T A x - x^T b.$$

It is equivalent to say that x_n solves

$$\min_{x \in \mathcal{K}_n} \phi(x).$$

8.4.1 1-D optimization problem

For a given point x_{n-1} and a given direction p_{n-1}, we have a line that passes through x_{n-1} along the direction of p_{n-1}. The points on the line are given by

$$x_{n-1} + \alpha p_{n-1} \quad \text{for} \quad \alpha \in \mathbb{R}$$

Alternatively, we denote this line by

$$x_{n-1} + < p_{n-1} >$$

where $< p_{n-1} >$ is a 1-D vector space. We can minimize the function ϕ along this line

$$\min_{x \in x_{n-1} + < p_{n-1} >} \phi(x) = \min_{\alpha \in \mathbb{R}} \phi(x_{n-1} + \alpha p_{n-1})$$

Now, $\phi(x_{n-1} + \alpha p_{n-1})$ is a quadratic polynomial of α, its minimum is reached at

$$\alpha_n = \frac{r_{n-1}^T p_{n-1}}{p_{n-1}^T A p_{n-1}}$$

The minimum is obtained at $x_{n-1} + \alpha_n p_{n-1}$.

If x_{n-1} happens to be a conjugate gradient iteration, i.e., x_{n-1} minimizes $\phi(x)$ in $\mathcal{K}_{n-1}$. The above procedure gives

$$\tilde{x}_n = x_{n-1} + \alpha_n p_{n-1}$$

Of course, $\tilde{x}_n$ is usually not x_n which minimizes ϕ in $\mathcal{K}_n$. However, we will find a special way of choosing p_{n-1}, such that $\tilde{x}_n = x_n$.

8.4.2 Subspace minimization problem

We now look for $x_n \in \mathcal{K}_n$ such that

$$\phi(x_n) = \min_{x \in \mathcal{K}_n} \phi(x)$$

We assume that $\mathcal{K}_n$ has the following basis

$$p_0, p_1, \ldots, p_{n-1}$$

Now,

$$\min_{x \in \mathcal{K}_n} \phi(x) = \min_{\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathbb{R}} \phi(\alpha_1 p_0 + \alpha_2 p_1 + \ldots + \alpha_n p_{n-1})$$

To find the minimum, we solve the system

$$\frac{\partial \phi}{\partial \alpha_i} = 0 \quad \text{for} \quad i = 1, 2, \ldots, n.$$

In fact,

$$\frac{\partial \phi}{\partial \alpha_i} = p_{i-1}^T A(\alpha_1 p_0 + \alpha_2 p_1 + \ldots + \alpha_n p_{n-1}) - p_{i-1}^T b$$

Therefore, we have the system for α_1, α_2, ..., α_n:

$$C \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} p_0^T b \\ p_1^T b \\ \vdots \\ p_{n-1}^T b \end{pmatrix}$$

where the $(i+1, j+1)$ entry of C is $p_i^T A p_j$.

If we assume that

$$p_i^T A p_j = 0 \text{ if } i \neq j$$

then the matrix C is diagonal and α_i is easily solved

$$\alpha_i = \frac{p_{i-1}^T b}{p_{i-1}^T A p_{i-1}}.$$

Furthermore, if we assume that $\{p_0, p_1, \ldots, p_{i-1}\}$ is a basis for $\mathcal{K}_i$ for all i (we only assume that for $i = n$ earlier), then

$$x_{n-1} = \alpha_1 p_0 + \alpha_2 p_1 + \ldots + \alpha_{n-1} p_{n-2}$$

is the conjugate gradient iteration that minimizes ϕ in $\mathcal{K}_{n-1}$ and

$$x_n = x_{n-1} + \alpha_n p_{n-1}$$

Indeed, you can show that the formula for α_n here is equivalent to the formula in last section. Therefore, the subspace minimization problem can be solved by 1-D optimization process under these assumptions on the search vectors p_0, p_1, ..., p_{n-1}.

8.4.3 Orthogonal residual

Clearly, we need a simple way to find these vectors p_0, p_1, ... It turns out that the following property on the residual is very important. *Let x_n be the n-th conjugate gradient iteration, $r_n = b - Ax_n$ be the residual, then*

$$r_n \perp \mathcal{K}_n.$$

8.4.4 The next conjugate direction

Suppose x_j is the conjugate gradient iteration that solves the subspace minimization problem $\min_{x\in\mathcal{K}_j} \phi(x)$, it is not difficult to realize that

$$\mathcal{K}_n =< x_1, x_2, ..., x_n >=< r_0, r_1, ..., r_{n-1} >$$

where $r_0 = b - Ax_0 = b$. We also assume that

$$\mathcal{K}_j =< p_0, p_1, ..., p_{j-1} > \quad \text{for} \quad j \le n$$

The question now is how to choose p_n, such that

- $\mathcal{K}_{n+1} =< p_0, p_1, ..., p_n >$;
- $p_n^T A p_j = 0$ for $j = 0$, 1, 2, ..., $n-1$.

To satisfy the first condition, we realize that $r_n = b - Ax_n$ is in $\mathcal{K}_{n+1}$ (and not in $\mathcal{K}_n$), therefore, we can choose

$$p_n = r_n + \text{a component in } \mathcal{K}_n$$

to satisfy the second condition. The component in $\mathcal{K}_n$ can be written as

$$\beta_n p_{n-1} + (*) p_{n-2} + ... + (*) p_0$$

since $\{p_0, p_1, ..., p_{n-1}\}$ is a basis of $\mathcal{K}_n$. We use the condition $p_j^T A p_n = p_n^T A p_j = 0$ (since $A = A^T$) to find the coefficients. For $j \le n-2$, we have

$$0 = p_j^T A p_n = p_j^T A r_n + (*) p_j^T A p_j$$

Now, $p_j^T A r_n = r_n^T (A p_j) = 0$, since $p_j \in \mathcal{K}_{n-1}$ or $A p_j \in \mathcal{K}_n$ (and $r_n \perp \mathcal{K}_n$ as in the last section), therefore, $(*) = 0$. Meanwhile, we obtain

$$p_n = r_n + \beta_n p_{n-1} \quad \text{for} \quad \beta_n = -\frac{r_n^T A p_{n-1}}{p_{n-1}^T A p_{n-1}}$$

8.4.5 The method

We now have the following conjugate gradient method:

- Let $x_0 = 0$, $r_0 = b$, $p_0 = r_0$.
- For $n = 1, 2, 3, ...$

$$\begin{aligned}
\alpha_n &= \frac{r_{n-1}^T r_{n-1}}{p_{n-1}^T A p_{n-1}} \\
x_n &= x_{n-1} + \alpha_n p_{n-1} \\
r_n &= r_{n-1} - \alpha_n A p_{n-1} \\
\beta_n &= \frac{r_n^T r_n}{r_{n-1}^T r_{n-1}} \\
p_n &= r_n + \beta_n p_{n-1}
\end{aligned}$$

We notice that the formulas for α_n and β_n are different. But they are equivalent to the formulas in previous sections. One step of this algorithm requires

- Evaluate $v = Ap_{n-1}$;
- $2m$ operations for $p_{n-1}^T v = p_{n-1}^T A p_{n-1}$;
- $2m$ operations for $x_n = x_{n-1} + \alpha_n p_{n-1}$;
- $2m$ operations for $r_n = r_{n-1} - \alpha_n v = r_{n-1} - \alpha_n A p_{n-1}$;
- $2m$ operations for $r_n^T r_n$;
- $2m$ operations for $p_n = r_n + \beta_n p_{n-1}$

This is a total of $10m$ operations, plus one matrix vector multiplication.

Exercise: Using the standard notations for the Conjugate Gradient method, where x_n is the n-th iteration of the approximate solution (for $Ax = b$, assuming $x_0 = 0$), r_n is the residual, p_n is the n-th A-conjugate direction, show that

$$\begin{aligned}
\alpha_n &= \frac{p_{n-1}^T b}{p_{n-1}^T A p_{n-1}} = \frac{r_{n-1}^T p_{n-1}}{p_{n-1}^T A p_{n-1}} = \frac{r_{n-1}^T r_{n-1}}{p_{n-1}^T A p_{n-1}} \\
\beta_n &= -\frac{r_n^T A p_{n-1}}{p_{n-1}^T A p_{n-1}} = \frac{r_n^T r_n}{r_{n-1}^T r_{n-1}}.
\end{aligned}$$

8.4.6 Rate of convergence

For a vector y and the symmetric positive definite matrix A, we define

$$||y||_A = \sqrt{y^T A y}.$$

Now, for the conjugate gradient method, we can prove

$$\frac{||x_n - x_*||_A}{||x_0 - x_*||_A} \leq 2 \left[\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right]^n \tag{8.6}$$

where $x_* = A^{-1} b$, $x_0 = 0$, $\kappa = \lambda_1 / \lambda_m$, λ_1 and λ_m are the largest and smallest eigenvalues of A, respectively.