Numerical Methods for Physicists

Anthony O'Hare

May 10, 2005

Contents

Preface

This handbook is intended to introduce the topic of solving mathematical problems that occur in physics using numerical techniques. It is written to accompany the Short Option in Numerical Methods (S17).

Introduction

Even simple physical situations generate equations for which no known closed-form solution exists, it is important for physicists to have a toolbox of numerical methods which can be used to tackle these sorts of problems. The methods outlined here are intended to develop and illustrate a minimum set of techniques that can be used for the most commonly encountered problems.

Useful Books and Web Links

The following list of books contain the topics that will be covered here. The first book on the list covers the material at the correct level and includes C++ code for several of the problems.

Numerical Methods For Physics, Alejendro Garcia

Numerical Analysis: Mathematics of Scientific Computing, Kincaid & Cheney

Computational Physics, Koonin & Meredith

Monte Carlo Methods, Volume 1: Basics, Kalos & Whitlock

Numerical Methods that work, Acton

The following websites also contain useful supplementary material, some of which has found it's waytn one form or another, into this manual.

Numerical Recipies in C/C++/Fortran/Pascal: Press, Veterling et. al. (www.nr.com)

http://www.damtp.cam.ac.uk/user/fdl/people/sd/lectures/nummeth98/contents.htm

http://hobbes.jct.ac.il/ naiman/nm/

Why should I use a numerical solution

Even simple physical situations often generate equations for which no known closed-form solution exists, it is important for physicists to have a toolbox of numerical methods which can be used to tackle these sorts of problems.

For example, a pendulum suspended by a fixed rod is described by the differential equation

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin(\theta) = 0$$

where g is the gravitational acceleration and l is length of the rod. The angle the pendulum makes with the vertical is θ. This is a classical 2^{nd} order differential equation but it is not linear since the second term contains a term proportional to $\sin(\theta)$. It is a common practise to use the 'small angle approximation' ($\sin(\theta) \approx \theta$ for small θ) to obtain a solution ($\theta(t) = \theta_0 \cos(\sqrt{\frac{g}{l}}t)$).

Where θ_0 is the angle through which we drop the pendulum initially. However if θ_0 is not small, the small angle approximation does not work, then we need a (numerical) method of solving this equation.

The boundary between analytical difficulty and numerical approximation becomes a bit blurred in a lot of cases and this is where a good knowledge of the problem is useful.

Normally, a numerical method is impervious to minor changes. We simply tweak a parameter or a line in our code. Such as changing our pendulum equation to remove the small angle approximation. However analytic methods depend heavily on any changes (e.g. $\frac{dy}{dx} + y\sin(x) = 0$ and $\frac{dy}{dx} + y\sin(x) = x$).

If the problem contains a mis-feature (e.g. discontinuity) a numerical solution may not even work but an analytic method (where one exists) will always give you a result.

In some cases there is no alternative other than to use a numerical method, for example the pendulum above, projectile motion when air resistance is taken into account, a simple model like the Ising model is difficult to solve in 2 dimensions but very easy to write a numerical simulation. In others the problem is unfeasably large, for example calculating the energy eigenvalues of a simple molecule can involve calculating eigenvalues of large matrices.

Errors in Numerical Methods

There are 3 main sources of errors in any numerical method.

Range Errors

Computers store individual floating point numbers in a small ammount of memory. Consequently, the accuracy of the number is determined by the ammount of memory set aside. Typical single precision numbers (`float` in C/C++) are allocated 4 bytes (32 bits) in memory while double precision number are allocated twice that ammount (64 bits). The exact details of how computers store the mantissa and exponent of floating point numbers is not important but it is important to know that there is a maximum range of these values.

A (32 bit) single precision number is accurate to about 7 decimal places and has a range of $-3.4 \times 10^{38} \ldots 3.4 \times 10^{38}$.

A (64 bit) double precision number is accurate to about 15 decimal places and has a range of $-1.7 \times 10^{308} \ldots 1.7 \times 10^{308}$.

It is easy to exceed the single precision limit so BE CAREFUL. It is best to work in the units natural to the problem you are solving e.g. for atomic problems work in angstroms, charge of the electron, for astrophysics you could use lightyears etc.

Truncation/Rounding Errors

Truncation or rounding errors can be introduced in a variety of ways.

You will remember the definition of the derivative of $f(x)$.

$$f'(x) = \frac{f(x+h) - f(x)}{h} \tag{0.0.1}$$

in the limit of $h \to 0$. If we evaluated the right hand side of the equation with $h = 0$ the computer could respond with a number of errors. Clearly, this is not useful. However if we set h to be something really small then we can get misleading results depending on how we define '*really small*'. Single precision numbers are only accurate to 6 or 7 decimal places and double precision are accurate to about 16 places, thus in double precision 3×10^{-20} would be evaluated as 3 and $h = 10^{-300}$ would almost certainly return 0 when evaluating the numerator in eq. 0.0.1.

We can define the absolute error as

$$\Delta h = \left| f'(x) - \frac{f(x+h) - f(x)}{h} \right| \tag{0.0.2}$$

You can see the behaviour of Δh in fig. 1. Notice that as h decreases Δh also decreases as we expect from eq. 0.0.1 but increases again, at $h \leq 10^{-16}$, as round-off errors affect the result. At the smallest values of h ($\leq 10^{-16}$) the errors are so large that the answer is worthless.

To test round off tolerance we define ϵ_r as the smallest number which, when added to 1 will give 1. The value of ϵ_r is defined in C/C++ as the `DBL_EPSILON` which is defined in `<float.h>` . On my computer DBL_EPSILON = 2.2204460492503130808 5e-16.

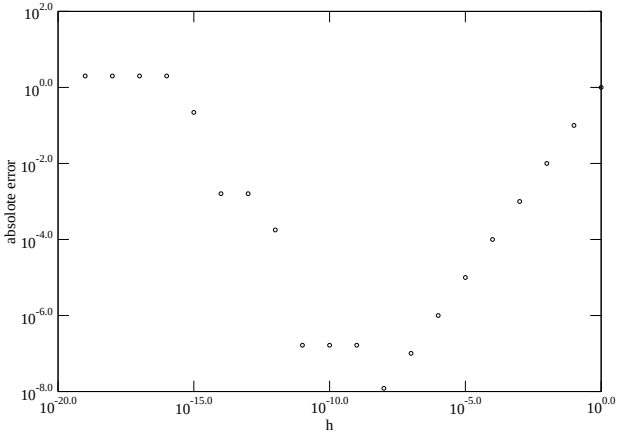


Figure 1: Absolute error (Δh, eq. 0.0.2) versus h for $f(x) = x^2$ at $x = 1$

If we are not careful in how we state a problem we can introduce errors that can accumulate through your program. For example, we can try and evaluate the following for small values of x

$$f(x) = \frac{1 - \cos(x)}{\sin(x)}$$

If we multiply the numerator and the denominator by $1 + \cos(x)$ we obtain a similar expresssion

$$f(x) = \frac{\sin(x)}{1 + \cos(x)}$$

Now, if we now choose a small $x = 0.007$ (in radians) then $\sin(0.007) = 0.69999 \times 10^{-2}$ and $\cos 0.007 = 0.99998$. The first expression for $f(x)$ gives

$$f(x) = \frac{1 - 0.99998}{0.69999 \times 10^{-2}} = \frac{0.2 \times 10^{-4}}{0.69999 \times 10^{-2}} = 0.28572 \times 10^{-2}$$

while the second expression gives

$$f(x) = \frac{0.69999 \times 10^{-2}}{1 + 0.99998} = \frac{0.69999 \times 10^{-2}}{1.99998} = 0.35 \times 10^{-2}$$

which is also the correct answer.

So what went wrong? In the first expression, after the subtraction we are left with only one relevant digit in the numerator because we subtracted two almost identical numbers, leading to

a loss of precision and ultimately the wrong result. If we had choosen a precision of six leading digits both expressions yield the same answer.

Conversely, if we were trying to evaluate $x \equiv \pi$ the second expression for $f(x)$ would have involved the cancellation of nearly equal numbers.

Bugs

In 1947 at Harvard University, engineers working on an early computer discovered an error in the computer output. After investigating the problem they discovered that a (dead) moth was stuck in one of the components. They stuck the moth in their logbook and labelled it the "first actual case of bug being found". Although, 'bugs' of this sort are unheard of today the term still refers to unintentional errors produced in computer code.

Bugs, today, refer to errors in computer code which can manifest themselves in a variety of ways. You need a bit of common sense to interpret the results that a computer program produces. How do you know if you've discovered something new or if you've got a bug in your program giving you the wrong answer? Computational physics can be likened to experimental physics in the sense that debugging code is a slow tedious but necessary task similar to aligning measuring instruments or fixing lab equipment.

In case you're still not convinced about the need to understand the cause and hence the elimination of errors these high profile numerical disasters should convince you.

> The Patriot Missile failure, in Dharain, Saudi Arabia, on February 25, 1991 which resulted in 28 deaths, is ultimately attributable to poor handling of rounding errors.

> The explosion of the Ariane 5 rocket just after lift-off on its maiden voyage off French Guiana, on June 4, 1996, was ultimately the consequence of a simple overflow.

> The loss of the Mars Climate Orbiter in 1999 was found to be the fault of a mix up between the use of pounds and kilograms as a measure for mass.

Important Formulae

Most of the methods that we will derive in this course are derived themselves from the Taylor series in one form or another. Since, it is so important I will (informally) derive it here.

Suppose we have a function $f(x_0+h)$ that we wish to expand in a power series of h, that is we wish to write

$$f(x_0+h) = a_0 + a_1 h + a_2 h^2 + a_3 h^3 + \ldots \tag{0.0.3}$$

Of course we need to determine the coefficients a_n.

We start tby setting $h \equiv 0$. This gives us

$$a_0 = f(x_0)$$

If we now differentiate eq(0.0.3) with respect to h, we see that

$$f'(x_0+h) = a_1 + 2a_2 h + 3a_3 h^2 + \ldots$$

and again setting $h \equiv 0$ we obtain

$$a_1 = f'(x_0)$$

If we further differentiate eq(0.0.3) with respect to h, we get

$$f''(x_0+h) = 2a_2 + 6a_3 h + \ldots$$

and setting $h \equiv 0$ we obtain

$$a_2 = \frac{f''(x_0)}{2}$$

We can continue this process to obtain a general formula for a_n

$$a_n = \frac{f^{(n)}(x_0)}{n!}$$

Now we are in a position to write out the power series in full

$$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{h^n}{n!} f^n(x_0) \tag{0.0.4}$$

We can define another (similar) Taylor expansion

$$f(x_0-h) = f(x_0) - hf'(x_0) + (-1)^n \frac{h^n}{n!} f^n(x_0) \tag{0.0.5}$$

which will prove useful later.

At this point we will introduce a notation that is found in several textbooks; $f_0 \equiv f(x_0)$, $f_1 \equiv f(x_0+h)$, $f_2 \equiv f(x_0+2h)$, $f_{-1} \equiv f(x_0-h)$ and $f_{-2} \equiv f(x_0-2h)$ and so on. Let's rewrite the Taylor series in the new notation:

$$\begin{aligned} f_1 &= f_0 + hf_0' + \frac{h^n}{n!} f_0^n \\ f_{-1} &= f_0 - hf_0' + (-1)^n \frac{h^n}{n!} f_0^n \end{aligned}$$

If we subtract the expansion for f_{-1} from f_1 (using our new notation) we get (after a little rearranging)

$$f_0' = \frac{f_1 - f_{-1}}{2h}$$

which we can use as an approximation for the derivative of $f(x)$ at $x = x_0$. This "3-point" approximation is called the *centered difference approximation* and would be exact if $f(x)$ were a second degree polynomial in the interval $[-h, h]$ since all third and higher order derivatives would vanish in the Taylor expansion. If we add the expansion for f_1 and f_{-1} we see that the first order derivative disappears and we can get an expression for the second derivative of $f(x)$ at $x = 0$.

$$f_0'' = \frac{f_1 - 2f_0 + f_{-1}}{h^2}$$

We can also use this expansion to write down the forward and backwards difference relations

$$f' = \frac{f_1 - f_0}{h}$$
$$f' = \frac{f_0 - f_{-1}}{h}$$

We can rewrite the Taylor series as an expansion around points a distance $2h$ from x_0 by replacing x_0 with $x_0 = \pm h$ 0.0.5):

$$\begin{aligned} f(2h) &= f(0) + 2hf'(0) + 2h^2 f'' + \mathcal{O}(h^3) \\ f(-2h) &= f(0) - 2hf'(0) + 2h^2 f'' + \mathcal{O}(h^3) \end{aligned}$$

We can combine these expansions with that of those for $f(h)$ and $f(-h)$ and cancel the second and higher order derivatives to obtain a 5-point approximation:

$$f' = \frac{f_{-2} - 8f_{-1} + 8f_1 - f_2}{12h}$$

and similarly

$$f'' = \frac{-f_{-2} + 16f_{-1} - 30f_0 + 16f_1 - f_2}{12h^2}$$

Higher order approximations can be derived by cancelling the appropriate terms of the expansions for $f(x \pm nh)$.

1

Finding Roots of Equations

We will start our numerical methods with some simple methods. Calculating roots of an equation is a problem that is, fortunately, easy to understand conceptually and will introduce some useful concepts of numerical methods.

Suppose we are given a function $f(x)$ and we wish to solve for x where $f(x) = 0$, for example $\sin(x) = 0$. We will look at three methods of solving this equation and discuss the pro's and cons of each method. This will give you a hint on how to 'skin your particular cat'.

Newton's method: fast but may not always work

Bisection method: slower but guaranteed to work in most circumstances.

Linear Interpolation

1.1 Newton's Method

This is sometimes referred to as the Newton-Raphsen method and is based on the Taylor series.

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + \mathcal{O}(h^3)$$

To solve our equation $f(x) = 0$ we substitute this into our Taylor series.

$$0 = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + \mathcal{O}(h^3)$$

If x_0 is close to the root x, then we have

$$\begin{aligned}(x - x_0) &\text{ is small}\\ (x - x_0)^2 &\text{ is much smaller}\\ (x - x_0)^3 &\text{ is much smaller still}\end{aligned}$$

which allows us to ignore the quadratic and higher terms leaving

$$f(x_0) + (x - x_0)f'(x_0) \approx 0$$

or, rearranging

$$x \approx x_0 - \frac{f(x_0)}{f'(x_0)} \tag{1.1.1}$$

Which we can iterate as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1.1.2}$$

This means that if we have a starting point (a guess) for x_0 we can iterate [1.1.2] until we find the root x.

Suppose we have $f(x) = x^2$, we know that a root is the origin and that $\frac{f(x)}{f'(x)} = \frac{x}{2}$. We can apply the Newton-Raphsen formula (1.1.2) with a starting point of $x = 0.1$ and obtain

$$\begin{aligned}x_{n+1} = &\quad x_n - \tfrac{x}{2}\\ = &\quad 0.1 - 0.05 = 0.05\\ = &\quad 0.05 - 0.025 = 0.025\\ = &\quad 0.025 - 0.0125 = 0.0125\\ = &\quad 0.0125 - 0.00625 = 0.00625\end{aligned}$$

which is indeed approaching our root of $x = 0$.

We can interpret x_{n+1} here as the point where the tangent to $f(x_n)$ crosses the x axis.

There are, however, instances where Newton's method doesn't work. Obvious candidates are cases where the function does not have a root or where the functions derivative does not exist at the root, e.g. $f(x) = x^{1/3}$, here $f'(x) = \frac{1}{3x^{2/3}}$ and $\frac{f(x)}{f'(x)} = 3x$ and use $x = 0.1$ we now get

$$\begin{aligned}x_{n+1} = &\quad x_n - 3x\\ = &\quad 0.1 - 0.3 = -0.2\\ = &\quad -0.2 + 0.6 = 0.4\\ = &\quad 0.4 - 1.2 = -0.8\\ = &\quad -0.8 + 2.4 = 1.6\end{aligned}$$

which is not approaching any root!

The assumption that x_0 be close to x, which we made in deriving this formula is the major drawback to this method. You need to know roughly where the root is before you find it!

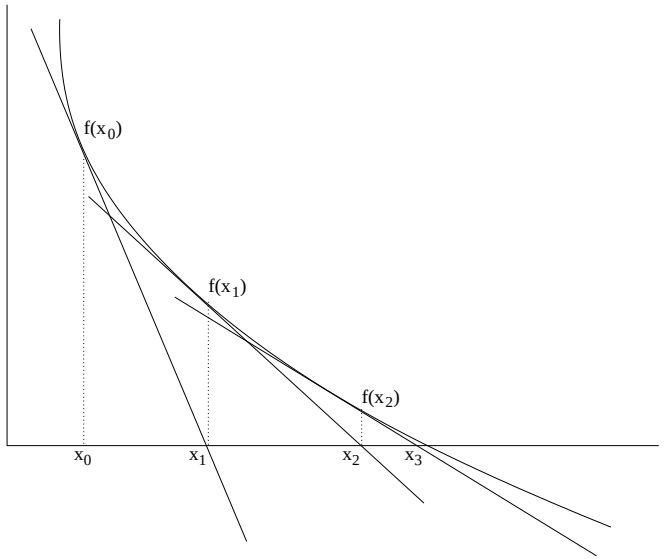


Figure 1.1: Newton-Raphsen method for finding the root of a single variabled function. The point where the tangent crosses the x-axis is the starting point for the next iteration.

1.2 Bisection Method

An alternative method of solving $f(x) = 0$ is the bisection method. It is generally slower that Newton's method but will converge for a wider range of functions.

The essential conditions for this method to work are:

$f(x)$ is continuous in $[a, b]$

$f(a)f(b) \leq 0$ (opposite signs)

Suppose we are given a continuous function $f(a) > 0$ and $f(b) < 0$. We will try and find a value of x in $[a, b]$ that satisfies $f(x) = 0$. It is obvious that there is a point between a and b at which $f(x)$ passes through zero as it changes sign.

The algorithm for the Bisection method is easy:

1. Let $c = \frac{a+b}{2}$.
2. If $f(c) = 0$ then c is an exact solution.

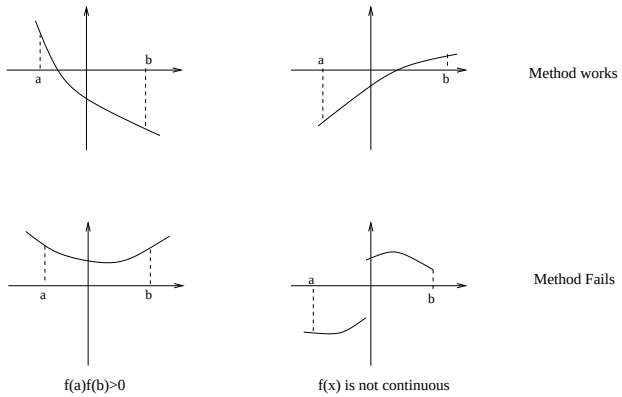


Figure 1.2: Conditions where the bisection method will work and where it fails

3. If $f(a) < 0$ and $f(c) > 0$ then the root lies in $[a, c]$.
4. else the root lies in $[c, b]$.

By replacing the interval $[a, b]$ with either $[a, c]$ or $[c, b]$, we can repeat the above process until we reach the root or the interval is smaller than a desired accuracy.

There is an obvious drawback to this method. You need to know roughly where the root you are interested in is before you start using the bisection method. Just because $f(a)$ and $f(b)$ have opposite signs only tells us that there are an odd number of roots in $[a, b]$ and, conversely, if they have the same sign there are either an even number of roots or none. For example, if we want to find the roots of $f(x) = \sin(x)$ between $x = 1$ and $x = 10$ we would find that $f(1) > 0$ and $f(10) < 0$ but we would not realise that there are 3 roots (at $x = \pi, 2\pi, 3\pi$), the bisection method would only find the root close to $x = 3$. Prove to yourself this is the case by walking through the bisection method.

This is a good example where the weakness of numerical methods exist. We cannot just switch our brain off and simply follow a method. As physicists we need to be able to use numerical methods as tools but always to keep in mind the goal we are looking for and understand the weaknesses in the tools we use.

1.3 Linear Interpolation

This method is similar to the bisection method since it requires a value on either side of the root. Instead of dividing the root in two a linear interpolation is used to obtain a new point which is (not necessarily) closer to the root than the same step of the bisection method.

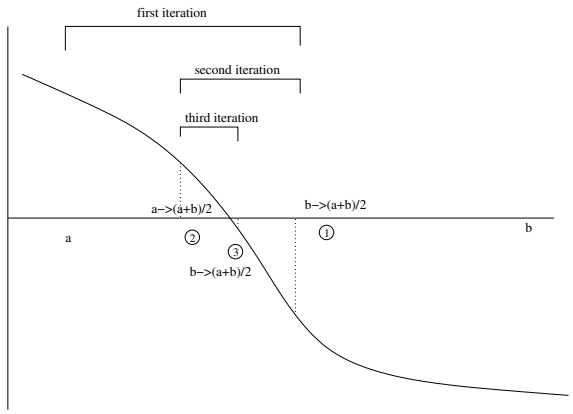


Figure 1.3: The Bisection Method: After examining $f(a)$ and $f(b)$ we find the root to lie in $[a,(a+b)/2]$ so we set $b=(a+b)/2$. A second iteration shows that the root lies in $[(a+b)/2,b]$ so we now set $a=(a+b)/2$. These steps can be repeated until we find the root to a required accuracy.

Linear interpolation works by drawing a straight line between two points and returning the appropriate point along the straight line. We can write the point c as

$$c = a - \frac{b-a}{f(b)-f(a)}f(a) = b - \frac{b-a}{f(b)-f(a)}f(b) = \frac{af(b)-bf(a)}{f(b)-f(a)}$$

1. Let $c = a - \frac{b-a}{f(b)-f(a)}f(a) = b - \frac{b-a}{f(b)-f(a)}f(b) = \frac{af(b)-bf(a)}{f(b)-f(a)}$
2. if $f(c)=0$ then c is an exact solution.
3. If $f(a)<0$ and $f(c)>0$ then the root lies in $[a,c]$.
4. else the root lies in $[c,b]$.

This is often referred to as the false position method.

Similar to the bisection method we continue implementing this method until we reach the root or the interval is smaller than a desired accuracy. Since we are using linear interpolation, it should be obvious that if the function $f(x)$ is linear then this method will be exact.

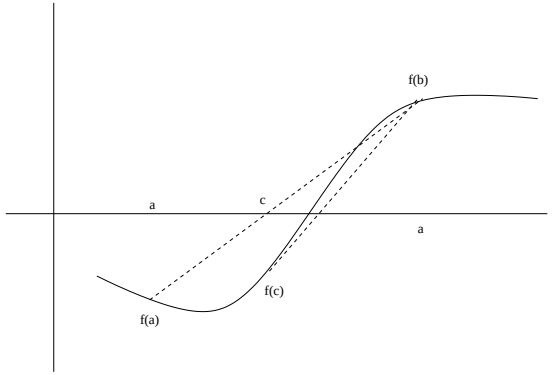


Figure 1.4: Linear Interpolation: This method works by drawing a line from one end of the interval to the other and calculating the intercept, c, with the axis. Depending on the value of $f(c)$ we can redefine our interval.

2

Fitting Curves to Data

In many areas of physics data analysis plays an important role. Experiments, observations and simulations can produce data for which a polynomial fit needs to be found or perhaps data need to be extrapolated from recorded data. In this lecture we will look at methods of fitting polynomials to data.

2.1 Linear Interpolation

Between any two points $(x_0, y_0), (x_1, y_1)$ there is a unique straight line, between any three points a unique quadratic etc. I will start with the, almost trivial, straight line. It will give us an insight into how we can interpolate data and ultimately how to fit curves to data.

A straight line is given by the formula

$$y = ax + b \tag{2.1.1}$$

where a is the slope of the line and b is the value of y when x is zero. Given two points $(x_0, y_0), (x_1, y_1)$ as in fig 2.1 we can calculate the slope of line as

$$\frac{y_1 - y_0}{x_1 - x_0}$$

.

However, before we blindly insert this into . 2.1.1 we need to think about the origin of our axes. In the formula for the straight line the (x, y) assumes the origin is at x_0 in fig 2.1. In translating the origin from $x = 0$ to $x = x_0$ we set $x = x - x_0$. This gives us

$$y = \frac{x - x_0}{x_1 - x_0}(y_1 - y_0) + y_0 \tag{2.1.2}$$

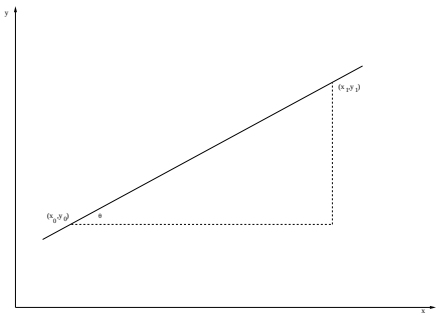


Figure 2.1: Through any two points there is a unique line ($y = ax + b$) that joins the points

which we can use to find any y along the line connecting (x_0, y_0) and (x_1, y_1).

This can easily be extended to a higher polynomial to a set of n data points, $(x_i, y_i) \ldots (x_n, y_n)$, giving Lagrange's formula of polynomial interpolation. I will just present the equation

$$y = \sum_{i=1}^{n} y_i l_i(x) \quad \text{where} \quad l_i(x) = \Pi_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} \tag{2.1.3}$$

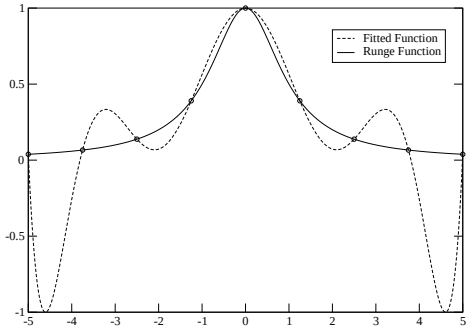


Figure 2.2: Fitting an n^{th} order polynomial through data points can fail for a highly oscillatory function such as the Runge function ($f(x) = (1 + x^2)^{-1}$).

Great, we can now fit an n^{th} order polynomial through n points. What can go wrong. Fig 2.2 shows a 9^{th} order polynomial fitted to the Runge function ($f(x) = (1 + x^2)^{-1}$). The moral of this example: Blindly fitting polynomials to data can be misleading since between data points the function can oscillate wildly.

2.2 Fitting Straight Lines

Interpolation is closely related to our next topic. We can now fit an n^{th} order polynomial to n data points. However, we have also seen that this is not, usually, a very clever idea. Can we find the best polynomial to fit our data? The answer is yes, but let us start with fitting a straight line. This will allow us to develop the techniques needed but not tie us up in algebra. You will probably have fitted a 'line of best fit' in an experimental lab, here we will see how it works.

Given a set of data (x_i, y_i) from, say an instrument reading or calculation, we will try and fit a straight line of the form

$$y = ax + b \tag{2.2.1}$$

to the data.

The object is the find the deviation of each y_i from the straight line and to minimise this deviation.

The squared deviation of the measurements (y_i) from the fitted straight line is given by:

$$d^2 = \sum_{i=1}^{N} (y_i - b - ax_i)^2 \tag{2.2.2}$$

Where N is the number of measured points. The values of a and b that minimise the squared deviation are found by differentiating d^2 with respect to a and b and setting the derivatives equal to 0 to obtain:

$$\begin{aligned} \frac{\partial(d^2)}{\partial a} &= 2\sum_{i=1}^{N}(y_i - b - ax_i)x_i = 0 \\ \frac{\partial(d^2)}{\partial b} &= 2\sum_{i=1}^{N}(y_i - b - ax_i) = 0 \end{aligned} \tag{2.2.3}$$

or, with a little re-arranging and noting that $\sum_{i=1}^{N} b = bN$,

$$\begin{aligned} a\sum x_i^2 + b\sum x_i &= \sum x_i y_i \\ a\sum x_i + bN &= \sum y_i \end{aligned} \tag{2.2.4}$$

Solving these equation we obtain expression for a and b.

$$\begin{aligned} a &= \frac{N\sum x_i y_i - \sum x_i \sum y_i}{N\sum x_i^2 - (\sum x_i)^2} \\ b &= \frac{\sum y_i - a\sum x_i}{N} \end{aligned}$$

Using these values of a and b we can calculate the fitted value of any y given an x.

Table 2.1: Data taken from a fictitious experiment. The first column represents a fixed variable while the second represents the measured value.

1	1.67711
2	2.00994
3	2.26241
4	2.18851
5	2.33006
6	2.42660
7	2.48424
8	2.63729
9	2.77163
10	2.89610
11	2.89083
12	3.08081
13	3.05305
14	3.24079
15	3.36212

Example:
Find the best straight line fit to the data in table 2.2.
From the data we have

$$\begin{aligned} \sum x_i &= 120.0 \\ \sum y_i &= 39.31149 \\ \sum xx_i &= 1240.0 \\ \sum xy_i &= 344.103190 \end{aligned} \tag{2.2.5}$$

giving

$$\begin{aligned} a &= 0.105755 \\ b &= 1.774730 \end{aligned} \tag{2.2.6}$$

The graph of the data points and the fitted line is shown in fig 2.3.

2.3 Fitting Polynomials

Fitting a higher order polynomial follows the same procedure, calculate the squared deviation in the measured values and find the parameters that minimise the deviation.

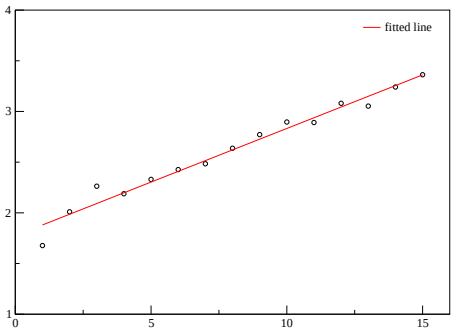


Figure 2.3: A straight line $y = ax + b$ can be fit to a set of data points. Here $a = 0.105755$ and $b = 1.774730$.

Here we will try to fit the polynomial

$$y = c_0 + c_1 x_j + c_2 x_j^2 + c_3 x_j^3 + ... + c_m x_j^m \tag{2.3.1}$$

to a set of data points (x_i, y_i).

Similar to fitting a straight line we find the coefficients c_j that minimise the squared deviation of the measurements (y_i) i.e.

$$d^2 = \sum_{j=1}^{N} (y_j - (c_0 + c_1 x_j + c_2 x_j^2 + c_3 x_j^3 + ... + c_m x_j^m))^2 \tag{2.3.2}$$

If, as in the case of the straight line, we minimise d^2 with respect to each coefficient c_k we get a system of $m + 1$ linear equations in $N + 1$ unknowns:

$$v_k = \sum_{i=1}^{m} A_{ki} c_i \tag{2.3.3}$$

where

$$v_k = \sum_{j=1}^{n} y_j x_j^k$$
$$A_{ki} = \sum_{j=1}^{n} x_j^{i+k} \tag{2.3.4}$$

Here m is the degree of the polynomial we want to fit and N is the number of data points that we wish to fit to. N and m need not have numerical value, indeed, bad fits occur when they are. There is an example of this a little later. The subscripts k and i iterate from 1 to m.

We will discuss methods of solving systems of equations such as these in a later lecture.

Table 2.2: Data taken from a fictitious experiment. The first column represents a fixed variable while the second represents the measured value.

-2	5.94824
-1	3.13729
0	2.17163
1	2.89610
2	6.08081

Example:
Given a data file containing 5 data points x_i and y_i (as in Table 2.3) we wish to fit polynomial of degree 3 to the data.
We have the following equations to solve.

$$\begin{bmatrix} & \cdots & \\ \vdots & A_{ki} & \vdots \\ & \cdots & \end{bmatrix} \begin{bmatrix} \vdots \\ c_i \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ v_k \\ \vdots \end{bmatrix}$$

The vector c is the vector of coefficients for which we have to solve. The k^{th} element of the v vector is given by $\sum_{j=1}^{n} x_j^k y_j$ giving

$$\begin{aligned}(-2^0)*(5.94824)+(-1^0)*(3.13729)+(0^0)*(2.17163)+(1^0)*(2.89610)+(2^0)*(6.08081)\\ = 17.89081\\ (-2^1)*(5.94824)+(-1^1)*(3.13729)+(0^1)*(2.17163)+(1^1)*(2.89610)+(2^1)*(6.08081)\\ = 0.02395\\ (-2^2)*(5.94824)+(-1^2)*(3.13729)+(0^2)*(2.17163)+(1^2)*(2.89610)+(2^2)*(6.08081)\\ = 54.14959\end{aligned}$$

We calculate the k, i^{th} elements of the A matrix from $\sum_{j=1}^{n} x_j^{i+k}$ where the x_j is given from table 2.3 giving

$$\begin{bmatrix} \sum_{j=1}^m x_j^0 & \sum_{j=1}^m x_j^1 & \sum_{j=1}^m x_j^2 \\ \sum_{j=1}^m x_j^1 & \sum_{j=1}^m x_j^2 & \sum_{j=1}^m x_j^3 \\ \sum_{j=1}^m x_j^2 & \sum_{j=1}^m x_j^3 & \sum_{j=1}^m x_j^4 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 10 \\ 0 & 10 & 0 \\ 10 & 0 & 34 \end{bmatrix}$$

Our system of equations to solve is now

$$\begin{bmatrix} 5 & 0 & 10 \\ 0 & 10 & 0 \\ 10 & 0 & 34 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 17.89081 \\ 0.02395 \\ 54.14959 \end{bmatrix}$$

We will discuss in a later chapter how to solve systems of equations. In the meantime Maple commands to solve this system of equations is

```
> with(linalg):
> A := <<5|0|10>,<0|10|0>,<10|0|34>>;
> v := <17.89081, 0.02395,54.14959>;
> linsolve(A,v);
```

The results are plotted in fig 2.4, the coefficients are:

$$c_0 = 0.954166, c_1 = 0.002395, c_2 = 1.311998$$

We have seen previously that fitting an n^{th} order polynomial to n data points does not give a good fit since a large order polynomial fit may oscillate wildly between data points.

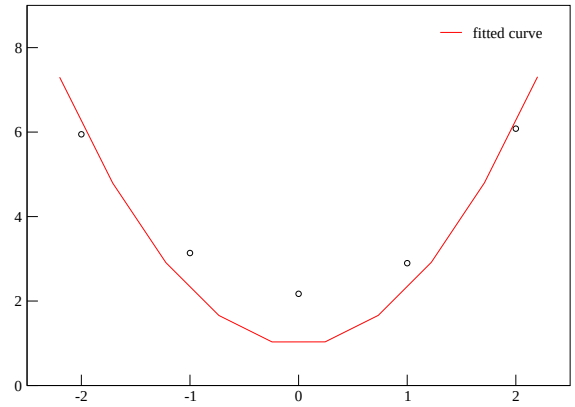


Figure 2.4: Here we fit a 2nd order polynomial to 5 data points and we can see that the fit is quite good. More data points would have provided a better fit. Blindly increasing the order of the polynomial does not, however, provide a better fit.

Our system of equations now becomes

$$\begin{bmatrix} 5 & 0 & 10 & 0 & 34 \\ 0 & 10 & 0 & 34 & 0 \\ 10 & 0 & 34 & 0 & 130 \\ 0 & 34 & 0 & 130 & 0 \\ 34 & 0 & 130 & 0 & 514 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} 17.89081 \\ 0.02395 \\ 54.14959 \\ 0.81937 \\ 198.49819 \end{bmatrix}$$

The solutions are

$$\begin{aligned} c_0 &= -0.17163 \\ c_1 &= -0.17184 \\ c_2 &= 3.73558 \\ c_3 &= 0.0512458 \\ c_4 &= -0.547262 \end{aligned}$$

which are plotted in fig 2.5

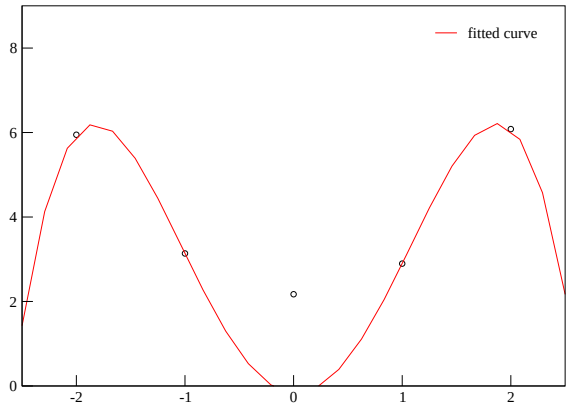


Figure 2.5: Highly oscillatory functions provide a bad fit if the data does not oscillate very much. Here we fit a 5^{th} order polynomial to 5 data points and we can see that the fit is less than perfect

3

Quadrature

Quadrature is the term used to describe the process of integration by a numerical method. Strictly speaking, the term 'numerical integration' that is found in many textbooks is wrong. One can distinguish three classes of integration problems:

1. quadrature, i.e. computation of definite integrals in one or more dimensions;
2. solution of ordinary differential equations (O.D.E.s),
3. solution of partial differential equations (P.D.E.s).

In this chapter we will look at simple methods of solving the first of these problems i.e. techniques for solving

$$\int_a^b f(x)dx \tag{3.0.1}$$

numerically. We will first consider the simplest case, a 1D finite interval [a,b] and a continuous function. At the end of the chapter we will briefly look at methods of dealing with discontinuities.

The simplest idea, and one that is introduced in high school, is to calculate the area under the curve $f(x)$ by using rectangles (Riemann sums) as in fig 3.1

Each rectangle has a base of h and a height of $f(a+nh)$, so our simplest method can be written as the sum

$$I = \sum_{i=0}^{N} f(a+ih)h$$

It is easy to see that the accuracy is proportional to the number of rectangles we choose. In numerical analysis, we call the length of the base of the rectangle the step size (often written as

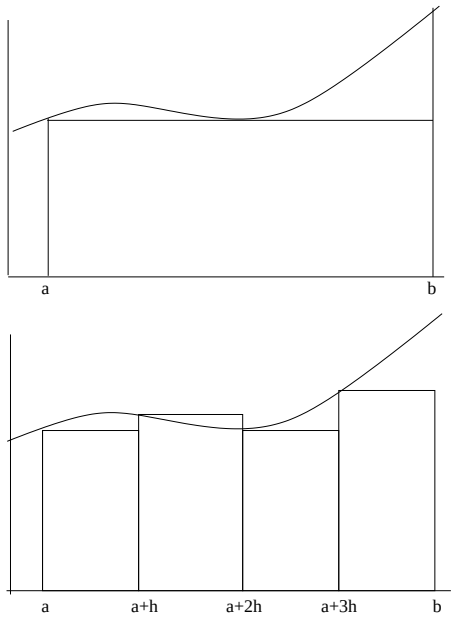


Figure 3.1: A simple integral may be split into several subintervals in which we can calculate the area of each box and sum the results. This simple method of calculating an integral is known as Riemann's method.

h). In fact to double the accuracy of our approximation we need half the step size (take twice the number of steps), so we say this method has an error of the order of $\mathcal{O}(h)$. We generally look for method that have a higher order of error since doubling the number of steps we take doubles the number of calculations we need to make.

This might seem like a trivial method but it has introduced the ideas of step size and accuracy. Our aim is to get the best accuracy for the largest step size, this allows us to use fewer steps (and less computations) for a given accuracy.

3.1 Newton Côtes Formulae

Newton Côtes methods are actually a family of methods for quadrature problems. The basic plan is to split the interval $a \leq x \leq b$ up into N equal sections each of length $h = (b-a)/N$. We will label these points x_i with i running from 0 to N, $x_0 = a$ and $x_N = b$. For brevity we will denote $f(x_i)$ as f_i and the center points of each interval $x = x_i + h/2$ by $x_{i+h/2}$.

The plan of the Newton Côtes formulae is to approximate the function in the subinterval by

a function that we can integrate exactly. In the following sections we will approximate the function by a straight line and a quadratic function and develop an integration technique. It should be obvious that if the function is actually a straight line or a quadratic in any interval then we will have integrated our function exactly in that interval.

Let us now develop an exact form of the integral even though we cannot calculate it. Even though we cannot calculate the integral by this method it will provide a neat method of finding the error in the methods we will soon develop.

Consider one subinterval, say between x_i and x_{i+1}. We will expand the function in this interval using Taylors series.

$$\begin{aligned}\int_{-h/2}^{h/2} f(x_{i+1/2}+y)dy &= \int_{-h/2}^{h/2}\sum_{n=0}^{\infty} f^{(n)}(x_{i+1/2})\frac{y^n}{n!}dy \\ &= \sum_{n=0}^{\infty} f^{(n)}(x_{i+1/2})\int_{-h/2}^{h/2}\frac{y^n}{n!}dy \\ &= \sum_{n=\text{even}}^{\infty} f^{(n)}(x_{i+1/2})\frac{2}{(n+1)!}\left(\frac{h}{2}\right)^{n+1} \qquad (3.1.1)\end{aligned}$$

Only the even derivatives of f survive the integration and the derivatives are evaluated at $x_{i+1/2}$.

We will now look at practical methods of calculating integrals based on this method.

3.1.1 Trapezium Rule

We will firstly approximate the function in each interval by a straight line. This means that every small element under $f(x)$ looks like a trapezium and as you may have guessed this method is called the trapezium rule.

Geometrically, this is a very easy rule to understand and to derive. We begin by spitting our interval into subintervals of width h as shown in fig 3.2. The area under the curve is simply the sum of the areas the rectangle $ABCE$ and the triangle CDE. Thus we have as the integral in this subinterval

$$\begin{aligned}\int_{x_i}^{x_{i+1}} f(x)dx &= \text{area of ABCE} + \text{area of CDE} \\ &= hf(x_i) + \frac{h}{2}\left(f(x_{i+1}) - f(x_i)\right) \\ &= \frac{h}{2}\left(f(x_i) + f(x_{i+1})\right) \qquad (3.1.2)\end{aligned}$$

So by adding up the areas of each subinterval in $[a,b]$ we have an expression that is referred to as the trapezium rule.

$$I = \frac{h}{2}\left(f(a) + 2f(h) + 2f(2h) + 2f(3h) + \ldots + f(b)\right)$$

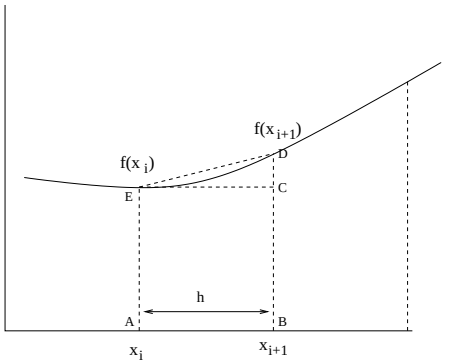


Figure 3.2: Approximation of an integral using the Trapezium Rule in the subinterval x_i to x_{i+1}

We will now use the Taylor series to represent the values of $f(x_i)$ and $f(x_{i+1})$ by in terms of $f(x_{i+1/2})$ and its derivatives via the Taylor series. This will give us an expression of 3.1.2 in the form of our exact expression of our integral 3.1.1.

$$\begin{aligned}f_i &= f_{i+1/2-1/2} = f_{i+1/2} + \frac{h}{2}f'_{i+1/2} + \frac{1}{2!}f''_{i+1/2} + \frac{1}{3!}f'''_{i+1/2} \\ f_{i+1} &= f_{i+1/2+1/2} = f_{i+1/2} - \frac{h}{2}f'_{i+1/2} + \frac{1}{2!}f''_{i+1/2} - \frac{1}{3!}f'''_{i+1/2} \qquad (3.1.3)\end{aligned}$$

So,

$$\begin{aligned}I(\text{estimated}) &= \frac{1}{2}h\left(f_i + f_{i+1}\right) \\ &= h[f_{i+1/2} + \frac{1}{2!}(h/2)^2 f''_{i+1/2} + \mathcal{O}(h^4)] \qquad (3.1.4)\end{aligned}$$

which we can compare the first few terms of the exact solution

$$I(\text{exact}) = hf_{i+1/2} + \frac{2}{3!}(\frac{h}{2})^3 f''_{i+1/2} + \mathcal{O}(h^5)$$

Thus the error $\Delta I = I(\text{estimated}) - I(\text{exact})$ is given by

$$\begin{aligned}\Delta I &= (\frac{1}{2\times 2!} - \frac{1}{2\times 3!})h^3 f''_{i+1/2} + \mathcal{O}(h^5) \\ &\approx \frac{1}{12}h^3 f''_{i+1/2} \qquad (3.1.5)\end{aligned}$$

The total error is just the sum of this error over all n sub intervals.

$$\begin{aligned}\Delta I &\approx \frac{1}{12}nh^3\langle f''\rangle \\ &= \frac{1}{12}(b-a)h^2\langle f''\rangle \end{aligned} \tag{3.1.6}$$

where $\langle f''\rangle$ is the average value of the second derivative of f over the interval a to b.

We can see the accuracy of the trapezium rule is proportional to the square of the interval. Thus if we half the size of the subinterval (i.e. double the number of subintervals) we increase the accuracy four fold.

Example:
Calculate the integral

$$\int_0^1 \cos(x)dx$$

with $h = 0.2$
With $h = 0.2$ we need to calculate $f(0.0)$,$f(0.2)$,$f(0.4)$,$f(0.6)$,$f(0.8)$ and $f(1.0)$. Doing this we get

$$\begin{aligned}\cos(0.0) &= 1.0 \\ \cos(0.2) &= 0.9807 \\ \cos(0.4) &= 0.92106 \\ \cos(0.6) &= 0.82534 \\ \cos(0.8) &= 0.69671 \\ \cos(1.0) &= 0.5403\end{aligned}$$

Using the Trapezium rule we have

$$\begin{aligned}\int_0^1 \cos(x)dx &= 0.2\left(\frac{1.0+0.5403}{2}+0.9807+0.92106+0.82534+0.69671\right) \\ &= 0.83867\end{aligned}$$

We can write the error as

$$\begin{aligned}\Delta I &\approx \frac{1}{12}(b-a)h^2\langle f''\rangle \\ &= \frac{0.04}{12}(-0.82735) \\ &= 0.0028\end{aligned} \tag{3.1.7}$$

So we have 0.83867 ± 0.0028, which we can compare to the exact solution of 0.84147.

3.1.2 Simpson's Rule

A better approximation to the function $f(x)$ in the region of x_i (where i is odd) is to replace the straight line by a quadratic of the form

$$f(x_i + y) = f_i + ay + b^2y \tag{3.1.8}$$

This was developed by Thomas Simpson (English) 1710-1761.

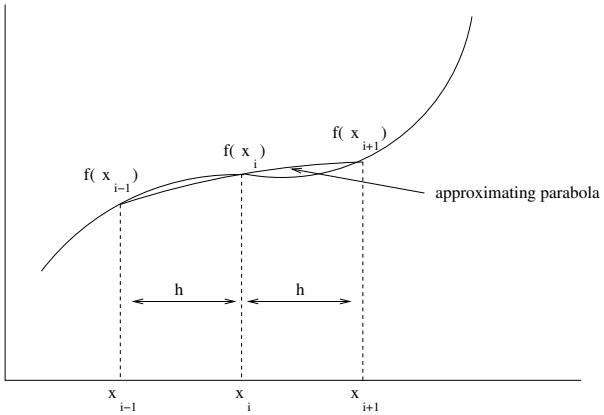


Figure 3.3: Simpson's rule approximates a function locally as a parabola across two subintervals. For this reason the total number of subintervals should be even.

The area of the double strip (see fig 3.3) from x_{i-1} to x_{i+1} is

$$I(\text{estimated}) = \int_{-h}^{h}(f_i + ay + by^2)dy = 2hf_i + 2bh^3/3 \tag{3.1.9}$$

We can apply $y = \pm h$ to 3.1.8 to obtain an expression for b.

$$\begin{aligned}f_{i+1} &= f(x_i + h) = f_i + ah + bh^2 \\ f_{i-1} &= f(x_i - h) = f_i - ah + bh^2\end{aligned}$$

giving us

$$bh^2 = \frac{1}{2}(f_{i+1} + f_{i-1} - 2f_i) \tag{3.1.10}$$

We can now eliminate b from 3.1.9 to get

$$\begin{aligned}I(\text{estimated}) &= 2hf_i + \frac{2}{3}(\frac{h}{2}(f_{i+1} + f_{i-1} - 2f_i)) \\ &= \frac{h}{3}(4f_i + f_{i+1} + f_{i-1})\end{aligned} \tag{3.1.11}$$

We note that we have set i as odd, so that in each odd subinterval we have $4f_i$ and one each of f_{i+1} and f_{i-1} giving us the approximation for the integral as

$$I(\text{estimated}) = \frac{h}{3}\left(f_0 + f_N + 4\sum_{n \text{ odd}} f_n + 2\sum_{m \text{ even}} f_m\right) \tag{3.1.12}$$

Exercise:
Use the procedure we used for the trapezium rule to show that the error associated with Simpson's rule is

$$\Delta I \approx \frac{(b-a)}{180}h^4\langle f^{(4)}\rangle \tag{3.1.13}$$

Example:
As for the Trapezium rule calculate the integral

$$\int_0^1 \cos(x)dx$$

with $h = 0.2$
As for the example for the Trapezium Rule we have

$$\begin{aligned}
\cos(0.0) &= 1.0\\
\cos(0.2) &= 0.9807\\
\cos(0.4) &= 0.92106\\
\cos(0.6) &= 0.82534\\
\cos(0.8) &= 0.69671\\
\cos(1.0) &= 0.5403
\end{aligned}$$

Using Simpson's rule we have

$$\begin{aligned}
\int_0^1 \cos(x)dx &= \frac{0.2}{3}(1.0 + 4(0.9807) + 2(0.92106) + 4(0.82534) + 2(0.69671) + 0.5403)\\
&= 0.84152
\end{aligned}$$

We can write the error as

$$\begin{aligned}
\Delta I &\approx \frac{1}{180}(b-a)h^4\langle f''\rangle\\
&= \frac{0.0012}{180}(-0.82735)\\
&= 7.35\times 10^{-6}
\end{aligned}$$

So we have $0.84152 \pm 7.35 \times 10^{-6}$, which we can compare to the exact solution of 0.84147.

3.1.3 Booles Rule

We can include higher order terms from the Taylors series to obtain higher order quadrature formulae. The generalisation of the Simpson rule using cubic and quartic polynomials to interpolate are given by:

Simpson's $\frac{3}{8}$ rule:

$$\int_{x_0}^{x_3} = \frac{3h}{8}\left[f_0 + 3f_1 + 3f_2 + f_3\right] + \mathcal{O}(h^5) \tag{3.1.14}$$

Booles rule:

$$\int_{x_0}^{x_4} = \frac{2h}{45}\left[7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4\right] + \mathcal{O}(h^5) \tag{3.1.15}$$

(This is often referred to as Bodes rule due to a typo in an early reference but is due to Boole (Boole & Moulton 1960))

A drawback of these methods though is that the number of points required by each method must be a multiple of the order of the method, for example Simpson's method is an order 2 (quadratic) method so we need a multiple of 2 grid points, Boole's method is a quartic ($n = 4$) method so requires a multiple of 4 grid points.

These methods are probably enough for most applications you will come across. However, a series of methods exist that vary the step size called Gaussian Quadrature. If $f(x)$ is slowly varying over the interval it may be advantageous to use Gaussian Quadrature .

3.2 Gaussian Quadrature

4

Monte Carlo Methods

The Monte Carlo method isn't actually a method, rather a family of methods involving large numbers of random samples, to find solutions to mathematical or physical problems using statistical techniques. In this chapter we will refer to such techniques as the Monte Carlo method. The method dates back to the Manhattan project in the 1940's and is named after the city of Monte Carlo because of its casinos (I suppose it sounds more romantic than the Las Vegas method!).

Given our description of the Monte Carlo method as a technique involving many random numbers, let us now describe briefly the major components of the method. These components are common to most Monte Carlo applications.

Random Number Generator — A source of random numbers uniformly distributed on the unit interval must be available.

Probability Distribution Functions (P.D.F.) — The physical (or mathematical) system must be described by a set of P.D.F..

Sampling Rule — A rule for sampling our P.D.F.. This usually involves the random number generator.

Scoring (sometimes referred to as Tallying) — The results of Monte Carlo samples must be gathered together into overall scores for the quantities we are interested in.

Error Estimation — No numerical method is complete without some sort of error estimate.

Variance Reduction Techniques — Techniques for reducing the variance (error) and reducing computational time.

It might be surprising to learn that we can use random numbers to do mathematical or physical calculations. The obvious question here is when would we use such a method? We could start

be asking how long does the Monte Carlo method take compared to a quadrature method (if we're talking about an integration problem).

We will look at error estimate in the Monte Carlo method later but if we let $\bar{t}_i$ be the computational time required to do one calculation then we can write the time taken using the Monte Carlo method, T_{MC}, to reach an error, ϵ_{MC}, as

$$T_{MC} = N\bar{t}_i = \left(\frac{\sigma}{\epsilon_{MC}}\right)^2 \bar{t}_i \tag{4.0.1}$$

where σ is the variance in one calculation.

We can write a corresponding time for a quadrature technique in n dimensions with a truncation error of the order of h^k. You will recall from the last chapter than the truncation errors for the trapezoidal rule and Simpson's rule were $k = 2$ and $k = 4$ respectively.

$$T_Q = N\bar{t}_i = \left(\frac{\sigma}{\epsilon_Q}\right)^{n/k} \bar{t}_i \tag{4.0.2}$$

A general rule of thumb is that you should use the Monte Carlo method if

$$\begin{aligned} T_Q &> T_{MC} \\ \frac{n}{k} &> 2 \end{aligned} \tag{4.0.3}$$

or

$$n > 6$$

4.1 Random Numbers

Creating a random number on a computer is difficult since computers use algorithms and 'do what they're told'. Random number generators have to be capable of generating a long list of random numbers, that are statistically acceptable, fast and reproducible. We will look at one method that is commonly used - 'Residue Arithmetic' i.e. operations on the remainder after division of two numbers (this operation is called the mod operator e.g. $5\text{mod}3 = 2$.

The general algorithm for creating random numbers in this method is

$$x_n = Cx_{n-1}(\text{mod}N) \tag{4.1.1}$$

It may be surprising that we can use a method as simple as this to generate random numbers. Lets look at a simple example with $C = 5$, $N = 32$ and $x_0 = 1$

$$\begin{aligned} x_1 &= 5x_0(\text{mod}32) = (5)(1) - (0)(32) = 5 \\ x_2 &= 5x_1(\text{mod}32) = (5)(5) - (0)(32) = 25 \\ x_3 &= 5x_2(\text{mod}32) = (5)(25) - (3)(32) = 29 \\ x_4 &= 5x_3(\text{mod}32) = (5)(29) - (4)(32) = 17 \end{aligned}$$

In fact the first 12 numbers of this sequence are

$$5, 25, 29, 17, 21, 9, 13, 1, 5, 25, 29, 17$$

which is not particularly useful since the sequence repeats itself every nine numbers, which we say has a period of 9. This is not at all useful if we require more than 9 random numbers! You can see a graph of a bad random number generator in fig 4.1. Choosing good constants for a random number generator is a difficult task, we seek numbers that are statistically random and have a large period. In 'C' the drand48() and lrand48() functions use a linear congruential formula

$$x_n = (ax_{n-1} + c) \text{mod } m \tag{4.1.2}$$

with $a = 2736731631558$, $c = 138$ and $m = 2^{48}$

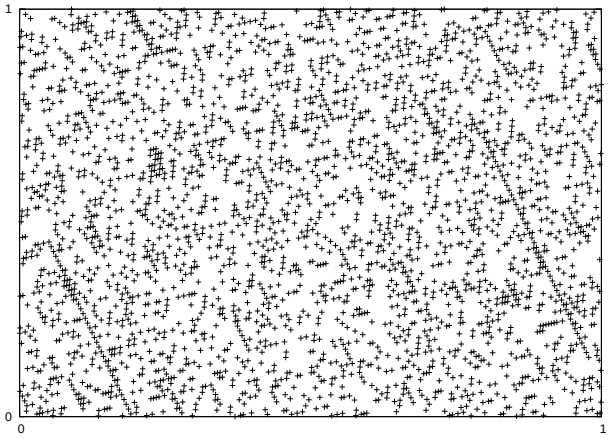


Figure 4.1: Bad choice of constants leads to correlations between successive 'random' numbers, these appear in the form of stripes in the distribution. Here $I_K = (aI_{K-1} + c) \text{mod } m$, where $a = 1277$, $c = 0$, $m = 131072$ and $I_0 = 1.0$

Why would you want a reproducible set of numbers? It is useful to be able to run a simulation with the same parameters to check for bugs or that a computational method is correct. To change the sequence all we need to do is to change the starting value, which we call a seed. In a program we can use the computer time (the number of seconds since 00:00:00, January 1, 1970) as a seed which guarantees a different seed every time we run our random number generator.

It is important to be able to trust the random number generator (RNG) we are using. We obviously want our RNG to give us random numbers with equal probability. This is know as the distribution, and we can check it quite easily by assigning each random number to a bin and counting the size of each bin. For example, suppose we are creating 1,000 random numbers in the range [0,1]. We can create 10 bins in this range $0.0 - 0.1, 0.1 - 0.2, \ldots 0.9 - 1.0$ and add one to each bin for every random number that falls inside that bin. Since the random number distribution is uniform we should expect each bin to contain approximately the same quantity of random numbers.

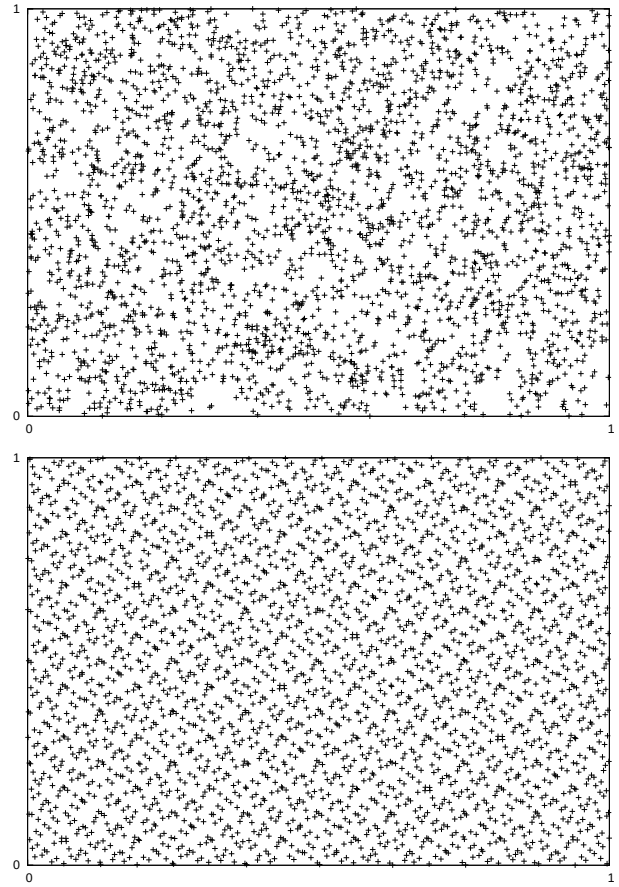


Figure 4.2: A pseudo random number generator is often too random for use in a Monte Carlo method. Random clusters and holes appear(above). Often a better choice is a quasi random number generator. Numbers are forced to have a uniform distribution (below).

4.2 Probability Distribution Functions

Let's start our discussion of the Monte Carlo technique with a deceptively simple example. Consider a physical problem described by N collisions distributed as follows

$0.2N$	occur in the range	$0.0 \leq r \leq 0.2$
$0.3N$	occur in the range	$0.2 \leq r \leq 0.5$
$0.5N$	occur in the range	$0.5 \leq r \leq 1.0$

, where r is some property of the system e.g. energy of a particle at a collision, so that our probabilities can be written as P_1 (the probability of collision A) = 0.2 and P_2 = 0.3 and $P_3 = 0.5$ so that P_1+P_2+$P_3 = 1.0$. We can now derive a simple program as follows

Choose a random number r in the range $[0, 1]$ ($[0, \sum P_i]$)

If $r - 0.2 < 0.0$ choose event A

else if $r - 0.5 < 0.0$ choose event B

else choose event C

We can formalise this by saying If E_1, E_2, E_3, ... E_n are independent, mutually exclusive events with probabilities P_1, P_2, P_3, ... P_n then a random number, r, such that

$$P_1 + P_2 + \ldots + P_{i-1} \leq r < P_1 + P_2 + \ldots + P_i$$

determines the event E_i

This is the Monte Carlo sampling principle for discrete events.

In a diagram we can think of this as a gating structure, imagine dividing up our interval into bins proportional to P_i. In figure 4.3 random number, r, falls into bin 2 so we choose event E_2.

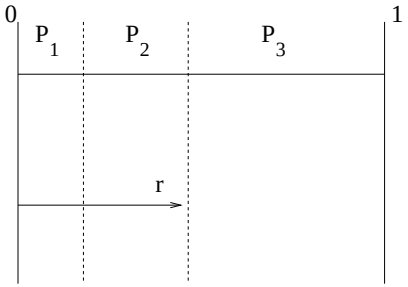
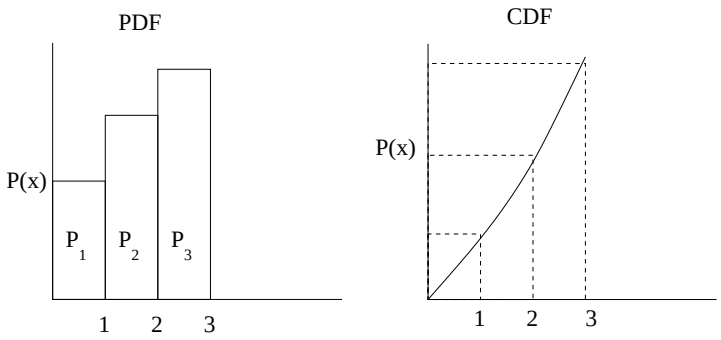


Figure 4.3: Gating structure for events E_1, E_2, E_3 with probabilities P_1, P_2, P_3

4.2.1 Continuous Probability Distributions

Suppose we want to choose a variable x from a continuous probability distribution. We will obtain a method via our original problem. For discrete events we have introduced the Probability Distribution Function (P.D.F.), which gives the probability of a random number being equal to a given value x. Since each event is assigned a value of x we can write the P.D.F. as

$$P(x) = P_i \, [i - 1 \leq x < i, i = 1, 2, 3, \ldots, n] \tag{4.2.1}$$



For a continuous probability distribution we introduce the Cumulative Distribution Function (C.D.F.), which gives the probability of a random number being less than a given value x. The C.D.F. is defined as

$$\mathcal{P}(x) = \int_0^x P(\eta) \mathrm{d}\eta \tag{4.2.2}$$

where $\mathcal{P}(\eta)$ is called the probability density and $\mathcal{P}(0) = 0$, $\mathcal{P}(n) = 1$.

So, if $P(x)\mathrm{d}x$ is the probability of x lying between x and $x + \mathrm{d}x$, where $a \leq x \leq b$, it should be obvious that

$$\int_a^b P(\eta) \mathrm{d}\eta = 1$$

Now we can map our computed value of the probability (our random number) to the physical variable by

$$p(r)dr = p(x)dx$$

i.e. the probability of finding a random variable, r, in r and $r + \mathrm{d}r$ is the same as the probability of finding the physical event , x, in x and $x + \mathrm{d}x$ Since $\int p(r)dr = 1$ then

$$r = \int_a^x P(x) \mathrm{d}x \tag{4.2.3}$$

which is our Monte Carlo principle for obtaining a variable x from a random number using a cumulative distribution function. We will look at how to sample from this distribution in the next section.

4.2.2 Sampling from P.D.F. and C.D.F.s

Consider a ball rolling freely backwards and forwards along a track between two fixed points a and b. Let the length of the track be $b - a$. The probability density, $P(x)$, is interpreted as the chance of finding the ball between x and $x + \mathrm{d}x$. How do we find the distribution function $P(x)$? Once we have this we can use our Monte Carlo rule.

We begin by saying that for free travel the ball has an equal chance of being at any x, i.e. that $P(x) = \text{constant} = A$ since the ball lies on the track. So how do we now calculate A? We know

that the ball must be somewhere in $[a, b]$ so

$$\int_a^b P(x)\mathrm{d}x = 1$$
$$A\int_a^b \mathrm{d}x = 1$$
$$=> P(x) = A = \frac{1}{b-a}$$

Now we can get random x values from a computer generated random number r from (4.2.3)

$$r = \int_a^x P(x)\mathrm{d}x = \int_a^x \frac{dx}{b-a} = \frac{x-a}{b-a}$$

or

$$x = a + (b-a)r$$

We call this a mapping of r onto x so that we can easily convert $0 < r < 1$ onto any x range. Suppose that $a = 0$ and $b = 2$, then $x = 2r$, or $a = -1$ and $b = 1$, in this case $x = 2r - 1$. These examples should be pbvious but this method works for more complicated problems.

Suppose we have a more complicated P.D.F.

$$P(\theta) = A\sin(\theta)$$

where $0 < \theta < \pi$.

We can now write

$$A\int_0^\pi \sin(\theta)\mathrm{d}\theta = 1$$
$$=> A = \frac{1}{2}$$

Thus we can write our PDF as

$$P(x) = \frac{1}{2}\sin(\theta)$$

and

$$r = \int_0^\theta \frac{1}{2}\sin(\theta)\mathrm{d}\theta = \frac{1}{2}\Big[-\cos(\theta)\Big]_0^\theta = \frac{1}{2}\Big[1-\cos(\theta)\Big]$$

So

$$\cos(\theta) = 1 - 2r$$

4.3 De Buffons Needle

This is a very simple Monte Carlo calculation that you can even try at home without a computer. The origins are uncertain but it attributed to Georges-Louis Leclerc Compte deBuffon (1707-1788). The game involves dropping a needle onto a set of parallel lines and counting the number of times that the needle crosses a line.

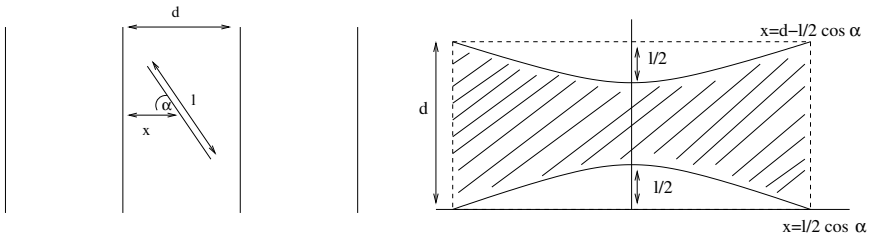
If we assume that the lines are separated by a distance d and the needle is a length l as in figure (4.4.a). Lets look at the condition for the needle **not** to cross a line.

$$x - \frac{l}{2}\cos(\alpha) > 0$$

and

$$x + \frac{l}{2}\cos(\alpha) < d$$

Check this from the diagram. We can make plot this condition on a graph (fig 4.4.b) where the probability of a miss is given by the shaded area.



a b. Conditions for a miss are shaded

Figure 4.4: DeBuffons Needle: The probability of a needle crossing a line is $2l/\pi d$

Let's call this probability $P_m(x)$ so that

$$P_m = \frac{1}{\pi d}\int\int dx d\alpha$$
$$P_m = \frac{1}{\pi d}\int_{\frac{-\pi}{2}}^{\frac{\pi}{2}} d\alpha \int_{\frac{l\cos(\alpha)}{2}}^{d-\frac{l\cos(\alpha)}{2}} dx$$
$$P_m = \frac{1}{\pi d}\int_{\frac{-\pi}{2}}^{\frac{\pi}{2}} (d - l\cos(\alpha))\, d\alpha$$
$$P_m = 1 - \frac{2l}{\pi d} \tag{4.3.1}$$

The probability of a hit $P_h = 1 - P_m = \left(\frac{2l}{\pi d}\right)$. If we make $d = 2l$ i.e. the separation be twice the length of the needle then we have a simple formula

$$P_h = \frac{1}{\pi} \tag{4.3.2}$$

This can be simulated by the Monte Carlo method. Assume that the probability distributions are uniform so that we can write $P(\alpha) = A$ and $P(x) = B$.

$$\int_0^{\frac{\pi}{2}} P(\alpha)d\alpha = \int_0^{\frac{\pi}{2}} Ad\alpha = 1$$

so $P(\alpha) = 1$ and

$$\int_0^1 P(x)dx = \int_0^1 Bdx = 1$$

giving $P(x) = 1$

We can now map our random numbers r_1 and r_2 as

$$\begin{aligned} r_1 &= \frac{2}{\pi}\int_0^\alpha d\alpha = \frac{2\alpha}{\pi} \\ r_2 &= \int_0^x P(x)dx = x \end{aligned} \tag{4.3.3}$$

We now have our random values of x and α as

$$\begin{aligned} x &= r_2 \\ \alpha &= \frac{\pi}{2} r_1 \end{aligned} \tag{4.3.4}$$

and a hit occurs whenever

$$x < \frac{1}{2}\cos(\alpha) \tag{4.3.5}$$

Write a computer program to do this. How many random numbers do you need to pick to get a reasonable value for π?

4.4 Integration

The most common application of the Monte Carlo method is in evaluating multidimensional integrals. We will look at a simple example of Monte Carlo integration and then apply the technique to more complicated integrals.

In our discussion of quadrature we evaluated the integral as

$$I = \int_a^b f(x)dx \equiv \sum_{i=1}^N \omega_i f(x_i) \tag{4.4.1}$$

where ω_i are weights determined by the quadrature rule. For Simpson's method we saw that $\omega_o = \omega_N = 1$, $\omega_{\text{even}} = 2$ and $\omega_{\text{odd}} = 4$.

In it's most basic form, integration using the Monte Carlo method uses the following ideas. Set all the weights to unity, $\omega_i = 1$. Since $dx = h = \frac{b-a}{N}$, where h is the step size we have

$$I = \int_a^b f(x)dx \equiv \frac{b-a}{N}\sum_{i=1}^N f(x_i) = (b-a)\langle f(x_i)\rangle \tag{4.4.2}$$

which is just the average value of the function in our interval.

This is the crudest method of integrating using the Monte Carlo method since we may choose several random numbers in a region where our function, $f(x)$, is zero or close to zero. This

uses computational time while not adding to our value of the integral. We will look at more efficient methods of calculating the integral in a later section, for now lets look at a piece of code to calculate the integral

$$I = \int_0^1 \cos(2x)^2 dx$$

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

double f(double x)
{
   return cos(2*x)*cos(2*x);
}

int main()
{
   int i, N=1000;
   double sum;

   fprintf(stdout, "Input N:\n");
   fscanf(stdin, "%d", &N);

   srand48( (unsigned int) time(NULL));

   sum = 0.0;
   for (i=0; i<N; i++)
   {
      sum = sum + f(drand48());
   }
   fprintf(stdout, "Int{cos(2x)^2} = %f\n", sum/N);

   exit(EXIT_SUCCESS);
}
```

For $N = 5 \times 10^7$ we get $I = 0.405446$, compared to the answer 0.405339968.

4.4.1 Double Integrals

The Monte Carlo method we have just looked at can easily be extended to double integrals and even integrals in higer dimensions, but we will only deal with double integrals here (I will leave it as an exercise to extend the method described to to higher dimensions).

In two dimensions we wish to compute the integral

$$\int_a^b \int_c^d f(x,y)dydx$$

Following the method outlined in the previous section, where we approximated a one-dimensional integral by an average of the function over a set of randomly chosen points in the interval we're integrating, we compute the average of $f(x, y)$ where x is sampled randomly from the rectangle $[a, b]$ and y is sampled randomly from the rectangle $[c, d]$. If we write the average of $f(x, y)$ as

$$\hat{f} = \frac{1}{N}\sum_{i=1}^{N} f(x_i, y_i)$$

The integral can be approximated as

$$\int_a^b \int_c^d f(x, y)dydx \approx (b - a)(d - c)\hat{f} \tag{4.4.3}$$

The error in this method is the standard Monte Carlo error,

$$\approx (b - a)(d - c)\sqrt{\frac{\hat{f}^2 - (\hat{f})^2}{N}} \tag{4.4.4}$$

where

$$\hat{f}^2 = \frac{1}{N}\sum_{i=1}^{N} f^2(x_i, y_i)$$

Exercise:
Calculate the integral

$$\int_0^{9/10} \int_0^1 \int_0^{11/10} (4 - x^2 - y^2 - z^2)dzdydx$$

4.4.2 Line and Surface Integrals

We can write the integral of a function over a surface as

$$I = \int_\Omega f(x, y)d\Omega$$

However, if we recognise that an integral over a surface can be thought of as a two-dimensional integral we can write our integral as

$$I = \int \int_\Omega f(x, y)dxdy$$

and then write an approximation to the integral as the average of the function times the area of the surface.

$$\hat{f} = \sum_{i=1}^{N} f(x, y)\Delta\Omega \tag{4.4.5}$$

The area $\Delta\Omega$ is given by Ω_0/N where Ω_0 is the area of the rectangle in fig 4.5. To evaluate (4.4.5) we simply generate N random points uniformly inside the rectangle in fig 4.5, if the point (x_i, y_i) lies within the surface then we add a the contribution $f(x_i, y_i)\Delta\Omega$ to the sum in (4.4.5), if the point lies outside the surface we add nothing.

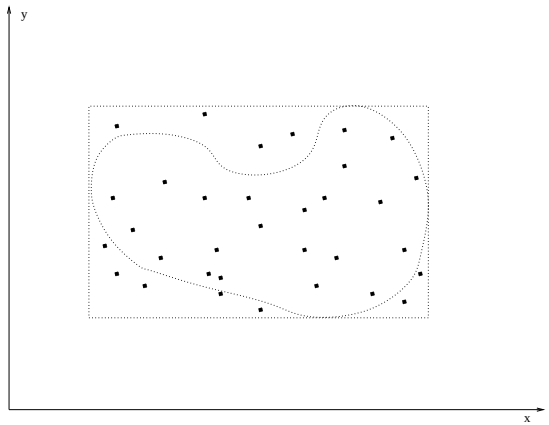


Figure 4.5: Evaluating the area inside a closed curve via the Monte Carlo method.

In the special case that $f(x, y) = 1$ then the sum in (4.4.5) will give the area of the surface. We will elucidate on this a little later. In higher dimensions, the plane surface we have discussed here will be replaced with an n dimensional surface and we would need to then to calculate the average of the function over n dimensions.

A line is then a simple (1-D) surface so the method above can be used to calculate the integral along a line.

Before we end this section on integration we should note the simplest method of calculating an integral using the Monte Carlo method, which is often seen in text books. We can think of an integral, such as 4.4.1, as the area between the x axis and the curve, $f(x)$. If we are sufficiently familiar with the function (we can plot it for example) then we can surround $f(x)$ in the interval by a sort of bounding box as in fig 4.6.

The integral is then interpreted as the fraction of the bounding box that $f(x)$ occupies times the area of this box. For this technique to work we have to find an appropriate bounding box, for example, if we wish to find the area of a circle with unit radius, centered on the origin, we could create a square of side 2, centered on the origin as in fig 4.7. We can now choose N random numbers in the range $x = -1..1$, $y = -1..1$ and count how many H that fall inside the circle i.e. $x^2 + y^2 <= 1$. The area of the circle is then given by

$$(2 \times 2)\left(\frac{H}{N}\right)$$

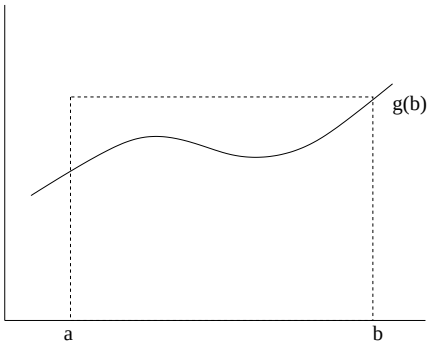


Figure 4.6: We can interpret the integral of $f(x)$ as the area between the x axis and the curve, $f(x)$. This allows us the pick random points in a suitably chosen bounding box and calculate the ratio of the areas, thus calculating the area under $f(x)$.

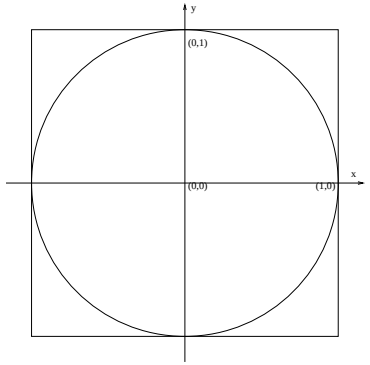


Figure 4.7: We can calculate the area of a circle using a simple Monte Carlo method. We surround the circle with a box, then calculate the fraction of random points chosen in this box that fall inside the circle. This gives us the ratio of the areas of the circle to the box and thus the area of the circle.

Example:
We can use this interpretation of Monte Carlo integration to calculate the integral of complex shapes. Here we will calculate the mass and the centre of mass of a hemisphere centered on the origin with a radius of 2. The equations for the mass and centre of mass are

$$\begin{aligned} M &= \int \rho dxdydz \\ CM_x &= \int \rho x\ dxdydz \\ CM_y &= \int \rho y\ dxdydz \\ CM_z &= \int \rho z\ dxdydz \end{aligned} \tag{4.4.6}$$

4.5 Errors in the Monte Carlo Method.

As well as the estimate of I the other important quantity we wish to evaluate using the Monte Carlo method is the variance σ^2 or the standard deviation σ. From statistical mechanics we can write the variance of the integral in 4.4.2 as

$$\begin{aligned} \sigma_f^2 &= \frac{1}{N}\sum_{i=1}^{N} f(x_i)^2 - \left(\frac{1}{N}\sum_{i=1}^{N} f(x_i)\right)^2 \\ &= \left(\langle f^2\rangle - \langle f\rangle^2\right) \end{aligned} \tag{4.5.1}$$

which is a measure of how much $f(x)$ deviates from $\langle f\rangle$ over the interval.

If we make several measurements of $f(x_i)$ to make an estimate of our integral I then for N measurements we have

$$\langle I\rangle = \frac{1}{N}\sum_{i=1}^{N}\langle f\rangle$$

and the variance is given by

$$\begin{aligned} \sigma_N^2 &= \frac{1}{N}\left[\langle\left(\frac{1}{N}\sum_{i=1}^{N} f(x_i)\right)^2\rangle - \left(\langle\frac{1}{N}\sum_{i=1}^{N} f(x_i)\rangle\right)^2\right] \\ &\approx \frac{1}{N}\left(\langle f^2\rangle - \langle f\rangle^2\right) \\ &= \frac{\sigma_f^2}{N} \end{aligned} \tag{4.5.2}$$

We can see that the error in the Monte Carlo method is

$$\approx \frac{1}{\sqrt{(N)}}$$

In the chapter on quadrature we saw that the error in the methods based on the Taylors series have an error that goes as $\mathcal{O}(h^k)$ where $k \geq 1$. Since h is defined as $(b-a)/N$ we can rewrite the error as $\approx N^{-k}$.

Let us now consider higher dimensions. Let us assume that we wish to integrate over a volume of length L and dimension n. We then have $(L/h)^n$ points and the error now goes as $N^{-k/n}$. However, since the Monte Carlo estimate we have developed here is a statistical method the error does not scale with dimension and thus is $N^{-1/2}$ So the Monte Carlo method is more efficient for

$$\frac{n}{k} > 2$$

as we have mentioned at the beginning of the chapter.

It should be mentioned that the error estimation developed here is for a sequence of pseudo-random numbers with no variance reduction techniques. These techniques, as you will appreciate from the name, reduce the variance thus are more efficient.

We have already mentioned that using random number produces a distribution that is actually too random for Monte Carlo techniques. The error associated with a quasi-random number generator (without derivation) goes as

$$\frac{\log(N)^n}{N}$$

4.6 Variance Reduction Techniques

There are several methods for reducing the variance of the Monte Carlo estimate of an integral. One of the most common and the one we will discuss here is called 'Importance Sampling' .

The idea is quite simple. You will recall that we have written our Monte Carlo estimate (without any variance reduction technique) as (4.4.2)

$$\int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(x_i)$$

We will now multiply and divide the integrand by a positive weight function $\omega(x)$, normalised so that

$$\int_a^b \omega(x)dx = 1$$

Our Integral is now written as

$$I = \int_a^b \omega(x)\frac{f(x)}{\omega(x)}dx \tag{4.6.1}$$

We will interpret this as a probbility distribution function multiplied by a function to be integrated ($f(x)/\omega(x)$). We can use our Monte Carlo rule for sampling from a PDF, $y(x)$

$$y(x) = \int_a^x \omega(x)dx \tag{4.6.2}$$

so that we get $\frac{dy(x)}{dx} = \omega(x)$; $y(x=a) = 0$; $y(x=b) = 1$.

Our Monte Carlo estimate for the integral can now be written as

$$I = \int_a^b \frac{f(y(x))}{\omega(y(x))}dx \equiv \frac{b-a}{N}\sum_i^N \frac{f(y(x))}{\omega(y(x))} \tag{4.6.3}$$

The Monte Carlo method we developed above for integrals is applied in exactly the same manner: average the values of $f(x)/\omega(x)$ for a random sample of points distributed according to the P.D.F $\omega(x)$.

If we choose a weight function that behaves approximately as $f(x)$ (i.e. is large when $f(x)$ is large and small when $f(x)$ is small) then we will sample more from those areas that add more to the estimate.

It should be obvious that if choose our weights, $\omega(x)$ to be 1 then the P.D.F would be uniform in $[a,b]$ and thus the estimate for the integral (4.6.3) simplifies to our non-variance-reduced form (4.4.2).

Also, if we were able to choose a weight function that exactly matched our integrand then the estimate for the integral becomes exact. The drawback is that we need to derive a P.D.F from the weight function (by integrating it), so if we can integrate the function to obtain a P.D.F there is no need to use the Monte Carlo method!

This leads us to a very important question. How do we choose the weight function? We know that we want $f(x)/\omega(x)$ close to unity, and that we need to be able to integrate $\omega(x)$. With these 2 conditions in mind you are free to choose your weights, although a bad choice could be worse than no variance reduction at all!

As an example, lets solve the integral

$$\int_0^1 \frac{dx}{1+x^2} = \frac{\pi}{4} = 0.78540 \tag{4.6.4}$$

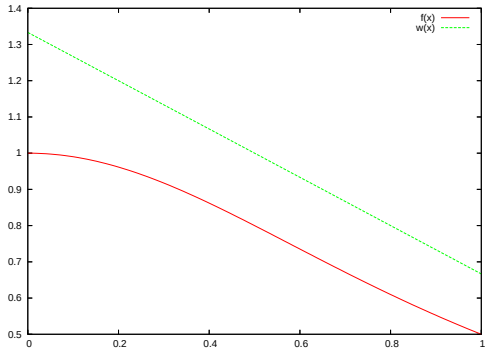


Figure 4.8: The function $f(x) = \frac{1}{1+x^2}$. We can choose a weight function $\omega(x)$ that approximates $f(x)$. We choose $\omega(x) = \frac{4-2x}{3}$ since it provides a good approximation to $f(x)$.

To choose a weight function we can plot the integrand, fig 4.8. From the figure we can see that a $\omega(x) = \frac{4-2x}{3}$ is a good choice. Note that

$$\int_0^1 \frac{4-2x}{3}dx = 1$$

and our P.D.F is

$$y(x) = \int_0^x \frac{4-2x}{3}dx = \frac{x(4-x)}{3}$$

from which we can obtain our random sample points

$$x^2 - 4x = -3y$$
$$(x-2)^2 - 4 = -3y$$
$$(x-2) = \sqrt{4-3y}$$
$$x = 2 - \sqrt{4-3y}$$

The following C code evaluates this integral using the above weight and using no variance reduction. The results for 1000 samples is 0.783605 (0.005191) for no variance reduction and 0.785880 (0.000631) using the above weight.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

double f(double x)
{
   return 1.0/(1+x*x);
}

double weight(double x)
{
   return (4-2*x)/3.0;
}

int main()
{
   int i, N=500;
   double fx;
   double sum=0.0, var=0.0;        /* sum and variance */
   double sum_vr=0.0, var_vr=0.0; /* sum and variance */
   double y_x;                     /* P.D.F. */

   fprintf(stdout, "Input N:\n");
   fscanf(stdin, "%d", &N);

   srand48( (unsigned int) time(NULL));

   for (i=0; i<N; i++)
   {
      y_x = drand48();
      sum = sum + f(y_x);
      var = var + f(y_x)*f(y_x);

      y_x = 2-sqrt(4-3*drand48());
      fx = f(y_x);
      sum_vr = sum_vr + f(y_x)/weight(y_x);
      var_vr = var_vr + (f(y_x)/weight(y_x))*(f(y_x)/weight(y_x));
   }
   fx = sum/N;
   fprintf(stdout, "With no Importance Sampling\n");
   fprintf(stdout, "Int{1/(1+x*x)} = %f (%f)\n",
                   fx, sqrt( (var/N - (fx*fx))/N) );

   fx = sum_vr/N;
   fprintf(stdout, "With Importance Sampling\n");
   fprintf(stdout, "Int{1/(1+x*x)} = %f (%f)\n",
                   sum_vr/N, sqrt( (var_vr/N - (fx*fx))/N));

   exit(EXIT_SUCCESS);
}
```


4.7 Random Walks

Random walks are used to model several physical processes, the typical example of which is Brownian motion. The idea is that a random walker (often referred to as a drunken sailor) can take a step in any direction he chooses, once he has made his choice he is free to step in another direction. Fig 4.9 shows a random walk on a 2D lattice, starting from an origin, there is a probability of 0.25 of moving to each of the neighbouring sites. For the second step there is also a probability of 0.25 of moving to each neighbouring sites and so on. Once a step has been taken, you will see that there is a probability (of 0.25) that the walker will step back onto the site that he has come from.

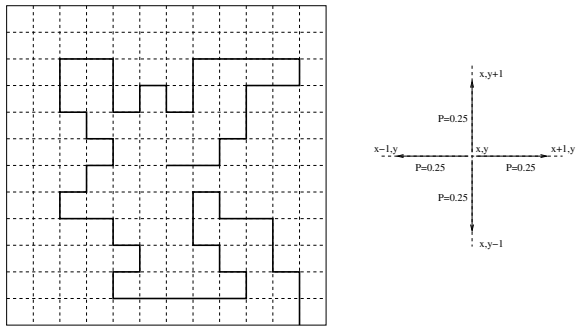


Figure 4.9: A typical random walk,left. At any point on an unbiased random walk the walker has a probability of 0.25 of moving in either of the 4 directions available.

Let a walker start at the origin and take N steps in the **x** and **y** directions of

$$(\delta x_1, \delta y_1), (\delta x_2, \delta y_2), (\delta x_3, \delta y_3), \ldots, (\delta x_N, \delta y_N)$$

We can measure the distance to the distance traveled from the starting point

$$\begin{aligned} R^2 &= (\Delta x_1 + \Delta x_2 + \cdots + \Delta x_N)^2 + (\Delta y_1 + \Delta y_2 + \cdots + \Delta y_N)^2 \\ &= \Delta x_1^2 \Delta x_2^2 + \cdots + \Delta x_N^2 + 2\Delta x_1 \Delta x_2 + 2\Delta x_1 \Delta x_3 + \ldots \end{aligned} \tag{4.7.1}$$

This expression is quite general and is true for any walk. However, if our walk is random then there is an equal probability of moving backwards as forwards, to the right as to the left so the cross terms will cancel leaving

$$\begin{aligned} R^2 &\approx \Delta x_1^2 + \Delta x_2^2 + \cdots + \Delta x_N^2 + \Delta y_1^2 + \Delta y_2^2 + \cdots + \Delta y_N^2 \\ &= N\langle r^2 \rangle \\ R &= \sqrt{N} r_{\text{rms}} \end{aligned} \tag{4.7.2}$$

This applies also to a random walk in 3 dimensions. This means that the distance traveled from our starting position is proportional to the number of steps taken, the longer we walk for the further we will be from where we started. We will use this fact in the next section.

Exercise:
Write a computer program that implements the random walk in two and three dimensions. After each step calculate the distance R from the origin and record it with the number of steps taken.
Plot R against $\sqrt{N}$ for 2D and 3D random walks.

4.8 Differential Equations

We will now turn our attention to solving partial differential equations (P.D.E.s) using the Monte Carlo method. We will take as an example, Laplace's equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{4.8.1}$$

You will recall that, in our discussion of finite differences from the Taylor series, that we can write

$$\frac{\partial^2 T}{\partial x^2} \equiv \frac{T_{x+1} - 2T_x + T_{x-1}}{h^2} \tag{4.8.2}$$

We can thus finite difference Laplace's equation by using 4.8.2 in 4.8.1 to get

$$T_{x,y} = \frac{T_{x+1,y}}{4} + \frac{T_{x-1,y}}{4} + \frac{T_{x,y+1}}{4} + \frac{T_{x,y-1}}{4} \tag{4.8.3}$$

To solve this using the Monte Carlo method we should first take a step back and think a little about what the above formula is saying. Suppose we have a very small grid that has one internal point which we call $T_{x,y}$ and the boundaries of the grid have values. Then 4.8.3 says

that the value of $T_{x,y}$ the sum of the nearest neighbours on the grid (i.e the boundaries) divided by 4. Suppose we have a more complicated grid, like that in fig 4.9 with $T_{x,y}$ somewhere near the centre of the grid. We calculate that $T_{x,y}$ is again a quarter of the sum of the nearest neighbours. We could apply this formula iteratively to each grid point in the lattice and work out the coefficients of neighbour. If we did this then we would find that each grid point would have the same coefficients as the probability of our random walk!

We can thus do a random walk from $T_{x,y}$ until we reach a boundary. We know we will always reach a boundary since our investigation of the random walk found that the distance traveled is proportional the square root of the number of steps taken. Thus if we do several random walks from $T_{x,y}$ and calculate the average of the value at the boundary we reach then we have a value for 4.8.1 at $T_{x,y}$.

Even without doing any analysis on this method (which is beyond the scope of this course) you will notice how inefficient this method is.

Exercise:
The following piece of code implements the above Monte Carlo technique for Laplaces equation on a 10x10 site grid. The temperatures to the left and right of the grid are 0°C, the top is 80°C and the bottom boundary 45°C.
Create a Monte Carlo Technique to solve

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \cos(x)\sin(y)$$

and modify the code below to solve the equation.
Plot your result.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define L 10

int main()
{
   int i,j;              /* grid points */
   int x,y,walks;        /* position and counter for random walks */
   int WALKS=1000;       /* the number of walks per grid point */
   double dir;           /* direction of random walk */
   double grid[L][L];
   double temperature;
   double Left_T=0.0;    /* temperature at the left of the grid */
   double Right_T=0.0;   /* temperature at the right of the grid */
   double Top_T=80.0;    /* temperature at the top of the grid */
   double Bottom_T=45.0;/* temperature at the bottom of the grid */

   /*** Set up the RNG ***/
```

```
srand48( (unsigned int) time(NULL));

/*** initialise the lattice ***/
for (i=0; i<L-1; i++)
   for (j=0; j<L-1; j++)
      grid[i][j] = 0.0;


/* do the Monte Carlo differentiation at each point
 * NOT along the edges, here we have already applied the
 * boundary conditions */
for (i=0; i<L; i++)
  for (j=0; j<L; j++)
  {
      walks = 0;
      temperature=0.0;
      while (walks<WALKS)
      {
         x=i;
         y=j;
         /*** do random walk ***/
         do
         {
            dir = drand48();
            if (dir<0.25) x=x-1;
            else if (dir<0.5) x=x+1;
            else if (dir<0.75) y=y-1;
            else  y=y+1;

         } while ( (x>=0) && (x<L) && (y>=0) && (y<L));
         if (x<0) temperature += Left_T;
         if (x>=L) temperature += Right_T;
         if (y<0) temperature += Bottom_T;
         if (y>=L) temperature += Top_T;
         walks++;
      }
      grid[i][j]=temperature/walks;
   }


/*** Output the lattice ***/
for (i=0; i<L; i++)
{
   for (j=0; j<L; j++)
      fprintf(stdout, "%5.3f   ", grid[i][j]);
   fprintf(stdout, "\n");
}
```



```
   return EXIT_SUCCESS;
}
```

5

Ordinary Differential Equations

Many of the laws of physics are most conveniently written in the form of differential equations. It should be no surprise therefore that the numerical solutions to differential equations are probably the most common applications of numerical methods to physical systems. In this chapter we will discuss methods of solving ordinary differential equations (O.D.E.s), the most general form of which is written as

$$\frac{dy}{dt} = f(y,t) \tag{5.0.1}$$

We can easily solve higher order O.D.E.s by using an auxiliary function. For example, you will recognise Newton's second law

$$F(r) = m\frac{d^2r}{dt^2} \tag{5.0.2}$$

We can define the momentum

$$p(t) = m\frac{dr}{dt}$$

So that 5.0.2 can be written as two coupled first order O.D.E.s

$$\frac{dr}{dt} = \frac{p}{m} \;\; ; \;\; \frac{dp}{dt} = F(r) \tag{5.0.3}$$

We will look at schemes to solve these coupled O.D.E.s a little later in the chapter.

Before we begin discussing methods to solve O.D.E.s, let us look at the general approach we will use. For a typical O.D.E. (5.0.1) we wish to solve for the value of y at the position x at every position between any two given boundaries. This defines a trajectory similar to that shown in fig 5.1 and every method we develop here will involve the same basic technique, knowing the value of y at a given time , $y(t)$ we will calculate the next point on the trajectory $y(t+\Delta t)$ using some numerical method. If we are given an initial value for y, say at time $t = 0$, we will

divide the interval into N sub-intervals, each of width h, and calculate the value of $y(t+nh)$ until we reach $y(t+Nh)$.

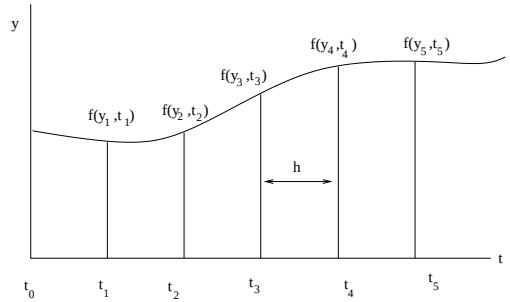


Figure 5.1: Evolution of the O.D.E. 5.0.1

The first serious attempt at solving these equations was made by Leonard Euler , a Swiss mathematician. We will begin our analysis of O.D.E.s with his method.

5.1 Eulers Method

Today the Euler method looks rather unsophisticated but it will provide us with a nice introduction into more accurate methods.

Firstly, let us use the first two terms of the Taylor expansion of $y(t)$

$$y(t+h) = y(t) + hy'(t) + \mathcal{O}(h^2)$$

We will introduce the following notation (that is common to most textbooks on the subject).

$$\begin{aligned} y(t_0) &= y_0 \\ f(y_0) &= f_0 \\ y(t) &= y(t_0+nh) = y_n \\ f(y_n) &= f_n \\ y(t+h) &= y(t_0+(n+1)h) = y_{n+1} \\ f(y_{n+1}) &= f_{n+1} \\ t+nh &= tn \end{aligned}$$

Using this notation we can rewrite the Taylor expansion as

$$y_{n+1} = y_n + hf_n + \mathcal{O}(h^2) \tag{5.1.1}$$

where $\mathcal{O}(h^2)$ is the order of the truncation error. This is the Euler method for solving ordinary differential equations.

The Euler method proceeds as follows, starting at y_0, calculate $y_1 = y_0 + hf(y_0)$. Now knowing y_1 calculate $y_2 = y_1 + hf(y_1)$ and continue until we reach y_n.

The term $\mathcal{O}(h^2)$ is the local truncation error , that is the error associated with truncating the Taylor expansion in going from y_n to y_{n+1}. In going from y_0 to y_N we have a global truncation error. The goal of numerical methods is to create a method that has a global error that is as low as possible while requiring few steps (the fewer the steps the less the computation required!). We can define the global error as

$$\begin{aligned} \text{global error} &= N\mathcal{O}(h^2) \\ &= \frac{L \times \mathcal{O}(h^2)}{h} \\ &= L \times \mathcal{O}(h) \end{aligned}$$

Here, we set L to be the length of the interval ($L = Nh$). We can see that the Euler method has a local error of $\mathcal{O}(h^2)$ but a global error of $\mathcal{O}(h)$. This means that if we want to double our accuracy we would need to double the number of steps we take. As a numerical method for solving O.D.E.s this is quite poor. We will see that with very little extra effort we can get a lot more accurate results.

5.2 Improved Eulers Method

If we integrate our typical O.D.E. (5.0.1) in the range $y = n..(n+1)$ we can rewrite it as

$$\int_{y_n}^{y_{n+1}} \frac{y(t)}{dt} dt = \int_{y_n}^{y_{n+1}} f(y,t) dt$$

or with a bit of integrating and rearranging

$$y_{n+1} = y_n + \int_{y_n}^{y_{n+1}} f(y,t) dt \tag{5.2.1}$$

So our methods for solving O.D.E.s becomes an integration problem. If you look at the Euler method for evolving an O.D.E. one step you should recognise the equation, it is the same approximation we used to calculate an integral using Riemann sums. This is shown in fig 5.2. Is it possible to use a more accurate integration method to evolve the O.D.E? The answer is yes. Lets try and apply the trapezium rule to our O.D.E.

We have, for one step,

$$\int_{y_n}^{y_{n+1}} f(y,t) dt = \frac{h}{2} \left(f(y_{n+1}) + f(y_n) \right)$$

which we can solve and rearrange to give

$$y_{n+1} = y_n + \frac{h}{2} \left(f(y_{n+1}) + f(y_n) \right) \tag{5.2.2}$$

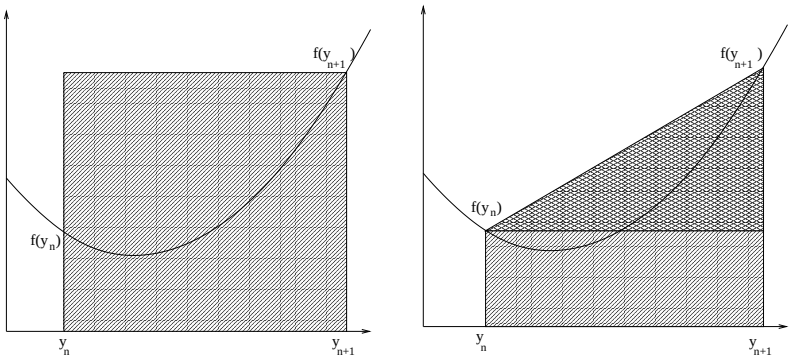


Figure 5.2: The Euler method calculates the solution to the ode at a point by the simplest numerical integrator (left) while the improved Euler method uses the trapezium rule (right) to evolve the solution.

The trouble with this approximation is that the value we are trying to estimate (y_{n+1}) appears on both sides of the equation. We will replace the y_{n+1} on the right hand side by an Euler approximation.

$$y_{n+1} = y_n + \frac{h}{2} \left(f(y_n) + f(y_n + hf(y_n)) \right) \tag{5.2.3}$$

This iteration formula is known as the Improved Euler method (or often called Heun's method).We actually compute it in three steps:

$$m_1 = f(y_n)$$

$$m_2 = f(y_n + hm_1)$$

$$y_{n+1} = y_n + h(m_1 + m_2)/2$$

Example:
Lets compare the Euler and Improved Euler methods in an example. Take the function $\frac{dy}{dx} = -xy$, and $y_0 = 1.0$. The exact solution is $exp(-x^2/2)$
The code below implements both methods we have looked at so far, the output is printed in that following table.

x	Euler	Improved Euler	Exact
0.000000	1.000000	1.000000	1.000000
0.100000	1.000000	0.995000	0.995012
0.200000	0.990000	0.980174	0.980199
0.300000	0.970200	0.955964	0.955997
0.400000	0.941094	0.923079	0.923116
0.500000	0.903450	0.882463	0.882497
0.600000	0.858278	0.835252	0.835270
0.700000	0.806781	0.782714	0.782705
0.800000	0.750306	0.726202	0.726149
0.900000	0.690282	0.667089	0.666977
1.000000	0.628157	0.606718	0.606531

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double f(double y, double x)
{
   return -x*y;
}

int
main ()
{
   double x=0.0, h=0.1;
   double c1,c2;
   double y_euler=1.0;
   double y_impr=1.0;

   fprintf(stdout, "x\t\tEuler\t\tImproved Euler\tExact\n");
   while (x<=1.0)
   {
      /* Print the results (including the exact solution)*/
      fprintf(stdout, "%f\t%f\t%f\t%f\n",
                      x,y_euler, y_impr, exp(-0.5*x*x) );

      /* implement the Euler method*/
      y_euler = y_euler + h*f(y_euler,x);

      /* implement the Improved Euler method*/
      c1 = f(y_impr,x);
      c2 = f(y_impr+h*c1, x+h);
      y_impr = y_impr+ (h*0.5)*(c1+c2);
```



```
      x += h;
   }
   exit(EXIT_SUCCESS);
}
```

Unlike the Euler method, we do not know all the values on the right hand side of our O.D.E. solver, we call methods of this type '*implicit*' methods while the Euler method is an example of an '*explicit*' method.

The Improved Euler (or Heun) method belongs to a class of methods called "Predictor-Corrector Methods" since we 'predict' a value of y_{n+1} (here we use the Euler method) and then 'correct it in the last step. There are several of these methods and all use a simple (less accurate) prediction method and then a higher accuracy correction step.

Other examples of explicit method are

Adams-Bashford two step method:

$$y_{n+1} = y_n + h\left(\frac{3}{2}f(y_n) - \frac{1}{2}f(y_{n-1})\right) + \mathcal{O}(h^3)$$

Adams-Bashford four step method:

$$y_{n+1} = y_n + \frac{h}{24}\left(55f(y_n) - 59f(y_{n-1}) + 37f(y_{n-2}) - 9f(y_{n-3})\right) + \mathcal{O}(h^5)$$

An example of an implicit multi-step method is

Adams-Moulton three step method:

$$y_{n+1} = y_n + \frac{h}{24}\left(9f(y_{n+1}) + 19f(y_n) - 5f(y_{n-1}) + f(y_{n-2})\right) + \mathcal{O}(h^5)$$

A common example of a predictor-corrector method is to use the Adams-Bashford 4-step method to make a prediction for y_{n+1} and the Adams-Moulton 3-step formula to correct the prediction. The advantage of these methods is that they allow for a continuous monitoring of the error by checking that the correction is small.

5.3 Runge Kutta Methods

Numerical methods for solving O.D.E.s are categorised according to their global error. The Euler method has a global error of $\mathcal{O}(h)$ and is classified as a first order method, the improved Euler method is classified as a second order method, having a global error of $\mathcal{O}(h^2)$.

So far in this chapter we have used a Taylor expansion of $y(t)$ to get an approximation to the integral in (5.2.2).

$$y_{n+1} = y_n + \frac{h}{2}\left(f(y_{n+1}) + f(y_n)\right)$$

We have seen that the Euler method uses the value at the beginning of a subinterval to solve for the value at the end of the subinterval and that the more accurate improved Euler method uses values at either end of the subinterval to solve the value at the end of the subinterval. In our chapter on quadrature we learned about Simpsons rule which used the value of the function across two intervals. In our discussion of O.D.E.s this would amount to taking the value at either end as in the Improved Euler method and at the midpoint of the interval.

We can use Simpsons method for our O.D.E. to get

$$y_{n+1} = y_n + \frac{h}{6}\left(f(y_n, t_n) + 4f(y_{n+1/2}, t_{n+1/2}) + f(y_{n+1}, t_{n+1})+\right) + \mathcal{O}(h^5) \tag{5.3.1}$$

which is known as the 2^{nd} order Runge Kutta method.

A third order Runge Kutta method is possible that requires 3 evaluations of $f(y,t)$ per step.

It is found by experience that a fourth order algorithm, which requires 4 evaluations of the function per step, gives the best balance between accuracy and computational effort. This fourth order Runge Kutta method is written as.

$$\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + \frac{1}{2}k_1, t_n + \frac{1}{2}h) \\ k_3 &= hf(y_n + \frac{1}{2}k_2, t_n + \frac{1}{2}h) \\ k_4 &= hf(y_n + k_3, t_n + h) \end{aligned}$$

$$y_{n+1} = y_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k4\right) + \mathcal{O}(h^5) \tag{5.3.2}$$

Higher order algorithms are also possible that give even more accurate results, however, these require more and more evaluations of the function per step. The 4^{nd} order Runge Kutta method gives the best balance between speed and accuracy.

Example:
Write a computer program to solve

$$\frac{dy}{dx} = x^2 + 0.11y^2 + 1$$

using the fourth order Runge Kutta method.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double f(double x, double y)
{
   return (x*x) + (0.11*y*y) + 1.0;
}

int
main ()
{
   double x=0.0, h=0.1;
   double y = 1.0;
   double k1, k2, k3, k4;

   fprintf(stdout, "x\t\tRunge Kutta\n");
   fprintf(stdout, "%f\t%f\n", x,y);
   while (x<=1.0)
   {
      k1 = f(x,  y);
      k2 = f(x+(h/2.0),  y+(k1/2.0));
      k3 = f(x+(h/2.0),  y+(k2/2.0));
      k4 = f(x+h,  y+k3);

      y = y + h*(k1 + (2.0*k2) + (2.0*k3) + k4)/6.0;
      x += h;

      fprintf(stdout, "%f\t%f\n", x,y);
   }
   exit(EXIT_SUCCESS);
}
```

5.4 Coupled O.D.E.s

The 4^{th} order Runge Kutta method outlined above is easy to follow in the simple case we have looked at. How do we solve differential equations that are coupled such as the Lotka-Volterra model?

First of all, I will explain what the Lotka-Volterra model is so that we describe a procedure to solve it. The Lotka-Volterra model is the simplest model of predator-prey interactions, it is normally described in terms of populations of foxes and rabbits. If we have a population of rabbits then they will reproduce as long as there is food available (we will assume they eat grass and it is always available). However, we also have a population of foxes that eat the rabbits, these will reproduce as long as there are rabbits to eat. If the population of foxes increase to a point where they eat all the rabbits (rabbit population $\rightarrow 0$) the foxes will die out (fox population $\rightarrow 0$).

Lets put this a bit more mathematically. We represent the number of rabbits in a given area as R, (for this discussion we will assume that the foxes and rabbits live on an island and so we are really talking about a density of rabbits) and so dR/dt will be the rate of change in the rabbit population and we will similarly define F and dF/dt for foxes. If the rate of reproduction of the rabbits is r then rR will be the rate of increase of rabbits, if m is the mortality rate of foxes then mF is the rate at which the foxes are dying. The final terms we require are the rate at which rabbits are eaten and the rate at which foxes reproduce. We will let b be the rate at which foxes reproduce per 1 rabbit eaten, so that bFR is the rate of increase of foxes and a be the predation rate so that aFR is the rate foxes eat rabbits.

We now have all the terms to write down our model.

$$\begin{aligned}\frac{dR}{dt} &= rR - aRF \\ \frac{dF}{dt} &= bRF - mF\end{aligned}$$

According to the Runge-Kutta we have already discussed we simply calculate $k_1,\ k_2,\ k_3,\ k_4$ for R and F. To keep our notation easy lets call the coefficients for the rabbits $k_1,\ k_2,\ k_3,\ k_4$ and for the foxes $l_1,\ l_2,\ l_3,\ l_4$. We can write down the expressions for k_1 and l_1

$$\begin{aligned}k_1 &= f(R,F,t) \\ l_1 &= g(R,F,t)\end{aligned}$$

where we have called $f(R,F,t) = rR - aRF$ and $g(R,F,t) = bRF - mF$.

Looking back at our previous example of the Runge Kutta method we had $dy/dt = f(y,t)$ and we saw that $k_1 = f(y,t)$ and to calculate k_2 we replaced y with $y + (k_1/2)$ and t with $t + h/2$. If we look closely at the formula it says *"to the dependant variable, add half the previous coefficient, and to the independant variable add h/2"*. In our case we need to replace R with $R + k_1/2$ and F with $F + l_1/2$, leaving

$$\begin{aligned}k_2 &= f(R + k_1/2,\ F + l_1/2,\ t + h/2) \\ l_2 &= g(R + k_1/2,\ F + l_1/2,\ t + h/2)\end{aligned}$$

We can repeat this for the other terms to get a prescription for applying the 4^{th} order Runge

Kutta method to coupled O.D.E.s

$$\begin{aligned}k_1 &= f(R,\ F,\ t) \\ l_1 &= g(R,\ F,\ t) \\ k_2 &= f(R + k_1/2,\ F + l_1/2,\ t + h/2) \\ l_2 &= g(R + k_1/2,\ F + l_1/2,\ t + h/2) \\ k_3 &= f(R + k_2/2,\ F + l_2/2,\ t + h/2) \\ k_3 &= f(R + k_2/2,\ F + l_2/2,\ t + h/2) \\ l_4 &= g(R + k_3,\ F + l_3,\ t + h) \\ l_4 &= g(R + k_3,\ F + l_3,\ t + h)\end{aligned}$$

$$\begin{aligned}R_{n+1} &= R_n + \frac{1}{6}\big(k_1 + 2k_2 + 2k_3 + k_4\big) \\ F_{n+1} &= F_n + \frac{1}{6}\big(l_1 + 2l_2 + 2l_3 + l_4\big)\end{aligned} \tag{5.4.1}$$

So once we know the population of rabbits and foxes at an initial time we can use the Runge Kutta to calculate the populations evolution in time. In our functions $f(R,F,t)$ and $g(R,F,t)$ you will notice that we have no t term so we can drop the $t,\ t + h/2,\ t + h$ terms in 5.4.1.

It should be obvious that the procedure outlined here can easily be extended to any number of coupled O.D.E.s.

5.5 Higher order O.D.E.s

Our application of the Runge Kutta method to coupled O.D.E.s is especially useful when we look at higher order O.D.E.s. As an example we'll look at the harmonic oscillator with resistance.

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = 0 \tag{5.5.1}$$

You should be familiar with this equation and recognise the first term as the kinetic energy term, the second as the resistance force and the final term as the restoring force, we will ignore the values of coefficients here since we are only interesting in the method to solve the equation.

Any 2^{nd} order O.D.E. like 5.5.1 can be split into 2 coupled first order O.D.E.s. In our example here we will set $dx/dt = v$. As a physicist you will recognise that we have written velocity equals the rate of change of displacement, but we only require a variable to represent dx/dt and have chosen v for simplicity, we were free to chose any variable we wished.

If we write 5.5.1 in terms of v we have

$$\begin{aligned}\frac{dx}{dt} &= v \\ \frac{dv}{dt} &= \frac{-cv}{m} - \frac{kx}{m}\end{aligned} \tag{5.5.2}$$

This set of coupled differential equations can be solved using the method outlines in the last section.

5.6 Numerovs Method

If the 2nd order O.D.E. to be solved contains no first order term (e.g. the Schröedinger equation, or the one dimensional Poisson equation) we can use the Taylor series to derive a 'simple' method referred to as Numerov's method .

The Numerov method is a particularly simple and efficient method for solving equations of the form

$$\frac{d^2y}{dx^2} + f(x)y = g(x) \tag{5.6.1}$$

By taking the Taylor series of $d(y+h)/dx = f(y+h)$ and $d(y-h)/dy = f(y-h)$ to the fourth power

$$f(y+h) \approx f(y) + hf'(y) + \frac{h^2}{2}f''(y) + \frac{h^3}{6}f'''(y) + \frac{h^4}{24}f''''(y)$$
$$f(y-h) \approx f(y) - hf'(y) + \frac{h^2}{2}f''(y) - \frac{h^3}{6}f'''(y) + \frac{h^4}{24}f''''(y)$$

and adding them we get

$$f(y+h) + f(y-h) - 2f(y) = h^2f''(y) + \frac{h^4}{12}f''''(y)$$

A little rearranging gives

$$\frac{f(y+h) + f(y-h) - 2f(y)}{h^2} = f''(y) + \frac{h^2}{12}f''''(y)$$

From now on we will adopt the usual notation $f(y+h) = y_{n+1}$, $f(y) = y_n$ and $f(y-h) = y_{n-1}$ and rewrite the last equation as

$$\frac{y_{n+1} - 2y_n + y_{n-1}}{h^2} = y_n'' + \frac{h^2}{12}y_n'''' \tag{5.6.2}$$

Let's take a minute to reflect on our progress. We have derive a three point approximation to a second derivative with no first derivative term but with a fourth derivative term. We can differentiate 5.6.1 twice to get an expression for y_n''''

$$y_n'''' = \frac{d^2}{dx^2}\Big(g(x) - f(x)y\Big) \tag{5.6.3}$$

We have a centered difference approximation for the second derivative.

$$f_n'' = \frac{f_{n+1} - 2f_n + f_{n-1}}{h^2}$$

which we can apply to 5.6.3.

$$y_n'''' = \frac{g(x)_{n+1} - 2g(x)_n + g(x)_{n-1}}{h^2} - \frac{\big(f(x)y\big)_{n+1} - 2\big(f(x)y\big)_n + \big(f(x)y\big)_{n-1}}{h^2}$$

We now have an expression for the fourth order term and the second order term ($y'' = g_n - f_ny_n$) which we can insert into 5.6.2 to get the Numerov method.

$$\Big(1 + \frac{h^2}{12}f_{n+1}\Big)y_{n+1} = \Big(2 + \frac{5}{6}h^2f_n\Big)y_n - \Big(1 + \frac{h^2}{12}f_{n-1}\Big)y_{n-1} - \frac{h^2}{12}\Big(g_{n-1} - 10g_n + g_{n-1}\Big) \tag{5.6.4}$$

The Numerov method has a local error of $\mathcal{O}(h^6)$ with only one calculation of $f()$ and $g()$ per step. Compare this to the 4th order Runge Kutta method which requires the evaluation of 6 functions in each step to obain the same local accuracy.

The Numerov method can be solved backwards or forwards with a local error of $\mathcal{O}(h^6)$ which is one order more accurate than the Runge Kutta method which could be used to integrate the problem as two coupled first order O.D.E.s.

The problem with the Numerov method, whether we integrate forwards or backwards, is that we need two starting values either y_n and y_{n-1} to forward integrate or y_n and y_{n+1} to backward integrate, where we usually only have one initial value $y_{n=0}$. To calculate y_2 with an accuracy of $\mathcal{O}(h^6)$ we need a value of y_1 with at least that accuracy, however, since the global error of the method is $\mathcal{O}(h^5)$ and we calculate y_1 only once we can use a method that is $\mathcal{O}(h^5)$ accurate to calculate y_1 and still retain our global accuracy.

One way of starting the Numerov method is suggected in *Computers in Physics, Vol 11, no. 5 1997 514-515.* The authors suggest

$$y_1 = \frac{y_0\Big(1 + \frac{f_2h^2}{24}\Big) + hy_0'\Big(1 + \frac{f_2h^2}{24}\Big) + \frac{h^2}{24}\Big(7g_0 - 7f_0y_0 + 6g_1 - g_2\Big) + \frac{h^4f_2}{36}\Big(g_0 - f_0y_0 + 2g_1\Big)}{1 + \frac{f_1h^2}{4} + \frac{f_1f_2h^4}{18}} \tag{5.6.5}$$

which preserves the global $\mathcal{O}(h^5)$ accuracy.

5.7 Finite Differences

We have already seen numerous examples of finite differences in writing approximations to terms involving differentiation of a variable. We will finish our discussion of O.D.E.s with another application of these finite differences, this time we will use as an example the evolution of a set of coordinates. The derivatives of the position of a body,$r(t)$ are written

$$\frac{d\mathbf{r}}{dt} = \mathbf{v}(t)$$
$$\frac{d\mathbf{v}}{dt} = \mathbf{a}(t)$$

If we add the Taylor expansion of $\mathbf{r}(\mathbf{t}+\triangle\mathbf{t})$ and $\mathbf{r}(\mathbf{t}-\triangle\mathbf{t})$ we get

$$\begin{aligned}\mathbf{r}(t+\triangle t) &= \mathbf{r}(t)+\triangle t\ \mathbf{v}(t)+\frac{\triangle t^2}{2}\mathbf{a}(t)+O(t^4)\\ \mathbf{r}(t-\triangle t) &= \mathbf{r}(t)-\triangle t\ \mathbf{v}(t)+\frac{\triangle t^2}{2}\mathbf{a}(t)+O(t^4)\\ \mathbf{r}(t+\triangle t) &= \mathbf{r}(t)-\mathbf{r}(t-\triangle t)+\frac{\triangle t^2}{2}\mathbf{a}(t)\end{aligned} \tag{5.7.1}$$

where we have replaced the first and second derivative of the position with it's velocity and acceleration. This approximation is called Verlet's method .

This is ideally suited to problems where we can write down Newton's second law .

As you can see the Verlet method requires 2 position terms ($\mathbf{r}(t)$ and $\mathbf{r}(t-\triangle t)$) to calculate the next position coordinates. Like the Numerov method we need some other method to calculate a value of $\mathbf{r}(t)$ before we can start the Verlet method. Methods like this are referred as non self-starting . In the next section we will look at variations of the Verlet methods that are self starting.

$$\mathbf{a}(t)=\frac{\mathbf{F}(\mathbf{r},t)}{m} \tag{5.7.2}$$

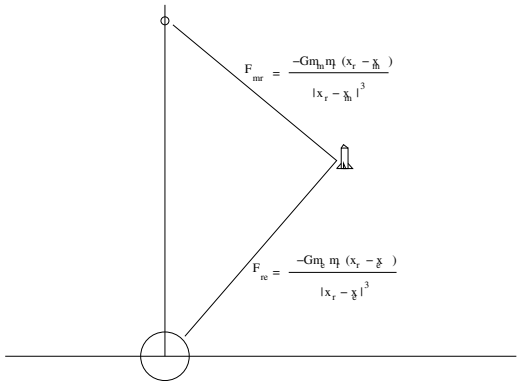


Figure 5.3: The gravitational forces on a rocket on a trajectory between the Earth and the moon can be resolved into x and y components. The trajectory can then easily be solved by any O.D.E. solver

Example:
Suppose we wish to track the trajectory of a rocket from when it is launched on the earth until it reaches it's destination, say, the moon (see fig 5.3). We can resolve the forces acting on the rocket into

$$\begin{aligned}\mathbf{F}(\mathbf{x}) &= -\frac{Gm_{\text{rocket}}m_{\text{earth}}(x-x_{\text{earth}})}{|x-x_{\text{earth}}|^3}-\frac{Gm_{\text{rocket}}m_{\text{moon}}(x-x_{\text{moon}})}{|x-x_{\text{moon}}|^3}\\ \mathbf{F}(\mathbf{y}) &= -\frac{Gm_{\text{rocket}}m_{\text{earth}}(y-y_{\text{earth}})}{|y-y_{\text{earth}}|^3}-\frac{Gm_{\text{rocket}}m_{\text{moon}}(y-y_{\text{moon}})}{|y-y_{\text{moon}}|^3}\end{aligned} \tag{5.7.3}$$

and thus write an expression for the components of the acceleration

$$\begin{aligned}\mathbf{a}(\mathbf{x}) &= -\frac{Gm_{\text{earth}}(x-x_{\text{earth}})}{|x-x_{\text{earth}}|^3}-\frac{Gm_{\text{moon}}(x-x_{\text{moon}})}{|x-x_{\text{moon}}|^3}\\ \mathbf{a}(\mathbf{y}) &= -\frac{Gm_{\text{earth}}(y-y_{\text{earth}})}{|y-y_{\text{earth}}|^3}-\frac{Gm_{\text{moon}}(y-y_{\text{moon}})}{|y-y_{\text{moon}}|^3}\end{aligned} \tag{5.7.4}$$

If we assume that the rocket starts at time $t=t+\Delta t$ then we can set $\mathbf{r}(t-\Delta t)$ to be the launch position on the earth. We can now decide on a value for Δt and use (5.7.1) to evolve the position. However, to calculate $\mathbf{r}(t+\Delta t)$ we need an estimation for $\mathbf{r}(t)$ This is the shortcoming of this method, like the Numerov method we have just seen we need to use another method to start it.

5.8 Verlet Method(s)

There are several variations on the Verlet method that are self starting (i.e. do not require the calculation intermediate step). I will just state them here.

Velocity Verlet method

$$\begin{aligned}\mathbf{r}(t+\Delta t) &= \mathbf{r}(t)+\Delta t\ \mathbf{v}(t)+\frac{\Delta t^2}{2}\mathbf{a}(t)\\ \mathbf{v}(t+\Delta t) &= \mathbf{v}(t)+\frac{1}{2}\Delta t\Big(\mathbf{a}(t)+\mathbf{a}(t+\Delta t)\Big)\end{aligned} \tag{5.8.1}$$

Leap-frog Verlet

$$\begin{aligned}\mathbf{r}(t+\Delta t) &= \mathbf{r}(t)+\Delta t\ \mathbf{v}(t+\frac{\Delta t}{2})\\ \mathbf{v}(t+\frac{\Delta t}{2}) &= \mathbf{v}(t-\frac{\Delta t}{2})\ \mathbf{r}(t)+\Delta t\ \mathbf{a}(t)\end{aligned} \tag{5.8.2}$$

Position Verlet method

$$\begin{aligned} \mathbf{r}(t+\Delta t) &= \mathbf{r}(t) + \frac{\Delta t}{2}\left[\mathbf{v}(t) + \mathbf{v}(t+\Delta t)\right] \\ \mathbf{v}(t+\Delta t) &= \mathbf{v}(t) + \Delta t\ \mathbf{a}(t + \frac{\Delta t\ \mathbf{v}(t)}{2}) \end{aligned} \tag{5.8.3}$$

6

Partial Differential Equations

Partial Differential Equations (P.D.E.s) occur in almost every physical situation where quantities vary in space or in space and time, such as hydrodynamics, electromagnetic waves, diffusion and quantum mechanics. Except in the simplest cases these P.D.E.s cannot be solved analytically so we need to have numerical methods to deal with them. Like O.D.E.s. P.D.E.s are classified by their order and whether they are linear or not.

Numerical methods for solving Partial Differential Equations (P.D.E.s) is a vast subject, however, the most common P.D.E.s encountered by physics undergraduates are second order so we will limit our discussion here to the basic techniques for solving second order P.D.E.s.

Broadly speaking 2^{nd} order P.D.E.s are categorised into 3 types:

Parabolic, which contain a first order derivative in one variable and second order derivatives in the remaining variables e.g. the diffusion equation or Schröedingers equation,

Elliptical having second order derivatives in all the independent variables, and which all have the same sign when grouped on the same side of the equation e.g. Laplace's equation, Poisson's equation.

and *Hyperbolic*. which have second order derivatives of all variables but one the second derivative of one variable has an opposite sign such as the wave equation

6.1 Finite Differencing

In a typical numerical treatment, the dependent variables are described by their values at discrete points (a lattice) of the independent variables (e.g. space and/or time), and the partial differential equation is reduced to a large set of difference equations. It would be useful to

revise our description of difference equations, as a refresher I have listed them below.

$$\begin{aligned}
\text{forward difference} &: f'(x) \approx \frac{f_{n+1} - f_n}{h} \\
\text{backward difference} &: f'(x) \approx \frac{f_n - f_{n-1}}{h} \\
\text{centered difference} &: f'(x) \approx \frac{f_{n+1} - f_{n-1}}{2h} \\
\text{centered difference} &: f''(x) \approx \frac{f_{n+1} - 2f_n + f_{n-1}}{h^2}
\end{aligned} \tag{6.1.1}$$

Make sure you know how these finite difference formulae are derived.

6.2 Elliptical P.D.E.s

We will consider the (almost) general 1D elliptical equation, Poisson's equation :

$$\frac{\partial^2 \phi}{\partial x^2} = S(x) \tag{6.2.1}$$

and cast it into a suitable form for a numerical treatment.

Let us define a lattice (array) of N sites in the x plane that spans our region of interest. Let the lattice spacing be δx and we label each point in our lattice by the indices i which run from $0 \ldots N$, so that the coordinates of each point are $x = i\delta x$. For simplicity we will refer to $\phi(x_i)$ as ϕ_i and similarly S_i for $S(x_i)$.

It is straightforward to apply the (3 point) central difference approximation to the second derivatives in (6.2.1) to obtain an approximation

$$\frac{\phi_{i-1} - 2\phi_i + \phi_{i+1}}{(\delta x)^2} = S_i \tag{6.2.2}$$

With boundary conditions we can write our approximations as a system of linear equations. For example, in 1D with $\phi(0) = \phi_0$ and

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \ldots & 0 \\ 1 & -2 & 1 & 0 & \ldots & 0 \\ 0 & 1 & -2 & 1 & \ldots & 0 \\ 0 & 0 & 1 & -2 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 \end{bmatrix} \begin{bmatrix} \phi(x_0) \\ \phi(x_1) \\ \phi(x_2) \\ \phi(x_3) \\ \vdots \\ \phi(x_N) \end{bmatrix} = \begin{bmatrix} \phi_0 \\ (\delta x)^2 S_1 \\ (\delta x)^2 S_2 \\ (\delta x)^2 S_3 \\ \vdots \\ \phi_N \end{bmatrix}$$

We can solve this system of equations to find the values of $\phi(x)$ at each of our lattice points x_i. We will discuss solving systems of linear equations in our next lecture. We will see that this may involve a lot of computation for a large number of lattice points. Often, a more useful approach is to use an iterative procedure.

6.2.1 Gauss Siedel Method

We can 'solve' our approximation for (6.2.1) at each lattice point as

$$\phi_i = \left[\frac{\phi_{i+1} + \phi_{i-1} - S_i \delta x^2}{2}\right] \tag{6.2.3}$$

We can easily create an iterative method based on the above approximation (actually this has already been done and is called the Gauss-Siedel method).

An iterative approach would work as follows: Guess an initial solution for each ϕ_i, a normal approach is to set ϕ_i to 0 inside the lattice and apply the boundary conditions at the boundaries. We can then sweep through the lattice starting at ϕ_1 and apply the above approximation (6.2.3) to each ϕ_i. After many sweeps we should find that the values of ϕ_i won't change much from sweep to sweep, we can take these values of ϕ_1 as our result.

6.2.2 Successive Over-Relaxation

An iterative method based on the above formula is quiet inefficient. A modification of the above is often used instead, in it we consider

$$\phi_i^{m+1} = \alpha\hat{\phi}_i + (1-\alpha)\phi_i^m$$

where $\hat{\phi}_i$ is calculated from (6.2.2) and m is the iteration number. If the lattice is uniform then this can be rewritten as

$$\phi_i^{m+1} = \phi_i^m + \frac{\alpha R_{i,j}}{2}$$

where $R_{i,j} = \phi_{i+1} + \phi_{i-1} - 2\phi_{i,j}$ and is known as the residual. We would like this to be zero for all i because then the ϕ_i^{m+1} would be equal to ϕ_i^m i.e. the ϕ_i is not changing with each iteration and we can stop. This, in general, will not happen and so the object of the relaxation method is to devise a method of reducing the residuals so that they get smaller with each successive iteration to a point that we can decide to stop the calculation.

The parameter α is called the *over-relaxation* parameter. This method is convergent for $0 < \alpha < 2$ and if $0 < \alpha < 1$ we speak of *under-relaxation* and of *over-relaxation* for $1 < \alpha < 2$.

The value of α that is chosen depends on the problem and the optimum value is not easy to calculate analytically. In fact, it depends on the analytic solution to the problem you are solving numerically! This weakness in the S.O.R. method is overcome by mapping the problem to one in which the analytic solution is known. For the simplest 2D case of the Laplaces equation (if we let $S(x,y) = 0$ in our example we get Laplaces equation) on a square with Dirichlet boundaries, it can be shown that the best value of α is the smaller root of

$$\left[\cos\left(\frac{\pi}{m_x+1}\right) + \cos\left(\frac{\pi}{m_y+1}\right)\right]\alpha^2 - 16\alpha - 16 = 0 \tag{6.2.4}$$

We will now turn our attention to elliptical equations in 2D. We will consider (again) Poissons equation,

$$-\left[\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right]\phi = S(x,y) \tag{6.2.5}$$

We define an NxN lattice in the (x,y) plane that spans our region of interest. Let the lattice spacing be δx and δy and we label each point in our lattice by the indices (i,j) which run from $0 \ldots N$, so that the coordinates of each point (i,j) are $(x = i\delta x, y = j\delta y)$. For simplicity we will refer to $\phi(x_i, y_j)$ as ϕ_{ij} and similarly we will refer to $\S(x_i, y_j)$ as S_{ij}.

Applying the (3 point) central difference approximation to the second derivatives in (6.2.5) we get

$$-\left[\frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{(\delta x)^2} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{(\delta y)^2}\right] = S_{i,j} \tag{6.2.6}$$

which we can rearrange to 'solve' for ϕ_{ij}.

$$\phi_{i,j} = \frac{S_{i,j}\delta x^2\delta y^2 + \delta x^2\left(\phi_{i,j+1} + \phi_{i,j-1}\right) + \delta y^2\left(\phi_{i+1,j} + \phi_{i-1,j}\right)}{2\left(\delta x^2 + \delta y^2\right)} \tag{6.2.7}$$

The Gauss-Siedel method can be used to solve this P.D.E. on an NxN grid such as that in fig 6.1

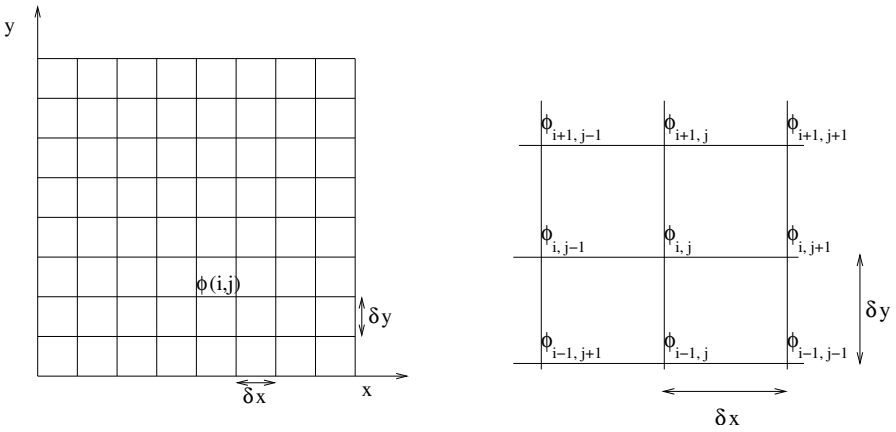


Figure 6.1: Lattice layout for the solution of an elliptical P.D.E. using Gauss-Siedel method. Each lattice site is updated with a function of the values at its neighbouring sites.

We set every $\phi_{ij} = 0$ initially and iterate across the lattice according to

$$\phi_{i,j}^{m+1} = \frac{S_{i,j}\delta x^2\delta y^2 + \delta x^2\left(\phi_{i,j+1}^m + \phi_{i,j-1}^m\right) + \delta y^2\left(\phi_{i+1,j}^m + \phi_{i-1,j}^m\right)}{2\left(\delta x^2 + \delta y^2\right)} \tag{6.2.8}$$

where the label m denotes the iteration.

Example:
Solve Laplace's equation on a 3x3 grid where $\phi = 0°C$ along $x = 0$, $\phi = 100°C$ along $x = L$, $\phi = 75°C$ along $y = 0$ and $\phi = 50°C$ along $y = L$
The finite difference for Laplace's equation is (the same as Poisson's equation with $S_{ij} = 0$)

$$\phi^{m+1}_{i,j} = \frac{\delta x^2 \left(\phi^m_{i,j+1} + \phi^m_{i,j-1}\right) + \delta y^2 \left(\phi^m_{i+1,j} + \phi^m_{i-1,j}\right)}{2\left(\delta x^2 + \delta y^2\right)}$$

The lattice for the above
We will assume $\delta x = \delta y = 1$, although this will depend on the length of the lattice, L, and the number of grid points, N. Thus our iterative approximation is given by

$$\phi^{m+1}_{i,j} = \frac{\phi^m_{i,j+1} + \phi^m_{i,j-1} + \phi^m_{i+1,j} + \phi^m_{i-1,j}}{4}$$

Which we can apply iteratively along with our boundary conditions until each successive iteration doesn't change ϕ_{ij} by more than a desired accuracy.

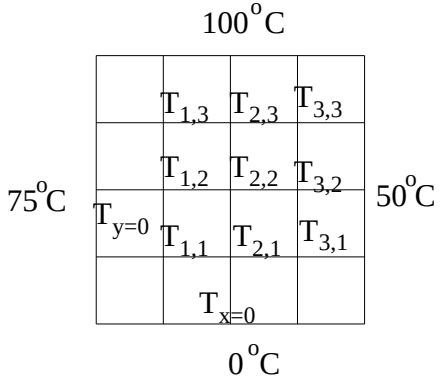


Figure 6.2: Grid for a solution of Laplace's equation with 3x3 points

6.3 Parabolic P.D.E.s

In contrast to elliptical equations where the boundary conditions are set here the problem is generally posed with initial values, i.e. we are given information about the field ϕ at an initial time and we need to evolve our solution subject to some restraining conditions e.g. the wave function vanishing at large r.

49.61	39.16	35.30
23.44	7.03	2.05
18.75	4.69	1.17

77.94	73.31	57.85
63.44	57.48	58.09
43.36	34.99	39.13

Figure 6.3: Solution for Laplaces equation on a 3x3 lattice with boundary condition given in fig 6.2. The grid on the left shows the solution after one iteration, the grid on the left after the solution has relaxed to 2 significant places of decimal (after 30 iterations).

The typical parabolic equations encountered in physics are the diffusion equation

$$\frac{\partial \phi}{\partial t} - \kappa \frac{\partial^2 \phi}{\partial x^2} = 0 \tag{6.3.1}$$

and the time-dependent Schröedinger equation

$$i\hbar \frac{\partial \phi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2 \phi}{\partial x^2} + V\phi \tag{6.3.2}$$

For the rest of this section we will concentrate on the diffusion equation. However, the methods outlined here can easily be used to derive a form for the Schröedinger equation. We begin by approximating the differential terms with their finite differences

$$\begin{aligned} \frac{\partial \phi}{\partial t} &= \frac{\phi_{n,j+1} - \phi_{n,j}}{\delta t} \\ \frac{\partial^2 \phi}{\partial x^2} &= \frac{\phi_{n+1,j} - 2\phi_{n,j} + \phi_{n-1,j}}{\delta x^2} \end{aligned}$$

where we have used the, by now usual, notation $\phi(x_n, t_j) = \phi_{n,j}$

Using these approximations for the diffusion equation (6.3.1) yields

$$\frac{\phi_{n,j+1} - \phi_{n,j}}{\delta t} - \kappa \frac{\phi_{n+1,j} - 2\phi_{n,j} + \phi_{n-1,j}}{\delta x^2} = 0$$

We can now solve this for either $\phi_{n,j+1}$ (forward difference) to give

$$\phi_{n,j+1} = \phi_{n,j} + \frac{\kappa \delta t}{\delta x^2}\left(\phi_{n+1,j} - 2\phi_{n,j} + \phi_{n-1,j}\right)$$

of for $\phi_{n,j-1}$ (using the backward difference approximation for $\partial\phi/\partial t$) to give

$$\phi_{n,j-1} = \phi_{n,j} - \frac{\kappa \delta t}{\delta x^2}\left(\phi_{n+1,j} - 2\phi_{n,j} + \phi_{n-1,j}\right)$$

Exercise:
Derive the forward and backward difference schemes for the Schrödinger equation.

6.3.1 Stability

To discuss the stability of our simply scheme we will use a trial solution of the form

$$a(x,t) = A(t)e^{ikx} \tag{6.3.3}$$

i.e. we try a wave with wave number, k, and (complex) amplitude, $A(t)$. In discretized form we have

$$a(x_n, t_j) = a_{n,j} = A_j e^{ink\delta x}$$

where we have written $x_n = n\delta x$ and $t_j = (j-1)\delta t$. Advancing the solution by one step we have

$$a_{n,j+1} = A_{j+1}e^{ink\delta x} = \zeta A_j e^{ink\delta x}$$

The coefficient $\zeta = A_{j+1}/A_j$ is called the *amplification factor* . We say that a scheme is unstable if the magnitude of the amplification factor is greater than unity, i.e. $|\zeta| > 1$. This approach is called *Von Neumann stability analysis*.

We shall apply the Von Neumann stability analysis to our diffusion equation

$$\phi_{n,j+1} = \phi_{n,j} + \frac{\kappa\delta t}{\delta x^2}\left(\phi_{n+1,j} - 2\phi_{n,j} + \phi_{n-1,j}\right)$$

$$\begin{aligned}
\zeta A_j e^{ink\delta x} &= A_j e^{ink\delta x} + \frac{\kappa\delta t}{\delta x^2}\left(A_j e^{i(n+1)k\delta x} - 2A_j e^{ink\delta x} + A_j e^{i(n-1)k\delta x}\right)\\
\zeta A_j e^{ink\delta x} &= A_j e^{ink\delta x} + \frac{\kappa\delta t}{\delta x^2}\left(A_j e^{ink\delta x}e^{ik\delta x} - 2A_j e^{ink\delta x} + A_j e^{ink\delta x}e^{-ik\delta x}\right)\\
\zeta A_j e^{ink\delta x} &= A_j e^{ink\delta x}\left[1 + \frac{\kappa\delta t}{\delta x^2}\left(e^{ik\delta x} - 2 + e^{-ik\delta x}\right)\right]\\
\zeta &= 1 + \frac{\kappa\delta t}{\delta x^2}\left(-2 + 2\cos(k\delta x)\right)\\
\zeta &= 1 - \frac{2\kappa\delta t}{\delta x^2}\left(1 - \cos(k\delta x)\right)
\end{aligned}$$

The stability condition is that $|\zeta| \leq 1$. So

$$-1 \leq 1 - \frac{\kappa\delta t}{\delta x^2}\left(1 - \cos(k\delta x)\right) \leq 1 \tag{6.3.4}$$

The maximum and minimum values of the cosine term are $1, -1$ respectively, so we can insert these values for the cosine term, to obtain our stability condition. Inserting $\cos(k\delta x) = 1$, the central term becomes 1, satisfying the condition (6.3.4).

If we insert $\cos(k\delta x) = -1$, the central term becomes $1 - 4\frac{\kappa\delta t}{\delta x^2}$, since $\kappa, \delta t, \delta x$ are positive (we have defined $\kappa > 0$ in 6.3.1) the central term is less than 1. For the central term to be greater than -1 we need

$$-1 \leq 1 - 4\frac{\kappa\delta t}{\delta x^2}$$

or

$$\delta t \leq \frac{\delta x^2}{2\kappa}$$

This is the stability condition for the forward difference scheme. If we use the backward difference scheme we find that it is unconditionally stable. However, the truncation error (the error from ignoring terms in the Taylor expansion) is linear in δt and goes as δx^2 in the spatial term, i.e. $\mathcal{O}(\delta t + \delta x^2)$.

6.3.2 Crank Nicholson Method

It turns out that we can average the forward and backward difference schemes and the linear terms in δt cancel out giving a method with $\mathcal{O}(\delta t^2 + \delta x^2)$. This is known as the Crank-Nicholson method and we'll just state it here

$$\frac{\phi_{i,j+1} - \phi_{i,j}}{\delta t} - \frac{\kappa}{2}\left[\frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{\delta x^2} + \frac{\phi_{i+1,j+1} - 2\phi_{i,j+1} + \phi_{i-1,j+1}}{\delta x^2}\right] = 0$$

6.3.3 Implicit Schemes

In our derivation of the finite difference approximation of the diffusion equation we took the second derivative of the spatial term at time, t_j. Now, we will take the second derivative at time t_{j+1}, our finite differences are now

$$\begin{aligned}
\frac{\partial\phi}{\partial t} &= \frac{\phi_{n,j+1} - \phi_{n,j}}{\delta t}\\
\frac{\partial^2\phi}{\partial x^2} &= \frac{\phi_{n+1,j+1} - 2\phi_{n,j+1} + \phi_{n-1,j+1}}{\delta x^2}
\end{aligned}$$

If we now insert these approximations in our diffusion equation and obtain a forward difference.

$$\phi_{n,j+1} = \phi_{n,j} + \frac{\kappa\delta t}{\delta x^2}\left(\phi_{n+1,j+1} - 2\phi_{n,j+1} + \phi_{n-1,j+1}\right)$$

If we now let the the boundary conditions be written as ϕ_0 and ϕ_N then our implicit finite difference scheme can be written simply as

$$\begin{bmatrix}
1 & 0 & 0 & 0 & \dots & 0 & 0 & 0\\
-\lambda & 1+2\lambda & -\lambda & 0 & \dots & 0 & 0 & 0\\
0 & -\lambda & 1+2\lambda & -\lambda & \dots & 0 & 0 & 0\\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots\\
0 & 0 & 0 & 0 & \dots & -\lambda & 1+2\lambda & -\lambda\\
0 & 0 & 0 & 0 & \dots & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\phi_{0,j+1}\\ \phi_{1,j+1}\\ \phi_{2,j+1}\\ \phi_{3,j+1}\\ \vdots\\ \phi_{N,j+1}
\end{bmatrix}
=
\begin{bmatrix}
\phi_{0,j}\\ \phi_{1,j}\\ \phi_{2,j}\\ \phi_{3,j}\\ \vdots\\ \phi_{N,j}
\end{bmatrix}$$

Here we have written $\lambda = \frac{\kappa\delta t}{\delta x^2}$ for simplicity. We can now solve this system of linear equations at each time t_{j+1} if we have appropriate initial conditions. We will develop methods for this in the next chapter.

If we do Von Neumann stability analysis we find that

$$\zeta = \frac{1}{1 + 2\lambda\left(1 - \cos(k\delta x)\right)}$$

so $|\zeta| \leq 1$ always and so the scheme is unconditionally stable.

6.3.4 2D

The methods and stability analysis developed here in 1D can easily be extended to 2D. If we consider the diffusion equation again we can easily write a finite difference approximation as

$$\frac{\phi_{n,m,j+1} - \phi_{n,m,j}}{\delta t} = \kappa\frac{\phi_{n+1,m,j} - 2\phi_{n,m,j} + \phi_{n-1,m,j}}{\delta x^2} + \kappa\frac{\phi_{n,m+1,j} - 2\phi_{n,m,j} + \phi_{n,m-1,j}}{\delta y^2}$$

From which we can write a forward time difference scheme (FTDS) as

$$\phi_{n,m,j+1} = \phi_{n,m,j} + \lambda_1\left(\phi_{n-1,m,j} - 2\phi_{n,m,j} + \phi_{n+1,m,j}\right) + \lambda_2\left(\phi_{n,m-1,j} - 2\phi_{n,m,j} + \phi_{n,m+1,j}\right) \quad (6.3.5)$$

For simplicity we have written

$$\lambda_1 = \frac{\kappa\delta t}{\delta x^2}$$

and

$$\lambda_2 = \frac{\kappa\delta t}{\delta y^2}$$

This difference scheme has only one unknown (on the left hand side) so we can solve for $\phi_{n,m,j+1}$ at each time step t_{j+1} and repeat this until the final $t_j = t_0 + j\delta t$ is reached. This particular scheme is first order accurate in time and second order accurate in both the δx and δy terms. We can apply stability analysis to this scheme to find that it is stable if

$$\lambda_1 + \lambda_2 \leq \frac{1}{2}$$

The iterative procedure outlined above can be seen in fig 6.4.

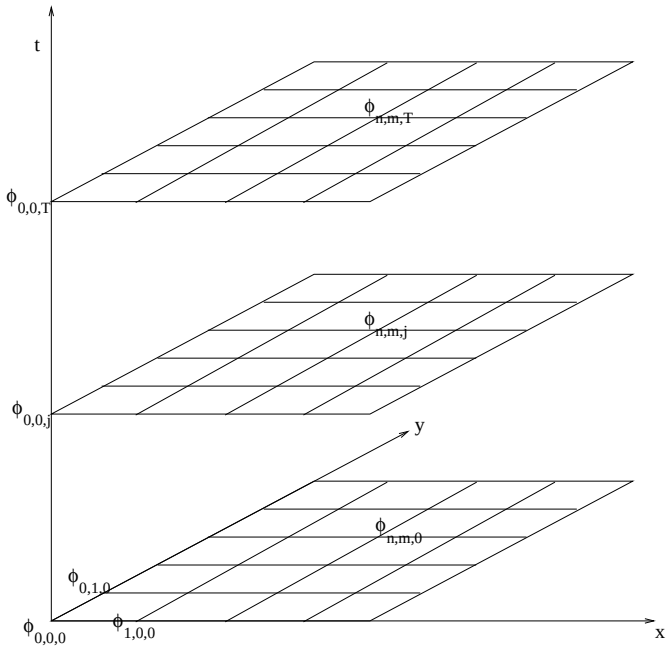


Figure 6.4: Representation of a 2D grid to solve a parabolic P.D.E. The initial conditions (the bottom grid) should be set we solve for the new time (move up the grid) until we reach $t_j = T$ our solution. It should be noted that in a computational calculation using this scheme we use a 2D array and continually update the array rather than use a 3D array.

6.4 Hyperbolic P.D.E.s

We will end our discussion of P.D.E.s by looking at the final type, hyperbolic P.D.E.s, and use the wave equation in 1D as our example

$$\frac{\partial^2\phi}{\partial t^2} = c^2\frac{\partial^2\phi}{\partial x^2} \quad (6.4.1)$$

We can again use our central difference equations to get

$$\begin{aligned}\frac{\phi_{i,j-1} - 2\phi_{i,j} + \phi_{i,j+1}}{\delta t^2} &= c^2\frac{\phi_{i-1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{\delta x^2} \\ \phi_{i,j+1} &= 2\left(1 - \left(\frac{c\delta t}{\delta x}\right)^2\right)\phi_{i,j} + \left(\frac{c\delta t}{\delta x}\right)^2\left(\phi_{i-1,j} + \phi_{i+1,j}\right) - \phi_{i,j-1} \quad (6.4.2)\end{aligned}$$

You should derive these formulae as an exercise.

Exercise:
Use stability analysis to show that this scheme is only stable if $\frac{c\delta t}{\delta x} \leq 1$

Since the method requires 2 time steps j and $j-1$ to calculate $\phi_{i,j+1}$, How do we start the method? We are usually given $\phi_{i,0}$ as initial conditions but we need $\phi_{i,0}$ and $\phi_{i,1}$ to calculate $\phi_{i,2}$, or $\phi_{i,-1}$ and $\phi_{i,0}$ to calculate $\phi_{i,1}$.

We will assume boundary conditions $\phi(0,t) = \phi_{0,t}$, $\phi(l,t) = \phi_{l,t}$ and initial conditions $\phi(x,0) = f(x)$ and $\frac{\partial\phi(x,0)}{\partial t} = g(x)$.

We can then use a centered difference approximation for $\partial\phi_{i,0}/\partial t$

$$\frac{\partial\phi_{x_i,0}}{\partial t} = \frac{\phi_{i,1} - \phi_{i,-1}}{2\delta t} = g(x_i) = g_i$$

and 'solve' this for $\phi_{i,-1}$ as $\phi_{i,-1} \approx \phi_{i,1} - 2\delta t g_i$ which we can use to to start our solution.

$$\begin{aligned}
\phi_{i,1} &= 2\left(1 - \left(\frac{c\delta t}{\delta x}\right)^2\right)\phi_{i,0} + \left(\frac{c\delta t}{\delta x}\right)^2 (\phi_{i-1,0} + \phi_{i+1,0}) - \phi_{i,-1} \\
&= 2\left(1 - \left(\frac{c\delta t}{\delta x}\right)^2\right) f_i + \left(\frac{c\delta t}{\delta x}\right)^2 (f_{i-1} + f_{i+1}) - \phi_{i,1} + 2\delta t g_i \\
&= \left(1 - \left(\frac{c\delta t}{\delta x}\right)^2\right) f_i + \frac{1}{2}\frac{c\delta t^2}{\delta x} (f_{i-1} + f_{i+1}) + \delta t g_i
\end{aligned} \tag{6.4.3}$$

Where we have simplified $\phi_{i,0} = \phi(x_i, 0) = f(x_i) = f_i$. Now given our $\phi_{i,0} = f_i$ and ϕ_{i_1} we can calculate $\phi_{i,2}$ and so on.

We have only really scratched the surface of P.D.E.s here. There have been developed methods and schemes that have both high order accuracy and unconditionally stability. These methods, in one way or another, are based on the methods developed here, and you should be able to apply these methods when and where you need them and understand the stability considerations.

7

Linear Algebra

Our goal for the first section of this chapter is to calculate the solutions to a system of equations such as

$$\begin{aligned} A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n &= B_1 \\ A_{21}x_1 + A_{22}x_2 + \ldots + A_{2n}x_n &= B_2 \\ &\ldots \\ A_{n1}x_1 + A_{n2}x_2 + \ldots + A_{nn}x_n &= B_n \end{aligned} \tag{7.0.1}$$

(7.0.2)

A system of equations like (7.0.2) can be written as a matrix equation

$$\begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1n} \\ A_{21} & A_{22} & \ldots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nn} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} \tag{7.0.3}$$

In matrix terms what we want is a method that *diagonalises* the matrix A while also transforming the the values of the B vector, so that we can simply read off the corresponding values of the vector B as solutions. When we talk about diagonalising a matrix what we mean is manipulating the matrix so that all the values along the main diagonal (from the top left to the bottom right) are 1 and all the other values are 0.

Since we will be discussing methods that act on matrices throughout this chapter we will end the chapter looking at eigenvalue problems.

7.1 Gaussian Elimination

A simply method of solving this matrix equation is Gaussian elimination, often referred to as the Gauss-Jordan method. The first step is to merge the matrix A and the vector B into an $N + 1 \times N$ matrix, referred to as an augmented matrix .

$$\begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1n} & B_1 \\ A_{21} & A_{22} & \ldots & A_{2n} & B_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nn} & B_n \end{bmatrix} \tag{7.1.1}$$

The object is to set the diagonals of the matrix to one and set all the other elements (except for those in the last column) to zero by applying a series of operations to each row. These operations are chosen so that one of the coefficients are removed at a time by subtracting rows in the augmented matrix.

Lets look at an application of Gaussian-elimination in solving the system of equations

$$\begin{aligned} 2x_1 + 3x_2 + 1x_3 &= 5 \\ -1x_1 + 4x_2 + 2x_3 &= -3 \\ 3x_1 - 2x_2 - 3x_3 &= 5 \end{aligned} \tag{7.1.2}$$

First we form the augmented matrix

$$\begin{bmatrix} 2 & 3 & 1 & 5 \\ -1 & 4 & 2 & -3 \\ 3 & -2 & -3 & 5 \end{bmatrix}$$

Our first task is to set a 1 in position A_{11}, we can do this by dividing row 1 by $A_{11}(= 2)$. We now get

$$\begin{bmatrix} 1 & 3/2 & 1/2 & 5/2 \\ -1 & 4 & 2 & -3 \\ 3 & -2 & -3 & 5 \end{bmatrix}$$

Our next task is to get a 0 in positions A_{21} and A_{31}. First lets think of this operation in terms of our equations. Each row in the matrix corresponds to an equations so to eliminate x_1 from the second equation we simply add equations 1 and 2. In the most general sense we will multiply each element in row 2 by A_{21} and subtract the corresponding column in row 1. For example, to eliminate x_1 from row 2 we have

$$\begin{aligned} x_1 + 3/2x_2 + 1/2x_3 &= 5/2 \\ \text{subtracting } -1 \text{ times } -x_1 + 4x_2 + 2x_3 &= -3 \\ \text{gives } 0x_1 + 11/2x_2 + 5/2x_3 &= -1/2 \end{aligned} \tag{7.1.3}$$

We now remove the x_1 term from row 3 by subtracting 3 times the corresponding column element from row 1 from each element in row 3 i.e. $A_{3c} = A_{3c} - 3A_{1c}$. We now get

$$\begin{bmatrix} 1 & 3/2 & 1/2 & 5/2 \\ 0 & 11/2 & 5/2 & -1/2 \\ 0 & -13/2 & -9/2 & -5/2 \end{bmatrix}$$

We now turn our attention to the diagonal term in row 2 and those coefficients of x_2 i.e. column 2. We can get a 1 in the diagonal position by dividing row 2 by 11/2 this leaves

$$\begin{bmatrix} 1 & 3/2 & 1/2 & 5/2 \\ 0 & 1 & 5/11 & -1/11 \\ 0 & -13/2 & -9/2 & -5/2 \end{bmatrix}$$

To get a zero in the other elements of column 2 we need to subtract $-13/2$ times row 3 from row 2 and 3/2 times row 1 from row 2. This will not affect the diagonals we have already calculated. Our matrix is now

$$\begin{bmatrix} 1 & 0 & -4/22 & 51/22 \\ 0 & 1 & 5/11 & -1/11 \\ 0 & 0 & -34/22 & -42/22 \end{bmatrix}$$

Getting our final diagonal element is now easy - we divide row 3 by $-34/22$. Following our previous method we can set the other column elements to 0 by replacing every element in row r with $A_{rc} = A_{rc} - A_{r3}A_{3c}$.

$$\begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 2 \end{bmatrix}$$

From which we can easily read the results $x_1 = 3$, $x_2 = -1$, $x_3 = 2$.

We can now write down the sequence of operations required to solve a system of equations via Gaussian elimination.

1 Starting at A_{11}, ensure that this element is not zero, if it is then swap another row so that the element in position A_{11} is non-zero. This steps corresponds to swapping the order we write our equations. In methods similar to this one this step is called pivoting.

2 Divide each element in this row by A_{11}. This ensures that the element in position A_{11} is now 1. Again thinking back to simultaneous equations, this step is the same as dividing each term by A_{11}.

3 Replace each element, A_{2c}, in row 2 by $A_{2c} = A_{2c} - A_{21}A_{1c}$ and $B_2 = B_2 - A_{21}B_1$. Now we should have a zero in position A_{21}.

4 Repeat the last step for every other row, r, replacing A_{rc} with $A_{rc} - A_{r1}A_{1q}$ and $B_r = B_r - A_{r1}B_1$.

5 Our matrix should now look like

$$\begin{bmatrix} 1 & A'_{12} & \dots & A'_{1N} & B'_1 \\ 0 & A_{22} - A_{21}A'_{12} & \dots & A_{2n} - A_{21}A'_{1n} & B_2 - A_{21}B'_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & A_{n2} - A_{n1}A'_{12} & \dots & A_{nn} - A_{n1}A'_{1n} & B_n - A_{n1}B'_n \end{bmatrix}$$

6 Repeat steps 1-4 for each diagonal element. If we find that we get a zero in the last diagonal position then the system of equations is not solvable.

This method is not the most efficient though as we are zeroing all the off-diagonal elements. A variation of the Gauss-Jordan method which does not do the back-substitution can be more efficient.

7.1.1 Back Substitution

The number of operations in the procedure above can be reduced if we only zero those elements which lie **below** the diagonal and we are not worried about getting a 1 along the diagonal. We are aiming to obtain a matrix in the form.

$$\begin{bmatrix} A'_{11} & A'_{12} & \dots & A'_{1n} & B'_1 \\ 0 & A'_{22} & \dots & A'_{2n} & B'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & A'_{nn} & B'_n \end{bmatrix} \tag{7.1.4}$$

To do this we apply a similar elimination procedure as the one above. We only need to zero the elements in the column below the diagonal element we have worked on.

Example:
Solve the problem in the previous section using Gaussian elimination with back-substitution

$$\begin{bmatrix} 2 & 3 & 1 & 5 \\ -1 & 4 & 2 & -3 \\ 3 & -2 & -3 & 5 \end{bmatrix}$$

We can eliminate the -1 from row 2 by replacing every element in that row by $A_{2q} - \frac{A_{21}A_{1q}}{A_{11}} = A_{2q} + \frac{A_{1q}}{2}$, and then eliminate the 3 from row 3 by replacing every element with $A_{3q} - \frac{A_{31}A_{1q}}{A_{11}} = A_{3q} + \frac{3A_{1q}}{2}$. This gives us

$$\begin{bmatrix} 2 & 3 & 1 & 5 \\ 0 & 11/2 & 5/2 & -1/2 \\ 0 & -13/2 & -9/2 & -5/2 \end{bmatrix}$$

The final problem is to eliminate the $-13/2$ from element A_{32}, we can do this by replacing every element in row 3 by $A_{3q} - \frac{A_{32}A_{2q}}{A_{22}} = A_{3q} + \frac{(-13/2)A_{1q}}{11/2}$. Doing this step gives us

$$\begin{bmatrix} 1 & 3/2 & 1/2 & 5/2 \\ 0 & 1 & 5/11 & -1/11 \\ 0 & 0 & 1 & 2 \end{bmatrix}$$

The final step is the back-substitution step. We can easily read

$$x_n = \frac{B'_n}{A'_{nn}}$$

from the n^{th} row in (7.1.4). We can insert this to the $(n-1)^{\text{th}}$ row, to calculate x_{n-1} and so on. The formula for this step is

$$\begin{aligned} x_n &= \frac{B'_n}{A'_{nn}} \\ x_{n-1} &= B'_{n-1} - x_n A'_{n-1,n} \\ &\dots \end{aligned} \tag{7.1.5}$$

Exercise:
Show that doing the back-substitution step in the above example gives the same results as the Gauss-Jordan method.

7.1.2 Matrix inverse

The Gauss Jordan method can be used to find the inverse of a matrix. If A is a matrix an I is the identity matrix then

$$A = AI = IA \tag{7.1.6}$$

We can create an augmented matrix from any matrix A and the identity matrix. If we then apply the Gauss-Jordan method to this matrix we transform the augmented matrix into one with the identity matrix on the LHS and the inverse of A on the RHS.

Example:
Calculate the inverse of the matrix

$$\begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix}$$

First we create an augmented matrix

$$\begin{bmatrix} 3 & 2 & 1 & 0 \\ 1 & 4 & 0 & 1 \end{bmatrix}$$

After applying the Gauss-Jordan method to the first row we have

$$\begin{bmatrix} 1 & 2/3 & 1/3 & 0 \\ 0 & -10/3 & 1/3 & -1 \end{bmatrix}$$

We now divide row 2 by $-10/3$ and eliminate element A_{12} to get

$$\begin{bmatrix} 1 & 0 & 12/30 & -1/5 \\ 0 & 1 & -1/10 & 3/10 \end{bmatrix}$$

We now have

$$A^{-1} = \begin{bmatrix} 12/30 & -1/5 \\ -1/10 & 3/10 \end{bmatrix}$$

We can check that this is indeed the inverse by

$$AA^{-1} = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 12/30 & -1/5 \\ -1/10 & 3/10 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{36}{30} - \frac{2}{10} & -\frac{3}{5} + \frac{6}{10} \\ \frac{12}{30} - \frac{4}{10} & -\frac{1}{5} + \frac{12}{10} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Exercise:
Calculate the inverse of the matrix

$$\begin{bmatrix} 3 & 3 & 2 \\ 1 & 0 & 4 \\ 1 & -2 & 2 \end{bmatrix}$$

8

Eigensystems

We now turn our attention to another important matrix equation. Eigenvalues are a special set of scalar values associated with a set of equations like those we have already looked at in the last section. An eigenvalue equation is

$$A\mathbf{v_n} = \lambda \mathbf{v_n} \tag{8.0.1}$$

Where we call λ the eigenvalues of A and $\mathbf{v_n}$ the eigenvectors corresponding to the eigenvalues.

For example, given the matrix

$$A = \begin{bmatrix} 0 & 1 & -1 \\ 1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

We can check that

$$A \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & -1 \\ 1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ -2 \\ 2 \end{bmatrix} = 2 \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}$$

and so 2 is an eigenvalue of A and we have found its corresponding eigenvector $\begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix}$ The other eigenvalues are $1, -1$ with eigenvectors $\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$ $\begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix}$ respectively.

A typical normalised eigenvector ($\mathbf{v_n}/|\mathbf{v_n}|$) has the following orthogonal properties:

$$\mathbf{v_n^T v_m} = \delta_{\mathbf{nm}} \text{ i.e. } \mathbf{0} \text{ if } \mathbf{n} \neq \mathbf{m}$$

and

$$\mathbf{v_n^T A v_m} = \lambda_{\mathbf{n}} \delta_{\mathbf{nm}}$$

You should test these conditions for the eigenvectors we have just seen.

8.1 Similarity Transformations

We will consider real symmetric matrices, i.e. matrices that contain only real values and elements $M_{ij} = M_{ji}$. The $n \times n$matrix will have n eigenvalues which we will label $\lambda_1 \ldots \lambda_n$ which may or may not be distinct. The algorithms for finding the eigenvalues (and the eigenvectors) of a matrix, M, is to perform a series of similarity transformations to M that reduce it to either a diagonal matrix or a tri-diagonal matrix. (A diagonal matrix is one where the only non-zero elements are on the diagonal M_{ii}, a tri-diagonal matrix has non-zero elements along the diagonal and on either side of the diagonal, $M_{i,i\pm1}$).

We call a matrix A a similarity transform of M if

$$A = U^{\mathrm{T}} M U \tag{8.1.1}$$

where $U^{\mathrm{T}}U = U^{-1}U = I$, here I is the identity matrix. The importance of a similarity transformation is that the resulting matrix, A, has the same eigenvalues as the original matrix M. In general, the eigenvectors are different. This is easy to show, if we have $Mx = \lambda x$ and $A = U^{\mathrm{T}}MU$ we multiply the eigenvalue equation on the left by U^{T} and insert $U^{\mathrm{T}}U$ between M and x, we get

$$(U^{\mathrm{T}}MU)(U^{\mathrm{T}}x) = \lambda U^{\mathrm{T}}x$$

which, from our definition of A is

$$A(U^{\mathrm{T}}x) = \lambda U^{\mathrm{T}}x$$

So λ is also an eigenvalue of A and the eigenvectors are given by $U^{\mathrm{T}}x$.

Our procedure for calculating the eigenvalues of M is now to apply N similarity transformations to M so that

$$U_N^{\mathrm{T}} \ldots U_1^{\mathrm{T}} M U_1 U_2 \ldots U_N = D \tag{8.1.2}$$

Another method that is often used is to apply similarity transforms to M to reduce it to a tri-diagonal matrix. There are several methods available to calculate the eigenvalues of a tri-diagonal matrix.

8.2 Jacobi Method

Here we consider the $(n \times n)$ orthogonal matrix

$$\mathbf{P}_{\rm pq} = \begin{bmatrix} 1 & 0 & \dots & 0 & \dots & 0 & \dots & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 & \dots & 0 & \dots & 0 & \dots & 0 \\ 0 & 0 & \ddots & 0 & \dots & 0 & \dots & 0 & \dots & 0 \\ 0 & 0 & \dots & \cos(\theta) & \dots & 0 & \dots & \sin(\theta) & \dots & 0 \\ 0 & 0 & \dots & 0 & \ddots & 0 & \dots & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & \dots & 1 & \dots & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & \dots & 0 & \ddots & 0 & \dots & 0 \\ 0 & 0 & \dots & -\sin(\theta) & \dots & 0 & \dots & \cos(\theta) & \dots & 0 \\ 0 & 0 & \dots & 0 & \dots & 0 & \dots & 0 & \ddots & 0 \\ 0 & 0 & \dots & 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix}$$

Since P is orthogonal, $P^{-1} = P^T$, and so $P^T MP$ is a similarity transformation. We can now apply this transform until the matrix M is diagonalised.

The matrix P describes a plane rotation about an angle θ in n-dimensional Euclidean space, i.e. P is a unit matrix except for the following elements

$$\begin{aligned} P_{\rm pp} &= \cos(\theta) \\ P_{\rm qq} &= \cos(\theta) \\ P_{\rm pq} &= \sin(\theta) \\ P_{\rm qp} &= -\sin(\theta) \end{aligned} \tag{8.2.1}$$

The choice of θ is arbitrary.

If we apply the similarity transformation we can see that $P^{\rm T}_{\rm pq}.M$ only changes rows p and q of M, while $M.P_{\rm pq}$ only changes the columns p and q, so after one transformation M looks like

$$M' = \begin{bmatrix} & \dots & m'_{\rm 1p} & \dots & m'_{\rm 1q} & \dots & \\ \vdots & & \vdots & & \vdots & & \vdots \\ m'_{\rm p1} & \dots & m'_{\rm pp} & \dots & m'_{\rm pq} & \dots & m'_{\rm pn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ m'_{\rm q1} & \dots & m'_{\rm qp} & \dots & m'_{\rm qq} & \dots & m'_{\rm qn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ & \dots & m'_{\rm np} & \dots & m'_{\rm nq} & \dots & \end{bmatrix}$$

Exercise:
Do the matrix multiplication of $M' = P^{\rm T} MP$ and show that the values of M' are

$$\begin{aligned} m'_{\rm ip} &= \cos(\theta) m_{\rm ip} - \sin(\theta) m_{\rm iq}, \text{ for } i \neq p, i \neq q \\ m'_{\rm iq} &= \cos(\theta) m_{\rm iq} + \sin(\theta) m_{\rm ip}, \text{ for } i \neq p, i \neq q \\ m'_{\rm pp} &= \cos^2(\theta) m_{\rm pp} + \sin^2(\theta) m_{\rm pp} - 2\sin(\theta)\cos(\theta) m_{\rm pp} \\ m'_{\rm qq} &= \sin^2(\theta) m_{\rm qq} + \cos^2(\theta) m_{\rm qq} + 2\sin(\theta)\cos(\theta) m_{\rm qq} \\ m'_{\rm pq} &= \left(\cos^2(\theta) - \sin^2(\theta)\right) m_{\rm pq} + \sin(\theta)\cos(\theta)\left(m_{\rm pp} - m_{\rm qq}\right) \end{aligned} \tag{8.2.2}$$

The idea behind the Jacobi method is to zero the off diagonal elements for each rotation (similarity transformation corresponding to a choice of p and q). Setting the element $m'_{\rm pq} = 0$ in (8.2.2) we have

$$\left(\cos^2(\theta) - \sin^2(\theta)\right) m_{\rm pq} + \sin(\theta)\cos(\theta)\left(m_{\rm pp} - m_{\rm qq}\right) = 0$$

which we can rewrite as

$$\frac{m_{\rm qq} - m_{\rm pp}}{m_{\rm pq}} = \frac{\cos^2(\theta) - \sin^2(\theta)}{\sin(\theta)\cos(\theta)}$$

from which we derive

$$\tan(2\theta) = \frac{2m^{(k)}_{\rm pq}}{m^{(k)}_{\rm pp} - m^{(k)}_{\rm qq}}$$

If the denominator, $m^{(k)}_{\rm pp} - m^{(k)}_{\rm qq}$, is zero we choose

$$\theta = \frac{\pi}{4}\frac{m^{(k)}_{\rm pq}}{|m^{(k)}_{\rm pq}|}$$

Having a value for θ we can easily calculate the other trigonometric values

$$\begin{aligned} \cos(\theta) &= \sqrt{\frac{1}{2}\left(1 + \frac{1}{\sqrt{1 - \tan^2(2\theta)}}\right)} \\ \sin(\theta) &= \sqrt{\frac{1}{2}\left(1 - \frac{1}{\sqrt{1 - \tan^2(2\theta)}}\right)} \end{aligned}$$

The solution proceeds as follows:

1. We perform a number of iterations on the matrix M until M is diagonalised. The eigenvalues are the diagonal elements of M. In practice, it is difficult to get all the off-diagonal elements to zero (see the section in the first chapter about precision), instead we keep iterating until the off-diagonal elements are less than a predetermined tolerance.

2. For each iteration we perform a number of similarity transformations to M.

3. The angle θ is calculated at each $p = 1..N, q = p+1..N$, the matrix P is thus calculated at each (pq) and the transformation can be applied.

According to 'Numerical Recipes' this method is guaranteed to work for all real symmetric matrices, though it may not be the quickest. Reducing the matrix to a tri-diagonal form and then applying another algorithm is usually quicker for large matrices. Reduction to a tridiagonal matrix always requires a finite number of steps, the Jacobi method requires iteration until a desired precision is met.

8.3 Power Method

We will now look at a method for finding the eigenvalue of a matrix, M, with the largest magnitude and the associated eigenvectors. We will label the eigenvalues λ_i such that

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n| \tag{8.3.1}$$

and label the eigenvectors associated with λ_1 as v_1. In this notation, λ_1 is referred to as the dominant eigenvalue.

If $M_{n\times n}$ has n linearly independent eigenvectors and the eigenvalues can be written as the inequality (8.3.1) we can use the power method to find the eigenvalue with the largest magnitude.

If x is any vector then it can be written

$$x = c_1v_1 + c_2v_2 + c_3v_3 + \ldots + c_nv_n$$

where $\{v_1, v_2, \ldots, v_n\}$ is a set of linearly independent eigenvectors. Thus

$$\begin{aligned} Mx &= c_1\lambda_1v_1 + c_2\lambda_2v_2 + c_3\lambda_3v_3 + \ldots + c_n\lambda_nv_n \\ M^2x &= c_1\lambda_1^2v_1 + c_2\lambda_2^2v_2 + c_3\lambda_3^2v_3 + \ldots + c_n\lambda_n^2v_n \\ M^3x &= c_1\lambda_1^3v_1 + c_2\lambda_2^3v_2 + c_3\lambda_3^3v_3 + \ldots + c_n\lambda_n^3v_n \\ &= \ldots \\ M^mx &= c_1\lambda_1^mv_1 + c_2\lambda_2^mv_2 + c_3\lambda_3^mv_3 + \ldots + c_n\lambda_n^mv_n \end{aligned}$$

Dividing the last equation by λ_1^m we have

$$\frac{1}{\lambda_1^m}M^mx = c_1v_1 + c_2\left(\frac{\lambda_2}{\lambda_1}\right)^m v_2 + c_3\left(\frac{\lambda_3}{\lambda_1}\right)^m v_3 + \ldots + c_n\left(\frac{\lambda_n}{\lambda_1}\right)^m v_n$$

But, since $\left(\frac{\lambda_2}{\lambda_1}\right)^m, \ldots, \left(\frac{\lambda_n}{\lambda_1}\right)^m$ get progressively closer to zero as $m \to \infty$ we can write,

$$\frac{1}{\lambda_1^m}M^mx = c_1v_1$$

if $c_1 \neq 0$. A way of ensuring this is for x to not be orthogonal to v_1. We now note that we can also write (since $m+1 > m$)

$$\frac{1}{\lambda_1^{m+1}}M^{m+1}x = c_1v_1$$

If we take the dot product of both sides of the last two expressions with any vector U that is not orthogonal to v_1 we get

$$\begin{aligned} \frac{1}{\lambda_1^m}M^mx \cdot U &= c_1v_1 \cdot U \\ \frac{1}{\lambda_1^{m+1}}M^{m+1}x \cdot U &= c_1v_1 \cdot U \end{aligned}$$

giving

$$\frac{1}{\lambda_1^{m+1}}M^{m+1}x \cdot U = \frac{1}{\lambda_1^m}M^mx \cdot U \neq 0$$

Leaving us with an expression for the largest (magnitude) eigenvalue.

$$\lambda_1 = \frac{\lambda_1^{m+1}}{\lambda_1^m} = \frac{M^{m+1}x \cdot U}{M^mx \cdot U}$$

So how do we choose U? We know that for any eigenpair (λ, v) of M

$$Mv = \lambda v$$

We can take the dot product of both sides with v to obtain

$$\frac{Mv.v}{v.v} = \lambda$$

If we choose $U = M^mx$ then we find that

$$\lambda_1 = \frac{M^{m+1}x \cdot M^mx}{M^mx \cdot M^mx} = \frac{MM^mx \cdot M^mx}{M^mx \cdot M^mx} = \frac{Mv \cdot v}{v \cdot v} \tag{8.3.2}$$

Which gives us the power method for finding the eigenvalue of largest magnitude of a matrix M. As well as the largest eigenvalue we can also write the eigenvectors associated with the largest eigenvalue as M^mx.

An important question we need to address at this point is: At what point do we stop doing the iteration in the power method? Normally we would say "stop when the error is lower than some required accuracy" i.e.

$$|\lambda_1^{\text{calc}} - \lambda_1^{\text{actual}}| < \epsilon$$

But we don't know $\lambda_1^{\text{actual}}$. However, we can use a useful theorem here that states:

If M is a real symmetric matrix with dominant eigenvalue λ_1, then if $\lambda_1^{\text{calc}} = (MX \cdot X)/(X \cdot X)$, where $X = M^mX_0$, as in the power method, then:

$$|\lambda_1^{\text{calc}} - \lambda_1^{\text{actual}}| \leq \sqrt{\frac{MX \cdot MX}{X \cdot X} - \left(\lambda_1^{\text{calc}}\right)^2} \tag{8.3.3}$$

We should note from this theorem that the error estimate is a maximum estimate. That is to say the maximum error is given by (8.3.3). We can also look a the convergence of the eigenvalue, although this is not totally reliable. The relative error between steps is

$$E_n = \frac{|\lambda_1^{\text{calc}_n} - \lambda_1^{\text{calc}_{n-1}}|}{|\lambda_1^{\text{calc}_{n-1}}|} \tag{8.3.4}$$

and we should stop when this error is relatively small. Since our theorem for the error in the power method only works for real symmetric matrices, equation 8.3.3 will not apply to non-symmetric matrices. In this case case the relative error 8.3.4 must be used.

Example:
Find the eigenvalue with largest magnitude of

$$\begin{pmatrix} 1 & 5 \\ 5 & 4 \end{pmatrix}$$

First we choose $x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and calculate

$$\begin{aligned} Mx &= \begin{pmatrix} 1 & 5 \\ 5 & 4 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 6 \\ 9 \end{pmatrix} \\ M^2x &= \begin{pmatrix} 1 & 5 \\ 5 & 4 \end{pmatrix}^2\begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 26 & 25 \\ 25 & 41 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 51 \\ 66 \end{pmatrix} \\ M^3x &= \begin{pmatrix} 1 & 5 \\ 5 & 4 \end{pmatrix}^3\begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 151 & 230 \\ 230 & 289 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 381 \\ 519 \end{pmatrix} \end{aligned}$$

So,

$$\lambda_1 = \frac{(M^3x)\cdot(M^2x)}{(M^2x)\cdot(M^2x)} = \frac{(381\times 51)+(519\times 66)}{(51\times 51)+(66\times 66)} \approx 7.7167$$

The actual eigenvalues are $\frac{5}{2} \pm \frac{sqrt109}{2} \approx 7.720, -2.720$
We can now also calculate the eigenvectors for the $\lambda_1 = 7.7167$ eigenvalue.
$v_1 = m^2x = \begin{pmatrix} 51 \\ 66 \end{pmatrix}$.
We can calculate the error (which we expect to be large since we have only used a few steps) using (8.3.4). We will label

$$\lambda^n = \frac{(M^nx)\cdot(M^{n-1}x)}{(M^{n-1}x)\cdot(M^{n-1}x)}$$

Using this notation we have already calculated $\lambda^3 = 7.7167$. It is left as an exercise to show that $\lambda^2 = 7.6923$. The relative error is thus

$$E_n = 0.0032$$

The power method can obtain the largest eigenvalue of non-symmetric matrices, but there are situations where it will not work.

1. If the matrix M is not diagonalisable the power method will fail. Unfortunately it is not easy to tell beforehand if M is actually diagonalisable.
2. If M does not have a dominant eigenvalue or if $\lambda_1| = |\lambda_2|$. In this case $|\lambda_2/\lambda_1|$ is either barely less than one or worse equal to one. This means that the high powers of $|\lambda_i/\lambda_2|$ does not tend to zero.
3. If the entries to M contain significant errors the errors will become significant in the powers of M.

In general, when using the power method you should,

1. Try the power method, if the values for $\frac{M^{m+1}x\cdot M^mx}{M^mx\cdot M^mx}$ approach a single value λ_1 then
2. Check if λ_1 and the $v_1 = M^mx$ form an eigenpair by
$$M(M^mx) = \lambda_1(M^mx)$$
3. if step 2 is OK, then accept (λ_1, v_1) as an eigenpair.

Exercise:
Find the eigenvalues of the following matrices

$$\begin{pmatrix} 3 & 4 \\ 1 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 5 & 4 \\ 5 & 4 & 2 \\ 3 & 2 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 4 \\ 2 & 2 \end{pmatrix}$$

8.3.1 Power Method with Scaling

The power method can generate vectors with large components. To avoid this we can multiply the vector

$$X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \text{ by } \frac{1}{\max\{|x_1|, |x_2|, \ldots, |x_n|\}}$$

at each step. This is called the scaling of X. As an example, the scaling of

$$\begin{pmatrix} 3 \\ -4 \\ 4 \end{pmatrix} \text{ is } \frac{1}{4}\begin{pmatrix} 3 \\ -4 \\ 4 \end{pmatrix} = \begin{pmatrix} \frac{3}{4} \\ -1 \\ 1 \end{pmatrix}$$

The power method now proceeds as follows:

1. Calculate MX_0. Let v_1 be the scaled version of MX_0.
2. Calculate Mv_1. Let v_2 be the scaled version of Mv_1.
3. Calculate Mv_2. Let v_3 be the scaled version of Mv_2.

Continue in this way. At a step m we then have

$$\lambda_1 = \frac{M_{m-1} \cdot v_{m-1}}{v_{m-1} \cdot v_{m-1}}$$

and v_m is the associated eigenvector.

8.4 Deflation

Sometimes we may wish to calculate the second largest (and subsequent) eigenvalues of a matrix once the dominant eigenvalue and eigenvectors are found. If M is a symmetric matrix with v_1 being the eigenvectors associated with the largest eigenvalue λ_1 then

$$\mathcal{M} = M - \lambda_1 U_1 U_1^T \tag{8.4.1}$$

where $U_1 = v_1/|v_1|$. $\mathcal{M}$ has eigenvalues $0, \lambda_2, \lambda_3, \ldots, \lambda_n$ and the same eigenvectors as M so we could apply the power method to $\mathcal{M}$ to find them. This method is called *deflation*.

Since λ_1 is not exact then applying the power method to $\mathcal{M}$ will propagate the error through λ_2 and so on. Successively applying the power method is also not the most efficient method of finding all the eigenvalues of a matrix.

Example:

We have already found $\lambda_1 = 7.7167$ and $v_1 = M^2x = \begin{matrix} 51 \\ 66 \end{matrix}$ for the matrix $M = \begin{pmatrix} 1 & 5 \\ 5 & 4 \end{pmatrix}$

We can also find the second eigenvalue using deflation. Since the eigenvectors of the largest eigenvalue are $v_1 = \begin{matrix} 51 \\ 66 \end{matrix}$ we can write

$$\begin{aligned} llU &= \begin{matrix} 0.61 \\ 0.79 \end{matrix} \\ U^{\mathrm{T}} &= \begin{matrix} 0.61 & 0.79 \end{matrix} \end{aligned} \tag{8.4.2}$$

Using (8.4.1) we have

$$\begin{aligned} \mathcal{M} &= M - \lambda_1 U_1 U_1^T \\ &= \begin{pmatrix} 1 & 5 \\ 5 & 4 \end{pmatrix} - 7.7167 \begin{pmatrix} 0.3721 & 0.4819 \\ 0.4819 & 0.6421 \end{pmatrix} \\ &= \begin{pmatrix} -1.6668 & 1.5462 \\ 1.5462 & -0.60193 \end{pmatrix} \end{aligned}$$

We now use the power method to find the largest eigenvalue of $\mathcal{M}$.

$$\begin{aligned} \mathcal{M}x &= \begin{pmatrix} -0.1206 \\ 0.94427 \end{pmatrix} \\ \mathcal{M}^2x &= \begin{pmatrix} 1.661 \\ -0.7549 \end{pmatrix} \\ \mathcal{M}^3x &= \begin{pmatrix} -3.9358 \\ 3.0227 \end{pmatrix} \end{aligned} \tag{8.4.3}$$

We now have

$$\lambda_2 = \frac{\begin{pmatrix} -3.9358 \\ 3.0227 \end{pmatrix} \cdot \begin{pmatrix} 1.661 \\ -0.7549 \end{pmatrix}}{\begin{pmatrix} 1.661 \\ -0.7549 \end{pmatrix} \cdot \begin{pmatrix} 1.661 \\ -0.7549 \end{pmatrix}} = \frac{-8.8192}{3.3288} = -2.6494$$

We have (from our previous example) the actual value of the eigenvalue as -2.7201. Since we only did three steps in the power series it is not surprising that the error in the eigenvalue is large. It is left to the reader to calculate the error using $\mathcal{M}x, \mathcal{M}^2x, \mathcal{M}^3x$.

8.5 Lanczos Method

9

Fourier Series

The superposition principle states that the net displacement at a point caused by two or more waves traversing the point is the vector sum of the displacements which would have been produced by the waves seperately.

This principle also works in reverse. If we let $f(t)$ be the air pressure close to a vibrating string (which we will assume is emitting a musical note) then we can expect $f(t)$ to be the sum of a number of harmonic functions of time, say $\cos(\omega t)$, for each of the harmonic frequencies i.e. $f(t) = \sum \cos(n\omega t)$. However, we could expect each frequency to have a unique phase so our function $f(t) = \sum \cos(n\omega t + \phi)$.

We can rewrite

$$\cos(\omega t + \phi) = \cos(\phi)\cos(\omega t) - \sin(\phi)\sin(\omega t)$$

and since ϕ is a constant we can write *any* sinusoidal oscillation with a frequency ω as

$$\begin{aligned} f(t) = \; & a_0 \\ & + \; a_1\cos(\omega t) + b_1\sin(\omega t) \\ & + \; a_2\cos(\omega t) + b_2\sin(\omega t) \\ & + \; \ldots + ldots \end{aligned}$$

or

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left[a_n\cos\left(n\omega t\right) + b_n\sin\left(n\omega t\right)\right] \tag{9.0.1}$$

or equivalently, in terms of a spacial function,

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left[a_n\cos\left(\frac{2\pi nt}{L}\right) + b_n\sin\left(\frac{2\pi nt}{L}\right)\right] \tag{9.0.2}$$

This is known as the Fourier series after Joesph Fourier who discovered a method to calculate the coefficients. We will continue our discussion of the Fourier series in terms of a spacial function. The discussion is equally valid in terms of a teime function.

If we take our expression for the Fourier series (9.0.2), and multiply by $\cos(2\pi px/L)$ and integrate over one full period in x, we get

$$\begin{aligned} \int_{x_0}^{x_0+L} f(x)\cos\left(\frac{2\pi px}{L}\right)dx \; = \; & \frac{a_0}{2}\int_{x_0}^{x_0+L}\cos\left(\frac{2\pi px}{L}\right)dx \\ & + \; \sum_{n=1}^{\infty} a_n\int_{x_0}^{x_0+L}\cos\left(\frac{2\pi nx}{L}\right)\cos\left(\frac{2\pi px}{L}\right)dx \\ & + \; \sum_{n=1}^{\infty} b_n\int_{x_0}^{x_0+L}\sin\left(\frac{2\pi nx}{L}\right)\cos\left(\frac{2\pi px}{L}\right)dx \end{aligned}$$

We can now find the values of a_n, b_n by considering this expression for different values of p. We should note the following orthogonality properties:

$$\begin{aligned} \int_{x_0}^{x_0+L}\sin\left(\frac{2\pi nx}{L}\right)\cos\left(\frac{2\pi px}{L}\right)dx \; &= \; 0 \text{ for all n p} \\ \int_{x_0}^{x_0+L}\cos\left(\frac{2\pi nx}{L}\right)\cos\left(\frac{2\pi px}{L}\right)dx \; &= \; \begin{cases} L & \text{for n = p = 0,} \\ L/2 & \text{for n = p > 0,} \\ 0 & \text{for n} \neq \text{p} \end{cases} \\ \int_{x_0}^{x_0+L}\sin\left(\frac{2\pi nx}{L}\right)\sin\left(\frac{2\pi px}{L}\right)dx \; &= \; \begin{cases} 0 & \text{for n = p = 0,} \\ L/2 & \text{for n = p > 0,} \\ 0 & \text{for n} \neq \text{p} \end{cases} \end{aligned} \tag{9.0.3}$$

We can see that when $p = 0$, 9.0.3 becomes

$$\int_{x_0}^{x_0+L} f(x)dx = \frac{a_0}{L}$$

and when $p \neq 0$ the only term on the RHS of 9.0.3 that is non-zero is

$$\int_{x_0}^{x_0+L} f(x)\cos\left(\frac{2\pi nx}{L}\right)dx = \frac{a_n}{L}$$

Exercise:
Find the b_n coefficient by replacing $\cos\left(\frac{2\pi nx}{L}\right)$ with $\sin\left(\frac{2\pi nx}{L}\right)$ in the above process.

The Fourier series of a periodic, single valued function, $f(x)$, with no infinities is expressed as

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left[a_n \cos\left(\frac{2\pi nt}{L}\right) + b_n \sin\left(\frac{2\pi nt}{L}\right)\right] \tag{9.0.4}$$

where the coefficients are

$$a_0 = \frac{2}{L}\int_{x_0}^{x_0+L} f(x)dx$$
$$a_n = \frac{2}{L}\int_{x_0}^{x_0+L} f(x)\cos\left(\frac{2\pi nx}{L}\right)dx$$
$$b_n = \frac{2}{L}\int_{x_0}^{x_0+L} f(x)\sin\left(\frac{2\pi nx}{L}\right)dx$$

Thus, any periodic, integrable function can be written as a sum of sine and cosine terms.

Example:
Calculate the Fourier coefficients of the step function

$$f(x) = \begin{cases} -1, & -\frac{T}{2} < x < 0 \\ 1, & 0 < x < \frac{T}{2} \end{cases}$$

Since our function is odd we will have only sine terms in our Fourier series. Knowing this we could skip the calculation of the a_n coefficients but for clarity we will do it here.

$$\begin{aligned}
a_n &= \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(x)\cos\big(\frac{2\pi nx}{T}dx \\
&= \frac{2}{T}\left[\int_{-\frac{T}{2}}^{0}(-1)\cos\left(\frac{2\pi nx}{T}\right)dx + \int_{0}^{\frac{T}{2}}(+1)\cos\left(\frac{2\pi nx}{T}\right)dx\right. \\
&= \frac{2}{T}\left[-\frac{T}{2\pi n}\sin\left(\frac{2\pi nx}{T}\right)\Big|_{-\frac{T}{2}}^{0} + \frac{T}{2\pi n}\sin\left(\frac{2\pi nx}{T}\right)\Big|_{0}^{\frac{T}{2}}\right] \\
&= \frac{1}{n\pi}\left[-\sin\left(\frac{2\pi n0}{T}\right) + \sin\big(-n\pi\big) + \sin\big(n\pi\big) - \sin\left(\frac{2\pi n0}{T}\right)\right] \\
&= \frac{2}{n\pi}\left[\sin\big(n\pi\big)\right] \\
&= 0
\end{aligned}$$

since n is an integer.

$$\begin{aligned}
b_n &= \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(x)\sin\big(\frac{2\pi nx}{T}dx \\
&= \frac{2}{T}\left[\int_{-\frac{T}{2}}^{0}(-1)\sin\left(\frac{2\pi nx}{T}\right)dx + \int_{0}^{\frac{T}{2}}(+1)\sin\left(\frac{2\pi nx}{T}\right)dx\right. \\
&= \frac{2}{T}\left[\frac{T}{2\pi n}\cos\left(\frac{2\pi nx}{T}\right)\Big|_{-\frac{T}{2}}^{0} - \frac{T}{2\pi n}\cos\left(\frac{2\pi nx}{T}\right)\Big|_{0}^{\frac{T}{2}}\right] \\
&= \frac{1}{n\pi}\left[\cos\left(\frac{2\pi n0}{T}\right) - \cos\big(-n\pi\big) - \cos\big(n\pi\big) + \cos\left(\frac{2\pi n0}{T}\right)\right] \\
&= \frac{2}{n\pi}\left[1 - \cos\big(n\pi\big)\right] \\
&= \frac{2}{n\pi}\left[1 - (-1)^n\right]
\end{aligned}$$

so $b_n = 4/\pi$ if n is odd, otherwise $b_n = 0$.
So the Fourier series is

$$f(x) = \frac{4}{\pi}\Big(\sin(\omega x) + \frac{\sin(3\omega x)}{3} + \frac{\sin(5\omega x)}{5} + \dots\Big)$$

where the angular frequency $\omega = 2\pi/T$. The first 4 terms of this series is plotted in fig 9.1 for $\omega = 1$.

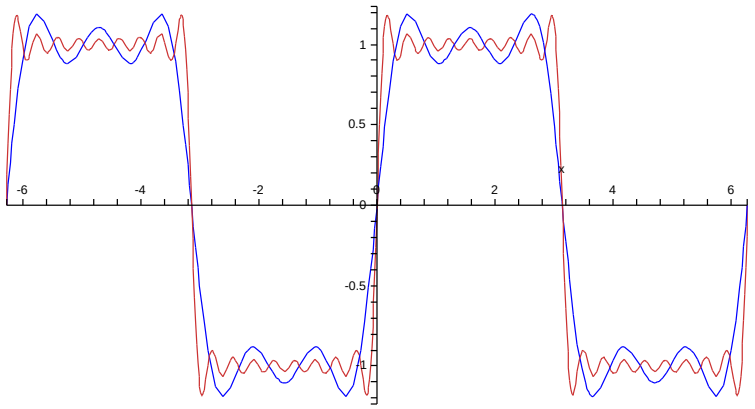


Figure 9.1: Fourier series representation of the step function using 4 and 9 terms

There is always an overshoot of about $5-15\%$ around the points where the function has a discontinuity. This is known as the Gibbs phenomenon and can clearly be seen in fig 9.1. Adding more terms to the series will not remove the overshoot.

Properties of Fourier Series

9.1 Non-periodic Functions

9.2 Fourier Transforms

$$\tilde{f}(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{i\omega t} dt \tag{9.2.1}$$

$$f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \tilde{f}(\omega) e^{i\omega t} d\omega \tag{9.2.2}$$

Example:
Find the Fourier Transform of a single slit, given by

$$f(x) = \begin{cases} 1, & -T \leq x\pi \\ 0, & \text{otherwise} \end{cases}$$

$$\begin{aligned} FT[f(x)] &= \int_{-\infty}^{\infty} f(x) e^{-i\omega t} \mathrm{dt} \\ &= \int_{-T}^{T} f(x) e^{-i\omega t} \mathrm{dt} \\ &= \left. \frac{e^{-i\omega t}}{i\omega} \right|_{-T}^{T} \\ &= \frac{e^{i\omega T} - e^{-i\omega T}}{i\omega} \end{aligned}$$

9.3 Optics

Index