Summer Of Science 2020

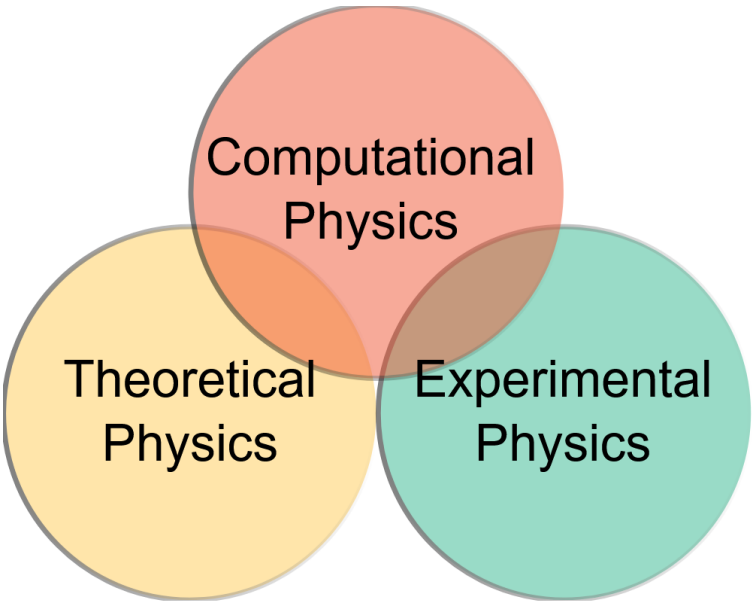
Computational Physics

Janak Ruia

Roll number: 190260038

April 2020

Mentor: Chaitanya Kumar



Contents

Chapter 1

Basics of Python

1.1 Introduction

I have used Python 3.8 to implement all the logic mentioned in this report.

1.2 Python programming

Being able to create and assign values to variables, output and input data, perform arithmetic, define functions, write comments, call built-in functions, control the program with "if" and "while" statements, working with arrays and lists (including reading arrays from a file and array slicing) and using "for" loops.

1.3 Importing Modules

I have used various modules included in python throughout this project, some of which were:

- Numpy
- Scipy
- math
- pylab

Chapter 2

Graphs in Python

2.1 Introduction

I have used the pylab package to create graphs in python. I have learned to create three major types of graphs:

- Line graphs
- Scatter plots
- Density plots

2.2 Line Graphs

These are the most basic graphs. The plot() function is used to plot line graphs in python, while the show() function is used to display the graphs plotted before the show() function is called.

2.2.1 Example:

The Fey's Function is defined as:

$$r = e^{\cos\theta} - 2\cos 4\theta + \sin^5 \frac{\theta}{12}$$

The code to plot it in python is:

```
from numpy import *
from pylab import *
theta=linspace(0,10*pi,3000)
r=exp(cos(theta))-2*cos(4*theta)+(sin(theta/12))**5
x=r*cos(theta)
y=r*sin(theta)
plot(x,y,"k-")
show()
```

The * in 1^{st} and 2^{nd} line tells the compiler to import all the functions and definitions from the respective packages. Here is the output graph of the code:

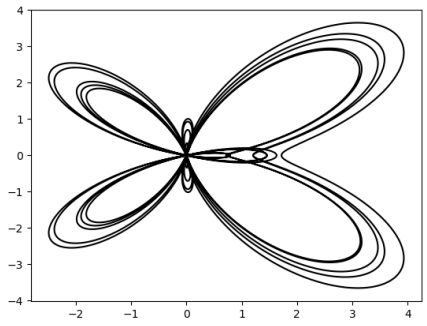


Figure 2.1: Polar plot of Fey's Function

2.3 Scatter Plots

Scatter Plots are primarily used when the variables on both the axes are dependent variables. Many situations like these arise in physics, eg. Temperature vs brightness of stars (HR diagrams)

2.3.1 HR Diagram:

The file stars.txt contains the magnitudes and surface temperatures of a set of stars. The code to plot the scatter HR diagram is:

```
from numpy import *
from pylab import *
data=loadtxt("stars.txt",float)
x=data[:,0]
y=data[:,1]
scatter(x,y)
xlabel("Temperature")
ylabel("Magnitude")
xlim(13000,0)
ylim(20,-5)
show()
```

And the corresponding plot is :

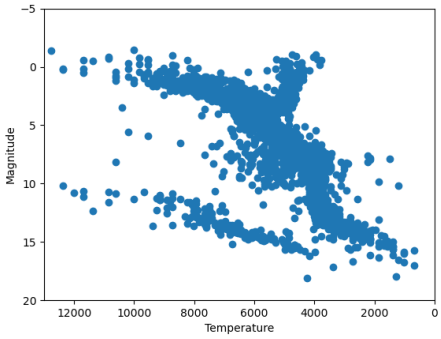


Figure 2.2: HR Diagram

2.4 Density Plots

These plots are used to work with two dimensional data. Many situations arise in physics when we need to record and plot two dimensional data, eg. atomic deposition on a solid surface, height of waves in a ripple tank, distribution of particles incident on an imaging detector, etc. The function imshow(data) is used in python to plot the two dimensional data. Here the argument "data" is an array containing two columns, corresponding to the recorded data. The function assigns each point in the data, a color, and thus produces a plot "heat plot" for the data.

2.4.1 Wave Interference

Here is a piece of code, which simulates the interference pattern between two circular waves in pond: If the pond is my coordinate plane, and (x_1, y_1) and (x_2, y_2) are the points of origin of the waves, the assuming the waves to be circular,

$$r_1 = \sqrt{(x - x_1)^2 + (y - y_1)^2}$$
$$r_2 = \sqrt{(x - x_2)^2 + (y - y_2)^2}$$

$$\psi_1 = A \sin kr_1$$

$$\psi_2 = A \sin kr_2$$

By Superposition:

$$\Psi = \psi_1 + \psi_2$$

```
from pylab import *
from numpy import *
from math import *
wavelength=5.0
k=2*pi/wavelength
A=1.0
sep=20.0
points=1000
side=100.0
spacing=side/points
x1=side/2+sep/2
y1=side/2
x2=side/2-sep/2
y2=side/2
final=empty([points,points],float)
for i in range(points):
        y=spacing*i
        for j in range(points):
                x=spacing*j
                r1=sqrt((x-x1)**2+(y-y1)**2)
                r2=sqrt((x-x2)**2+(y-y2)**2)
                final[i,j]=A*(sin(k*r1)+sin(k*r2))
imshow(final,origin="lower",extent=[0,side,0,side])
gray()
show()
```

Here, I have divided the plane into a lattice of points, and assigned each point a "height" based on the calculations.The output is shown in Figure 2.3:

2.4.2 Silicon Lattice

The file "stem.txt" contains data of the result of scanning tunnel microscopy of a silicon crystal.

```
from pylab import *
from numpy import *
stm=loadtxt("stm.txt",float)
imshow(stm, origin="lower",extent=[0,1000,0,1000])
gray()
show()
```

The output is shown in figure 2.4

2.4.3 Deterministic Chaos and the "Figtree plot"

One of the most famous phenomenon of chaos is the logistic equation:

$$x^{'} = rx(1-x)$$

For a particular value of r, we choose an initial x (0.5 in our case), feed it to the RHS, and take the $x^{'}$ and again feed it to the RHS. Repeating this process causes one of the three things to happen:

- The value settles down to a fixed point.
- The value settles down to periodic pattern - Limit Cycle
- All hell breaks loose - This is formally called "Deterministic Chaos". Deterministic because we can actually calculate every value, but no pattern emerges.

Here is a program to plot the famous Feigenbbaum plot, using x=0.5, and varying r over 1 to 4.

```
from pylab import *

def logmap(r,n):
        x=0.5
        #By using equation $x <- rx(1-x)$ n times the value is returned

        for i in range(n):
                x = r*x*(1-x)
        return x
n=1000
while n<1100:
        r = arange(1,4,0.01)
        x = logmap(r,n)
        plot(r,x,'k.')
        n+=1

xlabel('r')
ylabel('x')
show()
```

We first iterate x 1000 times for every r, so that it reaches one the three stages. Then we plot those values and repeat the process for 1001 iterations and so on till we reach 1100 iterations. Now the x's which reached a fixed point don't change. x's which reach their limit cycle (hopefully) complete their respective cycles atleast one time, and x's which become chaotic give unpredictable values over the range of r. The final plot is shown in fig. 2.5 Notice that if we zoom into one of the primary branches, the pattern repeats itself (fig. 2.6 Somehow, order emerged out of chaos!

2.4.4 Photo-electric effect

During the original experiment done by Millikan, he recorder some values of frequencies vs volts, which are given in the file "millikan.txt" The code calculates the slope and intercept for the data using the least square fitting method:

```
from pylab import *
from numpy import *
data=loadtxt("millikan.txt",float)
x=data[:,0]
y=data[:,1]
N=size(x)
Ex=(1/N)*sum(x)
Ey= (1/N)*sum(y)
Exx=(1/N)*sum(x*x)
Exy=(1/N)*sum(x*y)
m=(Exy-Ex*Ey)/(Exx-Ex**2)
c=(Exx*Ey-Ex*Exy)/(Exx-Ex**2)
final=m*x+c
h=m*1.6*10**-19
print("Plank's constant is: ",h)
plot(x,final,"b-")
plot(x,y,"k.")
xlabel('frequency')
ylabel('volts')
show()
```

And the corresponding plot is show in fig. 2.7

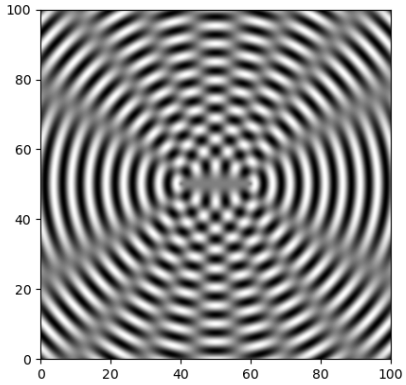


Figure 2.3: Wave Interference

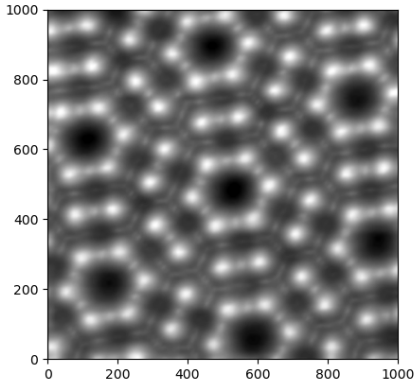


Figure 2.4: Surface of Silicon Lattice

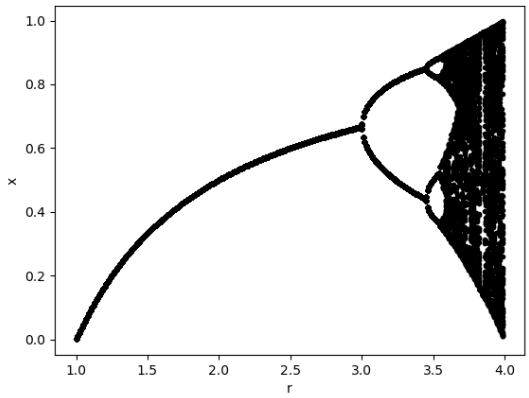


Figure 2.5: The Figtree plot

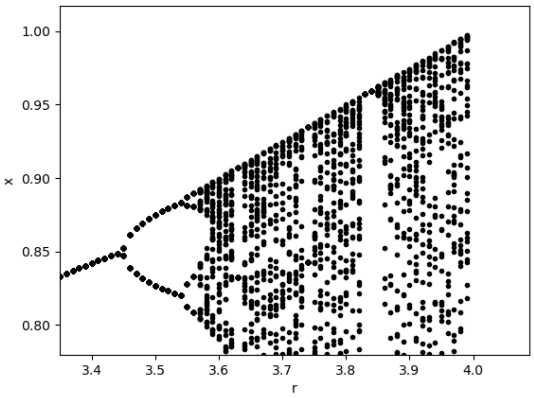


Figure 2.6: Order in chaos!

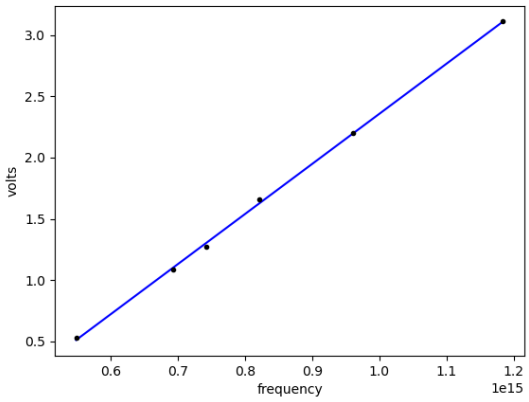


Figure 2.7: Millikan's Plot for calculating Planks constant

Chapter 3

Integrals

3.1 Introduction

Here, we will see various numerical methods to solve integrals. These methods come handy when you can't solve the integral analytically. We will look at various methods, discuss their logic and advantages over others.

- Trapezoid rule
- Simpson's rule
- Adaptive Integration
- Romberg Integration
- Gaussian Quadrature

3.2 Trapezoid Rule

This is most fundamental and basic numerical method in which, we just divide the domain of integration into multiple trapeziums and add up their respective areas. Let

$$I(a,b) = \int_a^b f(x)dx \tag{3.1}$$

Let's divide the domain into N slices of equal width $h = \frac{b-a}{N}$. The right side of k^{th} slice falls at $a+kh$, while it's left side falls at $a+(k-1)h$. The area of this trapezoid is: $A_k = \frac{1}{2} \cdot h \cdot (f(a+(k-1)h) + f(a+kh))$ Now the approximation to the total integration is:

$$\sum_{k=1}^{N} A_k = \frac{1}{2} h \sum_{k=1}^{N} [f(a+(k-1)h) + f(a+kh)]$$

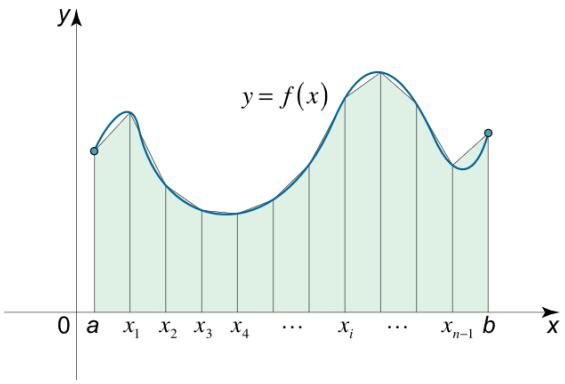


Figure 3.1: Trapezoid Rule. Image courtesy: math24.net

$$= \sum_{k=1}^{N} h \cdot \left[\frac{1}{2}(f(a) + f(b)) + \sum_{k=1}^{N-1} f(a + kh) \right] \tag{3.2}$$

This is also called the extended Trapezoid rule.

3.2.1 Example

Let's integrate a simple function using the rule mentioned above. Let I= $\int_0^2 x^4 - 2x + 1$

The code is:

```
def f(x):
    return x**4-2*x+1
N=10 #no. of slices
a=0.0
b=2.0
h=(b-a)/N
s=0.5*(f(a)+f(b))
for k in range(N):
    s+=f(a+k*h)
print(h*s)
```

The output is: 4.50656 We know that the true answer is 4.4, so the error is nearly of 2 percent. Increasing N gives more and more accurate answer. This rule is ideal in many physics calculation because it can give a reasonable accuracy in a reasonable time. However, if greater accuracy is needed, we can switch to other methods like Simpson's rule.

3.3 Simpson's Rule

Trapezoid rule approximates the curve with straight line segments. In Simpson's Rule, we try and approximate the curve by quadratic curves (it takes three sample points to plot a quadratic) Let the integrand be $f(x)$ with the three

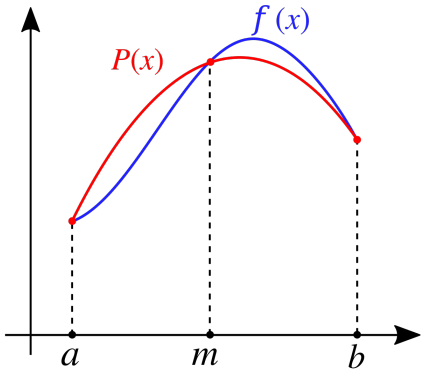


Figure 3.2: Simpson's Method: Image courtesy- Wikipedia

sample points being $-h, 0, h$. If we try to fit a quadratic $Ax^2 + Bx + C$, then we get three linear equations and three unknowns, after solving which we get the result:

$$\int_{-h}^{h} (Ax^2 + Bx + C) \cdot dx = \frac{1}{3}h[f(-h) + 4f(0) + f(h)] \tag{3.3}$$

This is the Simpson's Rule. Notice that the integral depends only on the sample points, so all we have to do is plug in the sample points and the value of h. Also notice that we can slide the curve along the x axis without changing the area under the curve, so this formula is valid for any three evenly spaced points. If we are integrating from a to b, the first pair of slices would be $a, a+h, a+2h$, and the second would be $a+2h, a+3h, a+4h$, and so on. Using the equation mentioned above repeatedly and adding over all the pairs of slices in the domain, we see:

$$I(a,b) \approx \frac{1}{3}h \left[f(a) + f(b) + 4 \cdot \sum_{k-odd(1\ldots N-1)} [f(a+kh)] + 2 \cdot \sum_{k-even(2\ldots N-2)} [f(a+kh)] \right] \tag{3.4}$$

This is also called the extended Simpson's Rule. To see the power of Simpson's Rule, if we integrate the same function : $x^4 - 2x + 1$ form 0 to 2 using Simpson's rule and N=10 again, we get the result: 4.400427 which is accurate to 0.01 percent, orders better than Trapezoid rule, for nearly the same computational time.

3.3.1 Example

Let's try to perform this integral:

$$E(x) = \int_0^x \exp -t^2$$

Let's make a graph of this function vs. x The code is:

```
from pylab import plot,show
from numpy import *
def f(x):
        return e**(-x**2)

def F(b):
        N=int(b*1000)
        h=0.001
        so,se=0,0
        for i in range(1,N,2):
                so+=f(i*h)
        for i in range(2,N,2):
                se+=f(i*h)
        return 1/3*h*(f(0)+f(b)+4*so+2*se)

x=linspace(0,5,100)
y=[]
for k in x:
        yd=F(k)
        y.append(yd)
plot(x,y)
show()
```

And the graph obtained as output is shown in fig.3.3: The beauty of numerical analysis is precisely this, there was no known way to compute this integral analytically!

3.3.2 Example 2

When light from stars passes through the circular aperture of a telescope(assume unit radius) and is focused on the focal plane, it produces a circular diffraction pattern ,consisting of a central bright spot surrounded by concentric rings. The intensity of light in this pattern is:

$$I(r) = \left(\frac{J_1(kr)}{kr}\right)^2$$

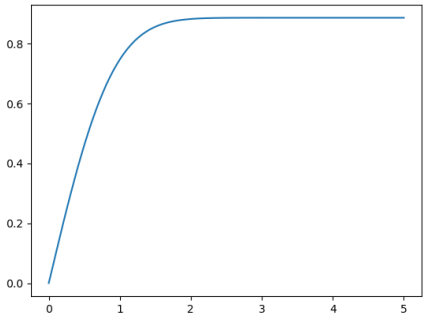


Figure 3.3: The Gaussian Integral

where r is the distance in the focal plane from the centre of the pattern, $k = \frac{2\pi}{\lambda}$ and $J_1(x)$ is a Bessel function. The Bessel functions are given by:

$$J_m(x) = \frac{1}{\pi}\int_0^{\pi} \pi \cos m\theta - x \sin\theta d\theta \tag{3.5}$$

where m is non negative integer and $x \geq 0$ Here is code to calculate and plot the first three Bessel function for $x \in [0, 20]$, using Simpson's rule and N=1000:

```
from numpy import *
from pylab import plot,show

def J(m,x):
        N=100
        h=pi/N
        so,se=0,0
        for k in range(1,N,2):
                so+=cos(m*h*k-x*sin(h*k))
        for k in range(2,N,2):
                se+=cos(m*h*k-x*sin(h*k))
        return 1/3*h*(1+cos(m*pi)+4*so+2*se)
x=arange(0,20,0.1)
m=[0,1,2,3]
for k in m:
        y=[]
        for i in x:
                yd=J(k,i)
                y.append(yd)
        plot(x,y)
show()
```

And the output is shown in fig.3.4: Now, the following function will use the

bessel function in the program above to make a density plot of the intensity of the circular diffraction pattern.

```
from numpy import *
from pylab import *
from besselfunc import J

points=1000
wavelength=500
k=2*pi/wavelength
side=1000
spacing=side/points
x0,y0=side/2,side/2
final=empty([points,points],float)
for i in range(points):
        y=spacing*i
        for j in range(points):

                x=spacing*j
                kr=k*sqrt((x-x0)**2+(y-y0)**2)
                I=(J(1,kr)/(kr))**2

                final[i,j]=I


imshow(final,origin="lower",extent=[0,side,0,side])
hot()
show()
```

Where I have divided the grid into points with 1 cm spacing and calculated the intensity at each point. The output is shown in fig.3.5:

3.4 Errors on Integrals

3.4.1 Euler-Mclaurien Formula

Numerical integrals are only an approximation, with the major source of error called the approximation error. We have seen that for a given number of slices, error of Simpson's rule is much smaller than that of trapezoidal rule. Now let's try and look deeper into these errors: From now on, let's call: $x_k = a + kh$ as a shorthand for the position at which we evaluate the integral (also called the sample points). Now, consider a slice from x_{k-1} to x_k, and let's perform a taylor expansion of $f(x)$ about x_{k-1}

$$f(x) = f(x_{k-1}) + (x - x_{k-1})f^{'}(x_{k-1}) + \frac{1}{2}(x - x_{k-1})^2 f^{''}(x_{k-1}) + ...$$

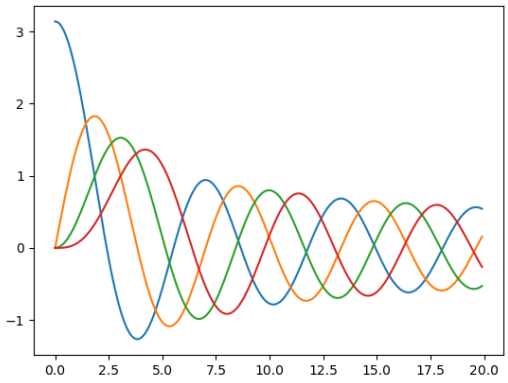


Figure 3.4: Bessel Functions for m=0,1,2

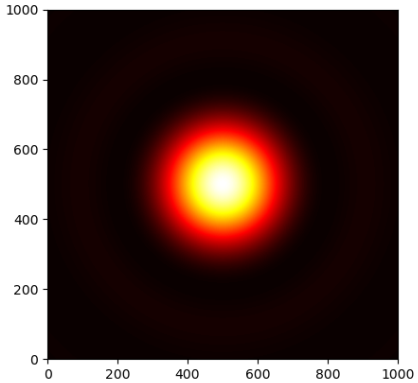


Figure 3.5: Diffraction pattern

Now, integrating this equation from x_{k-1} to x_k and making the substitution $u = x - x_{k-1}$

$$\int_{x_{k-1}}^{x_k} f(x)dx = hf(x_{k-1}) + \frac{1}{2}h^2 f^{'}(x_{k-1}) + \frac{1}{6}h^2 f^{''}(x_{k-1}) + O(h^4),$$

where $O(h^4)$ denotes rest of the terms in the series with powers of h greater than or equal to 4, which we can neglect. We can perform a similar expansion around $x = x_k$ and the result is:

$$\int_{x_{k-1}}^{x_k} f(x)dx = hf(x_{k-1}) - \frac{1}{2}h^2 f^{'}(x_{k-1}) + \frac{1}{6}h^2 f^{''}(x_{k-1}) - O(h^4),$$

Now, we can take the average of the above two equations, to get a final approximation for the integral : $\int_{x_{k-1}}^{x_k} f(x)dx$. Summing this expression over all slices k, we get the full integral:

$$\int_a^b f(x)dx = \frac{1}{2}h\sum_{k=1} N[f(x_{k-1})+f(x_k)]+\frac{1}{4}h^2[f^{'}(a)-f^{'}(b)]+\frac{1}{12}h^3\sum_{k=1} N[f^{''}(x_{k-1})+f^{''}(x_k)]+O(h^4) \tag{3.6}$$

A close look at the expression above reveals that the first term is precisely the Trapezoid rule, with the rest of the terms as the error to the rule. Also note that we don't know the actual value of the error, otherwise we would just add it to the approximation to remove the error. We call the value of the co-efficient of the leading order of h (here h^2) the approximation error ϵ. Thus:

$$\epsilon = \frac{1}{12}h^2[f^{'}(a) - f^{'}(b)] \tag{3.7}$$

This is the Euler-Mclaurien formula for the error in traezoidal rule. Python also introduces some rounding error, which is of the order of -16 (this just means that it rounds off numbers after that many decimal places). Now it can be shown that, this rounding error is negligible and becomes considerable when $N \approx 10^8$. A similar analysis for Simpson's Rule shows the approximation error to be:

$$\epsilon = \frac{1}{90}h^4[f^{'''}(a) - f^{'''}(b)] \tag{3.8}$$

It's clear that Simpson's rule is two orders more accurate than Trapezoid rule, for a given value of N.

3.4.2 Practical error estimates

For the trapezoid rule, we know that the approximation error is ch^2. Then, if I is the true integral then, $I = I_1 + ch_1^2$ Using the above equation for N_1, $N_2 = 2N_1$ and equating the two equations, we get

$$\epsilon_1 = ch_2^2 = \frac{1}{3}(I_2 - I_1)$$

Since not all functions are well behaved, we cannot always use Euler-Maclaurin formula. However, we can always use the above mentioned formula. The equivalent for Simpson's rule is :

$$\epsilon_2 = \frac{1}{15}(I_2 - I_1)$$

3.5 Adaptive integration

In this method, we start with $N_1 = 10$ and keep on doubling the value of N and checking the errors.

$$\epsilon_i = \frac{1}{3}(I_i - I_{i-1})$$

The moment we find the error to be small enough we stop the loop and return the value of the integral. Now, this method is very useful because instead of recalculating the integral in every loop, we can use the results of an iteration in the next one. This is possible because we consider evenly spaced sample points. When we double the number of slices, in effect we only have to recalculate the value of the function at the new sample points, i.e. we can reuse the values at old sample points. Using the logic mentioned above and some basic algebra, we get the following result:

$$I_i = \frac{1}{2}I_{i-1} + h_i \cdot \sum_{k_o dd} f(a + kh_i)$$

3.6 Romberg Integration

We can do better than adaptive integration with just a little more effort. The true value of the integral is $I = I_i + ch_i^2 + O(h_i^4)$. Using the ideas discussed above we can write $I = I_i + \frac{1}{3}(I_i - I_{i-1}) + O(h_i^4)$. Using only the trapezoidal rule, we have estimated the integral upto the 3rd order. This is as good as Simpson's rule. This is the basis of Romberg Integration.

$$R_{i,1} = I_i, R_{i,2} = I_i + \frac{1}{3}(I_i - I_{i-1}) = R_{i,1} + \frac{1}{3}(R_{i,1} - R_{i-1,1}) \tag{3.9}$$

$$I = R_{i,2} + c_2 h_i^4 + O(h_i^6)$$

Again, doubling the value of N and equating the two values of I, we get :

$$c_2 h_i^4 = \frac{1}{15}(R_{i,2} - R_{i-1,2}) + O(h_i^6)$$

Then,

$$I = R_{i,2} + \frac{1}{15}(R_{i,2} - R_{i-1,2}) + O(h_i^6)$$

We now have an approximation upto the fifth order! Generalizing the results:

$$I = R_{i,m} + \frac{1}{4^m - 1}(R_{i,m} - R_{i-1,m}) + O(h_i^{2m+2}) \tag{3.10}$$

$$R_{1,m+1} = R_{i,m} + \frac{1}{4^m - 1}(R_{i,m} - R_{i-1,m}) \tag{3.11}$$

$$c_i h_i^{2m} = \frac{1}{4^m - 1}(R_{i,m} - R_{i-1,m})$$

Which is accurate to h^{2m+1}. To summarize Romberg Integration:

1. Calculate the first two estimates of the integral using the trapezoidal rule, $I_1 = R_{1,1}$ and $I_2 = R_{2,1}$
2. Using this we calculate the more accurate estimate, $R_{2,2}$. This is as much as we can do with initial two estimates.
3. Now we calculate the third estimate, $I_3 = R_{3,1}$, and using the values of $R_{2,2}$ and $R_{1,1}$, we calculate $R_{3,2}$ and $R_{3,3}$
4. At each stage, we calculate an extra estimate $I_i = R_{i,1}$ with very little extra effort, and then we can find all the remaining $R_{i,m}$'s.
5. We can also calculate the errors at each stage by noticing that error at stage i is: $c_i h_i^{2m}$ and using the relation mentioned above.

A good pictorial representation of the summary is given in fig.3.6: This whole

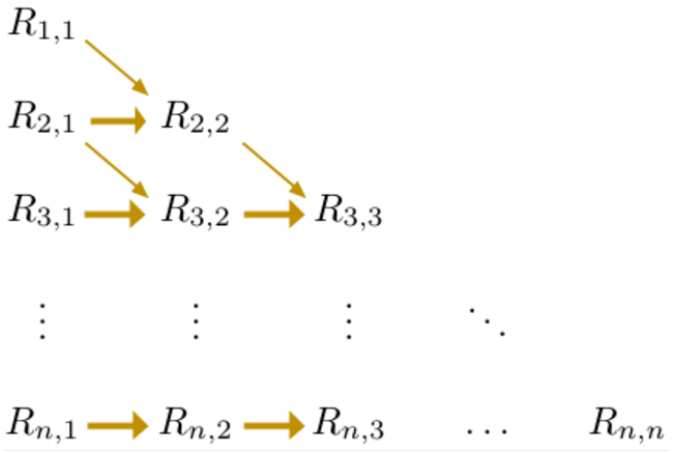


Figure 3.6: Romberg Integration

process, called Romberg integration, is essentially an add-on to the trapezoidal rule, but allows us to estimate the integral to a much higher degree with nearly no significant rise in computational time (although it requires quite a bit of programming).

Degree	Polynomial	Coefficients
1 (trapezoidal rule)	Straight line	$\frac{1}{2}, 1, 1, \ldots, 1, \frac{1}{2}$
2 (Simpson's rule)	Quadratic	$\frac{1}{3}, \frac{4}{3}, \frac{2}{3}, \frac{4}{3}, \ldots, \frac{4}{3}, \frac{1}{3}$
3	Cubic	$\frac{3}{8}, \frac{9}{8}, \frac{9}{8}, \frac{3}{4}, \frac{9}{8}, \frac{9}{8}, \frac{3}{4}, \ldots, \frac{9}{8}, \frac{3}{8}$
4	Quartic	$\frac{14}{45}, \frac{64}{45}, \frac{8}{15}, \frac{64}{45}, \frac{28}{45}, \frac{64}{45}, \frac{8}{15}, \frac{64}{45}, \ldots, \frac{64}{45}, \frac{14}{45}$

Figure 3.7: Some weights for their corresponding degrees. Image taken from the book 'Computational Physics' by Mark Newman

3.7 Higher Order Integration

All the rules we have seen thus far, are a special case of the following equation, which tries to estimate an integral, using the value of integrand at the sample points and certain "weights".

$$\int_a^b f(x)dx \approx \sum_{k=1}^{N} w_k f(x_k)$$

Some weights are shown in the fig.

3.8 Gaussian Quadrature

This will be the final, and the most accurate and ingenious numerical method of integration we discuss. Gauss came up with this method at a time when there were no computers, to solve integrals by hand, with a breath-taking precision.

3.8.1 Non uniform Sample points

Suppose we want to calculate the weights so that we can approximate the integral of any function $f(x)$. This means, we need to find a polynomial which we can fit into the sample points. Since we have N sample points, it must a N-1 degree polynomial. For the curve fitting, we use the method of interpolating polynomials: Consider:

$$\phi_k = \prod_{m \neq N} \frac{x - x_m}{x_k - x_m}$$

where $m \in [1, N]$. This N-1 degree polynomial is called the interpolating polynomial. For value of k in 1 to N, we can define N different polynomials.

$$\phi_k(x_m) = \begin{cases} 1 & if m = n \\ 0 & if m \neq n \end{cases}$$

Or, more concisely:

$$\phi_k(x_m) = \delta_{km} \tag{3.12}$$

where δ_{km} is the Kronecker Delta function, it's 1 when k=m and 0 everywhere else. Now consider:

$$\Phi(x) = \sum_{k=1}^{N} f(x_k \phi_k(x))$$

This N-1 degree polynomial, when evaluated at $x = x_m$, gives $f(x_m)$. Hence, this is the polynomial that we were looking for.

$$\int_a^b f(x)dx \approx \int_a^b \Phi(x)dx = \int_a^b \sum_{k=1}^{N} f(x_k)\phi_k(x)dx = \sum_{k=1}^{N} f(x_k) \int_a^b \phi_k(x)dx$$

It is clear now that the weights we nedd for our integration rule are :

$$w_k = \int_a^b \phi_k(x)dx$$

Now to integrate any function using only a set of sample points, all we need to do is calculate the weights by integrating the corresponding interpolating polynomial. Although, this may seem to defeat the purpose, it does not. In fact, if one has calculated the weights for a particular set of sample points and domain, they can map those weights onto any other domain. Conventionally, weights are specified in the interval $x = -1$ to $x = +1$

$$w_k = \int_{-1}^{1} \phi_k(x)dx$$

If the domain of integration is $x = a$ to $x = b$:

$$x_k^{'} = \frac{1}{2}(b-a)x_k + \frac{1}{2}(b+a)$$

$$w_k^{'} = \frac{1}{2}(b-a)w_k$$

$$\int_a^b f(x)dx \approx \sum_{k=1}^{N} w_k^{'} f(x_k^{'}) \tag{3.13}$$

3.8.2 Sample points for Gaussian Quadrature

If we have the liberty to choose N sample point, and, using them, the weights, we can accurately calculate the integrals upto the degree $2N-1$. I have omitted the proof, as it's indeed lengthy and unrelated to the matter at hand. The bottom line is: To get an integration rule accurate upto the degree $2N-1$,

the sample points x_k must be chosen to coincide with the zeroes of the N^{th} Legendre polynomial $P_N(x)$. The corresponding weights are :

$$w_k = \left[\frac{2}{1-x^2} \cdot \left(\frac{dP_N}{dx} \right)^{-2} \right]_{x=x_k}$$

The values of w_k and x_k are generally found in books and online for the domain of integration $x \in [-1, 1]$. Essentially, all we have to do now, is re-scale the values of x_k and w_k and plug them into the equation 3.13

3.8.3 Example

Finally, once again, let's compute the same integral: $\int_0^2 x^4 - 2x + 1$ using the Gaussian Quadrature. Note that the function gaussxw(N), takes input as N and returns two variables, x and w, for the assumed domain $[-1, 1]$.

```
from gaussxw import gaussxw
def f(x):
    return x**4-2*x+1
N=3
a=0.0
b=2.0
x,w=gaussxw(N)
xp=0.5*(b-a)*x+0.5*(b+a)
wp=0.5*(b-a)*w

s=0.0
for k in range(N):
    s+= wp[k]*f(xp[k])
print(s)
```

Surprisingly (or not), the outout is 4.4. This is the exact value of the integral. Notice that we have divided the domain in only three slices and got the answer upto 100 percent accuracy! This is the power of Gaussian Quadrature. The method has some disadvantages like, since the sample points aren't distributed uniformly, we can't apply adaptive methods. However, it's pretty good in itself.

3.8.4 Example 2

Debye's theory says, heat capacity of a solid at temperature T is:

$$C_v = 9V\rho k_B \left(\frac{T}{\theta_D} \right)^3 \cdot \int_0^{\frac{\theta_D}{T}} \frac{x^4 e^x}{(e^x - 1)^2} dx$$

where V is the volume, ρ is the number density of atoms, and θ_D is the Debye Temperature. The following program makes the graph of heat capacity as a function of temperature from T=0.001K to T=500k

```
from gaussxw import gaussxw
from numpy import *
from pylab import *
def f(x):
        a=x**4 * exp(x)
        b=(exp(x)-1)**(2)
        return a/b
N=50
k=1.3806e-23
x,w=gaussxw(N)
V=1000e-6
rho=6.022e28
thetaD=428
def cv(T):
        coeff=9*V*rho*k*(T/thetaD)**3
        xd=((thetaD/T)/2)*(x+1)
        wd=((thetaD/T)/2)*w
        s=0
        for i in range(N):
                s+=wd[i]*f(xd[i])
        return coeff*s
T=linspace(0.001,500,100)
y=[]
for j in T:
        y.append(cv(j))
plot(T,y,"g.")
show()
```

The output is shown in fig.3.8

3.8.5 Example 3

An anharmonic oscillator is one whose period varies with amplitude and can't be predicted analytically. To calculate it's motion, we will conserve energy:

$$E = \frac{1}{2}m\left(\frac{dx}{dt}\right)^2 + V(x)$$

Let us set the initial amplitude a, we release the particle at $t = 0$ from the position $x = a$. The total energy, thus, is $E = V(a)$ By rearranging the equation and integrating from $t = 0 to T/4$ we can show:

$$T = \sqrt{8m}\int_0^a \frac{dx}{\sqrt{V(a) - V(x)}}$$

suppose the potential is $V(x) = x^4$ and $m = 1$. The following program uses gaussian quadrature with N=20 points to approximate the integral and plots the values of time period for values of $a \in [0, 2]$

```
from gaussxw import gaussxw
from numpy import *
from pylab import *
def f(x,a):
        return sqrt((sqrt(8))/(a**4-x**4))
N=20
x,w=gaussxw(N)
a=linspace(0,2,200)
def F(a):
        xd=a/2*(x+1)
        wd=a/2*w
        s=0
        for i in range(N):
                s+=wd[i]*f(xd[i],a)
        return s
T=[]
for i in a:
        T.append(F(i))
plot(a,T,"b-")
xlabel("Amplitude")
ylabel("Time period")
show()
```

The output is shown in fig.3.9

3.8.6 Example 3

The wavefunction of the nth energy level of the 1D quantum harmonic oscillator is:

$$\psi_n(x) = \sqrt{\frac{1}{\sqrt{2^n n! \sqrt{\pi}}}} e^{-x^2/2} H_n(x)$$

for n from 0 to ∞, where $H_n(x)$ is the hermite polynomial, satisfying:

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x)$$

and $H_0(x) = 1$ and $H_1(x) = 2x$. The following program plots the wavefunction for $n = 30$ and $x \in [-10, 10]$.

```
from numpy import *
from math import factorial
from pylab import *
from gaussxw import gaussxw
def H(n,x):
        if n<0:
                raise Exception("Enter non negative n")
        if n==0:
                return(1)
```

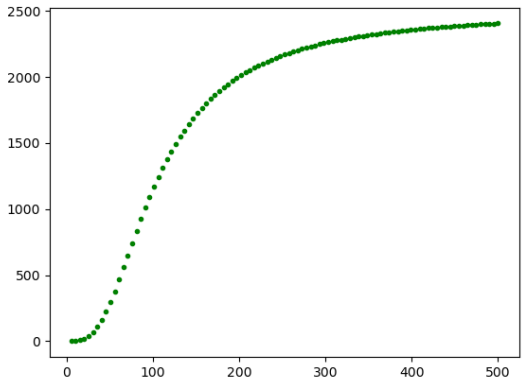


Figure 3.8: Debye plot

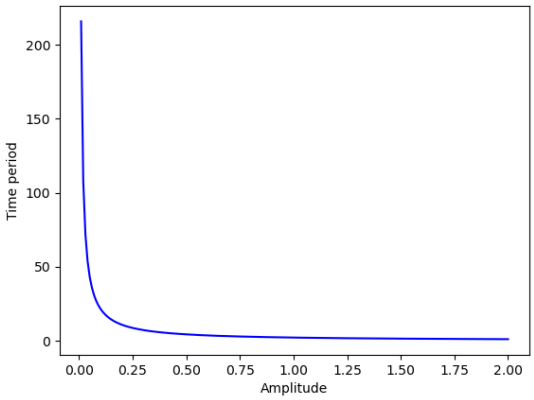


Figure 3.9: Time period decreases as amplitude increases

```
    elif n==1:
            return(2*x)
    else:
            return 2*x*H(n-1,x)-2*n*H(n-2,x)

y=linspace(-10,10,200)
psi2=(exp(-y**2/2)*H(30,y))/sqrt(2**30*factorial(30)*sqrt(pi))
plot(y,psi2,"g-")
show()
```

The plot is shown in fig.3.10

3.9 Multiple Integrals

Integrals of the form:

$$I = \int_0^1 \int_0^1 f(x,y)dxdy$$

Can be dealt with by noticing that:

$$F(y) = \int_0^1 f(x,y)dx \approx \sum_{i=1}^{N} w_i f(x_i, y)$$

And:

$$I = \int_0^1 F(y)dy \approx \sum_{j=1}^{N} w_j F(y_j)$$

Hence, we can conclude that:

$$I \approx \sum_{j=1}^{N}\sum_{i=1}^{N} w_i w_j f(x_i, y_j)$$

This is called the Gauss-Legendre Product.

3.9.1 Example

Consider a square uniform metal sheet, of side=10cm, floating in space, whose mass is 10 metric tonnes. We wish to calculate the z-component of gravitational force at a distance z from its centre, above the plane. The formula is:

$$F_z = G\sigma z \int\int_{-L/2}^{L/2} \frac{dxdy}{(x^2+y^2+z^2)^3/2}$$

The following function uses Gaussian-Legendre product to calculate and plot the force for $z \in [0, 10]$

```
from numpy import *
from pylab import *
from gaussxw import *
G=6.674e-11
sa=100
def F(x,y,z):
        return G*sa*z/(x**2+y**2+z**2)**(1.5)
z=linspace(0,10,50)
x,w=gaussxw(100)
xd=5*x
yd=xd
wd=5*w
force=[]

for k in z:
        s=0
        for j in range(100):
                for i in range(100):
                        s+=wd[i]*wd[j]*F(xd[i],yd[j],k)


        force.append(s)
plot(z,force)
xlabel("z")
ylabel("Fz")
show()
```

The output is shown in figure 3.11

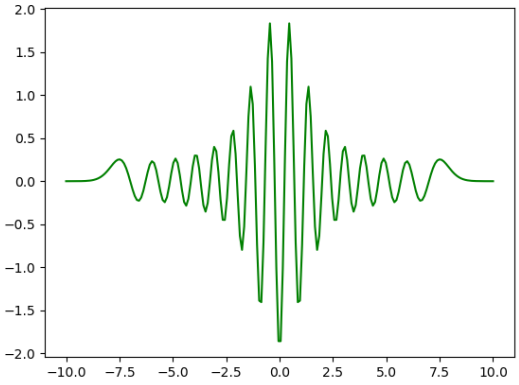


Figure 3.10: Wavefunction of a Quantum harmonic oscillator

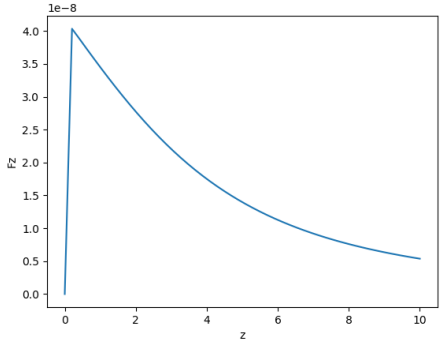


Figure 3.11: Gravitational Force of a uniform sheet

Chapter 4

Derivatives

Generally, we don't need numerical derivatives because calculating the derivative of a function analytically is quite straight forward. As we already know, the forward derivative at a point of a function is:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \approx \frac{f(x+h) - f(x)}{h}$$

for very small values of h. Similarly, backward difference can also be taken:

$$\frac{df}{dx} \approx \frac{f(x) - f(x-h)}{h}$$

4.1 Errors on derivatives

By the Taylor Expansion of $f(x)$ around x, it is quite evident:

$$f^{'}(x) = \frac{f(x+h) - f(x)}{h} + \frac{1}{2} h f^{''}(x) + ...$$

While calculating derivative using forward difference, the leading order error in h is: $\frac{1}{2} h |f^{''(x)}|$ This is the approximation error for the forward/backward difference method. Smaller values of h will reduce the approximation error, however, it will increase the rounding error. Basically, if h is very small, $f(x+h)$ and $f(x)$ are very close to each other (causing their difference to be a very small number), and since the computer is only accurate up to a number of decimal places, this causes rounding errors. It can be shown that the optimum level of precision on our derivatives is of 8 digits. Optimal error is the least value of error when both the approximation and rounding error are included.

4.2 Central Difference

What if we write:

$$\frac{df}{dx} \approx \frac{f(x+h/2) - f(x-h/2)}{h}$$

To calculate the approximation error in this, again we write

$$f(x+h/2) = f(x) + \frac{1}{2}hf^{'}(x) + \frac{1}{8}h^2f^{''}(x) + ...$$

$$f(x-h/2) = f(x) - \frac{1}{2}hf^{'}(x) + \frac{1}{8}h^2f^{''}(x) + ...$$

Subtracting the two equations, we get that the approximation error to the leading order of h^2 to be $\frac{1}{2}h^2|f^{'''}(x)|$ and it can be shown that this method gives us results which are about about 100 times more accurate than forward/backward difference. Note that like integration, we can use higher order derivatives, but in this report, I am going to go forward with just the three methods mentioned above.

4.3 Derivatives of a sampled function

Many a times in physics, we have to plot and differentiate a function, whose values are measured in an experiment, only at some specified sample points. If the sample points are measured at a uniform gap, we can either go ahead with forward difference, or we can employ central difference with a slight modification:

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x-h)}{2h}$$

Because we don't know the values of f at $x + h/2$. Now, again, there is a problem. The central difference has lesser approximation error than forward difference, but uses a larger interval. So, which is better? We can show that for the former, the optimal error is, with h replaced by $2h$, $h^2|f^{'''}(x)|$, while the optimal error for the later is: $h|f^{''}(x)|$ Thus, central difference is better when:

$$h \leq \left|\frac{f^{''}(x)}{f^{'''}(x)}\right|$$

4.4 Second Derivative and Partial Derivatives

By definition, it is the derivative of a derivative. Hence we can argue:

$$f^{'}(x+h/2) \approx \frac{f(x+h) - f(x)}{h}$$

And the similar one for $f^{'}(x-h/2)$.

$$f^{''}(x) \approx \frac{f^{'}(x+h/2) - f^{'}(x-h/2)}{h} = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

This is the simplest approximation of a second derivative, with approximation error of $\frac{1}{12}h^2f^{''''}(x)$ and optimal error of $\frac{1}{6}h^2f^{''''}(x)$. For partial derivatives, we just apply the central difference to the variable of differentiation and leave the other variables untouched.

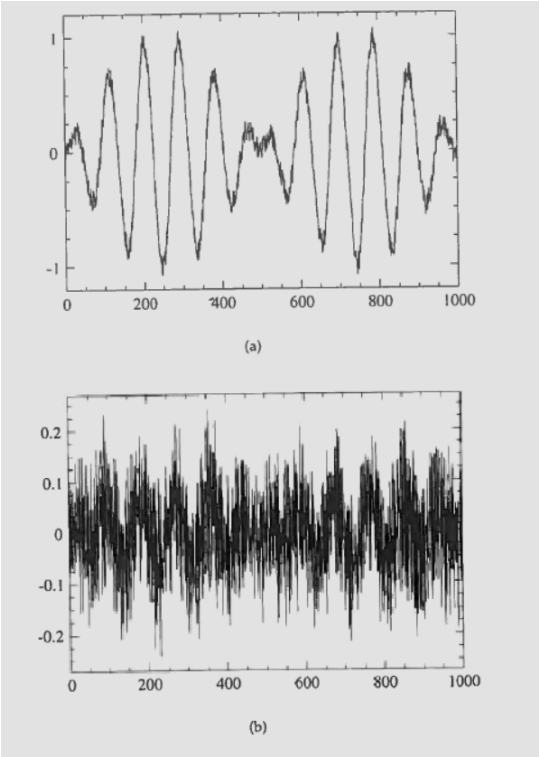


Figure 4.1: part (a) is the plot of a noisy data, and part (b) is the plot of it's derivative. Image Courtesy: "Computational Physics" by Mark Newman

4.5 Noisy data

The problem section 4.3 is that, experimental data contains noise. Now, the direct plot of the data might not be affected a lot by the noise, but the plot of the derivative loses all it's information! This effect is shown in the fig.4.1 Sadly, we can't do much regarding this problem of noise. Some things that we can do are:

- Increase the size of h, so that "noisy" fluctuations impact the derivative to a lesser extent.
- Fit a curve near the point of differentiation, then differentiate the curve.
- Use Fourier Transform to "smoothen" out the curve

Chapter 5

Linear Equations

5.1 Introduction

Let's explore some methods to solve linear equations. We will be representing linear equations in their matrix form. Example:

$$a0x + a1y + a2z = v1$$

$$b0x + b1y + b2z = v2$$

$$c0x + c1y + c2z = v3$$

Will be represented as:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} v1 \\ v2 \\ v3 \end{pmatrix} \tag{5.1}$$

Or equivalently:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{v} \tag{5.2}$$

5.2 Gaussian Elimination

In this technique, we will first model the matrix **A** in a better format, then use back-substitution to solve for each unknown. The steps are:

- Divide the first row by the first element (a_{00})
- Using multiplication and subtraction, make all the elements in first column below a_{00} zero.
- Consider the sub-matrix formed by deleting the first row and column, and repeat the above mentioned steps for this sub-matrix

- Keep repeating step 3 till you cover the entire matrix

Note that any operations done on the matrix **A** shall also be performed on the matrix **v**.

After completing the above mentioned steps, we get the matrix in the upper-triangular form, while preserving the information provided by the original equations. Now, back substitution is a cake-walk, just start with the bottom most element, and climb your way up. Here is a program to implement the technique:

```
from numpy import *
A=array([[2,1,4,1],
                [3,4,-1,-1],
                [1,-4,1,5],
                [2,-2,1,3]],float)
v=array([-4,3,9,7],float)
N=len(v)
# Gaussian Elimination
for m in range(N):

        #divide by diagonal elements
        div=A[m,m]
        A[m,:]/=div
        v[m]/=div

        #subtract from lower elements
        for i in range(m+1,N):
                mult=A[i,m]
                A[i,:]-=mult*A[m,:]
                v[i]-=mult*v[m]
        print(A)
#back substitution
x=empty(N,float)
for m in range(N-1,-1,-1):
        x[m]=v[m]
        for i in range(m+1,N):
                x[m]-=A[m,i]*x[i]
print(x)
```

And the output it gives is: $\begin{pmatrix} 2. \\ -1. \\ -2. \\ 1. \end{pmatrix}$

5.2.1 Pivoting

Now, consider the matrix **A**, but what if I make the first element $a_{00} = 0$. Notice how our little algorithm will fail at the first step itself! Well, this is a

big issue. The solution: PIVOT! Pivoting simply means interchanging rows to avoid such conflict. Here, if we encounter such issue, we could, for example, switch the first row with the second row (provided $a_{10}! = 0$). Again, make sure to perform these operations on **v** also. Now, notice that we could have such trouble anywhere while implementing Gaussian Elimination. Hence, it is advisable to use pivoting at every step. Most commonly used type of pivoting is partial pivoting. Here, what we do is, for every new row i, we will access all the elements $a_{ij} \forall j <= i$ and swap the current i^th row with the row, whose first element is the farthest from zero in magnitude. Hence we ensure that the above mentioned issue doesn't trouble us.

5.3 LU Decomposition

Gaussian Elimination (GE) is a good method, but many a times, we need to solve equations with the same matrix **A** but with different **v**. Here, applying the entire method repetitively becomes inefficient. Every time, after carrying out GE, we get the same upper triangular matrix on the LHS. In the LU method, we will pre-multiply the matrix **A** with matrices L_i such that they encapsulate the entire process of converting the matrix **A** to upper triangular form!
These matrices turn out to be:

$$L_0 = \frac{1}{a_{00}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -a_{10} & a_{00} & 0 & 0 \\ -a_{20} & 0 & a_{00} & 0 \\ -a_{30} & 0 & 0 & a_{00} \end{pmatrix}$$

$$L_1 = \frac{1}{b_{11}} \begin{pmatrix} b_{11} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -b_{21} & b_{11} & 0 \\ 0 & -b_{31} & 0 & b_{11} \end{pmatrix}$$

$$L_2 = \frac{1}{c_{22}} \begin{pmatrix} c_{22} & 0 & 0 & 0 \\ 0 & c_{22} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -c_{32} & c_{22} \end{pmatrix}$$

$$L_3 = \frac{1}{d_{33}} \begin{pmatrix} d_{33} & 0 & 0 & 0 \\ 0 & d_{33} & 0 & 0 \\ 0 & 0 & d_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For a 4x4 matrix **A**, Now, to solve the equation **Ax**=**v**, we will pre-multiply these four matrices on both sides.

$$L_3 L_2 L_1 L_0 \mathbf{Ax} = L_3 L_2 L_1 L_0 \mathbf{Av}$$

Notice here, that the RHS is completely determinable, and all we have to do is calculate the product of all the L's. Let's define:

$$\mathbf{L} = L_0^{-1} L_1^{-1} L_2^{-1} L_3^{-1}, \mathbf{U} = L_3 L_2 L_1 L_0 \mathbf{A}$$

A simple computation will show that **L** is a lower triangular matrix, while **U** is upper triangular (Hence the names). We know : **LU**=**A** Hence, we can write the original equation as:

$$\mathbf{LUx} = \mathbf{v}$$

This method also shows that any matrix **A** can be decomposed into the product two matrices, **L** and **U**.

Note that, for most of our problems, we will rely on the in-built python libraries for solving linear equations.

5.4 Eigenvalues And Eigenvectors

5.4.1 Quick Recap:

An eigen vector of a matrix **A** is **v** if :

$$\mathbf{Av} = \lambda\mathbf{v}$$

where λ is the corresponding eigen value. For an nxn matrix **A**, there will be n eigenvectors and corresponding n eigenvalues. The eigenvectors are orthogonal to each other, i.e. their dot products are zero. Moreover, we assume each eigenvector is of unit length.
We can combine all the N equations $\mathbf{A}v_i=\lambda_i v_i$ into the single equation

$$\mathbf{AV} = \mathbf{DV}$$

Where **V** is the matrix containing all the eigenvectors of **A**, and **D** is the diagonal matrix containing all the corresponding eigenvalues of **A**. Notice that **V** is an orthogonal matrix, meaning:

$$\mathbf{V^T V} = \mathbf{V V^T} = \mathbf{I}$$

5.4.2 QR Decomposition

This method is very similar to the LU decomposition. Here we break a matrix into an orthogonal matrix (Q) and an upper triangular matrix (R). The fact is, any square matrix can be QR decomposed. Let's break the matrix **A** into it's QR form:

$$\mathbf{A} = \mathbf{Q_1 R_1}$$

Multiplying on the left by $\mathbf{Q_1^T}$ and using the fact that it is orthogonal:

$$\mathbf{Q_1^T A} = \mathbf{R_1}$$

Now let's define the **new** matrix:

$$\mathbf{A_1} = \mathbf{R_1 Q_1}$$

Then we have:

$$\mathbf{A_1} = \mathbf{Q_1^T A Q_1}$$

Now, we just repeat the process by writing the QR decomposition of $\mathbf{A_1}$: $A_1 = Q_2\ \mathbf{R_2}$ And define the new matrix $\mathbf{A_2} = \mathbf{R_2}\ \mathbf{Q_2}$

$$\mathbf{A_2} = \mathbf{Q_2^T A_1 Q_2} = \mathbf{Q_2^T Q_1^T A Q_1 Q_2}$$

And if we keep doing this continuously, the resulting matrix $\mathbf{A_k}$ will be a diagonal matrix. Now, if we call this diagonal matrix $\mathbf{D}$, and define $\mathbf{V}=\Pi_{i=1}^{k}\mathbf{Q_i}$, then notice that $\mathbf{V}$ is orthogonal, and that the following equation is true:

$$\mathbf{D} = \mathbf{A_k} = \mathbf{V^t A V}$$

Or,

$$\mathbf{AV} = \mathbf{VD}$$

Which is precisely the equation for eigenvectors of the matrix $\mathbf{A}$. Hence, we found a way to find the eigenvalues and eigenvectors of a matrix.

Again, python has some inbuilt libraries for these problems!

Chapter 6

Non Linear Equations

Most of the problems in physics are based on solving non-linear equations. Here we will look at methods to solve non-linear equations

6.1 Relaxation Method

Consider the equation:

$$x = 2 - e^{-x}$$

Sadly, there is no known analytical way to solve it! So, we turn to Computational methods. One simple way is to just iterate the equation, i.e. start with a guess of x, plug it into the RHS, and the output you get shall be considered as the new x and the process shall be repeated. If we are lucky, after a few iterations, the value will converge to a fixed point. Here is a program to do so, starting with $x = 1$:

```
from math import exp
x=1.0
for k in range(10):
    x=2-exp(-x)
    print(x)
```

And the output is:

```
1.6321205588285577
1.8044854658474119
1.8354408939220457
1.8404568553435368
1.841255113911434
1.8413817828128696
1.8414018735357267
1.8414050598547234
```

1.8414055651879888
1.8414056453310121

Notice how tiny yet powerful that piece of code is! It solved the equation which wasn't possible for analytic methods to do. However, this method has some problems, like, the equation must be of the form: $x = f(x)$ and second, convergence is not guaranteed for every equation. Moreover, if an equation has multiple solutions, this method can only provide one of them for a given starting value. Consider the equation:

$$x = e^{1-x^2}$$

If we just apply the same code as above for this equation, with starting value 0.5, we get the output as:

2.117000016612675
0.030755419069985038
2.715711832754083
0.0017034651847384463
2.71827394057758
0.001679913095081425
2.7182741571849562
0.0016799111168229455
2.718274157203024
0.001679911116657934

Values of x oscillates! Luckily, this can be solved by rearranging the equation:

$$x = \sqrt{1 - \log x}$$

Now if we do the coding:

1.3012098910475378
0.8583154914892762
1.0736775779454883
0.9637999044091371
1.0182689104343374
0.990906635925747
1.004557096969838
0.997724037576543
1.0011386299421705
0.9994308469350205

We approach the answer. The mathematics behind this is: Assume the solution of the equation is $x*$, this means $x* = f(x*)$ and our guess is x'. Performing Taylor expansion of $f(x)$ about x' and neglecting the higher order h terms:

$$x' = f(x) = f(x*) + (x - x*)f'(x*)$$

And using $x* = f(x*)$

$$x' - x* = (x - x*)f'(x*)$$

Thus, the gap between our guess and the solution decreases, only when the magnitude of $f'(x*)$ is less than 1. In the example above, post rearrangement, the condition was satisfied, and so we got the solution.So, if the method fails for an equation, invert it (assuming that you can, of course).

6.1.1 Error analysis

Let us define the error on the current estimate as $\epsilon = x* - x$ and that on the next estimate as ϵ'. Then we have: $\epsilon' = \epsilon'(x*)$ and $x* = x + \epsilon = x + \frac{\epsilon'}{f'(x*)}$. Equating this with $x* = x' + \epsilon'$, we get:

$$\epsilon' \approx \frac{x - x'}{1 - \frac{1}{f'(x)}}$$

6.1.2 Example

Here is a program to solve the equation $x = 1 - e^{-cx}$ for various values of c:

```
from math import exp
from numpy import linspace
from pylab import *
c=linspace(0.01,3,300)
xf=[]
for i in c:
        error=1
        accuracy=1e-8
        x0=0.5
        while abs(error)>accuracy:
                x1=1-exp(-i*x0)
                error=(x0-x1)/(1+1/(i*exp(-i*x0)))
                x0=x1
        xf.append(x1)
        if i==2:
                print(x1)
plot(c,xf)
show()
```

I repeat the relaxation method, until the error is less than the required accuracy. The output is shown in fig:6.1 Now, we can easily extend this logic into solving multidimensional equations as follows: Suppose we are given 2 equations to solve for in 2 variables:

$$x' = f(x, y) and y' = g(x, y)$$

We will start with an initial guess for both x and y, then input them into the RHS to get the updated values for each. Repeating this process could yield the solution.

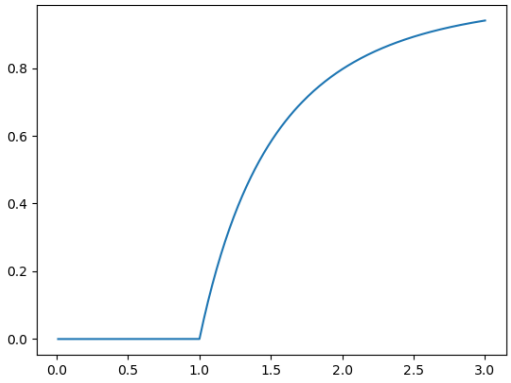


Figure 6.1: relaxation methods to solve non-linear equation

6.2 Binary Search

Also known as the bisection method, this is a more robust method in which we start with an interval and if a solution exists in it, we will find it. We simply rearrange the equation to the form : $f(x) = 0$ and then find the roots of $f(x)$. The method is summarised as follows:

- Given a pair x_1, x_2, check if $f(x_1), f(x_2)$ have different signs. Also choose target accuracy for the solution.
- Calculate the midpoint $x' = \frac{1}{2}(x_1 + x_2)$ and evaluate $f(x')$.
- If $f(x')$ has the same sign as $f(x_1)$ set $x_1 = x'$. Else set $x_2 = x'$.
- If mod $x_2 - x_1$ is less than target accuracy, repeat from step two, otherwise return $f(x')$.

Notice that in this method, our interval halves at every iteration, hence, even if we had made a wild guess for the interval, chances are that we will reach the solution rather quickly. It turns out that if our interval is of the order 10^10 with target accuracy of 10^-10, we will need... 67 steps! That's actually great. However, this method has it's own disadvantages, like $f(x_1)andf(x_2)$ must have different signs. That's a task for us! Moreover, if a root has multiplicity of more than 1, we will never know! A nice trick for the first part is, if you think of some vicinity where the root may be located, take a small interval around it, if no root was found, double the interval and repeat. Another problem is, if the root just touches the x-axis, like roots of $(1-x)^2$, the method won't detect the root at all. Finally, another drawback is that it doesn't extended easily into multiple dimensions.

Example: Wien's Constant

Planck's Radiation law says:

$$I(\lambda) = \frac{2\pi hc^2\lambda^{-5}}{e^{hc/\lambda k_b T} - 1}$$

Differentiating this equation wrt λ and setting it to 0, to find the wavelength when emitted radiation is the strongest:

$$5e^{-hc/\lambda k_b T} + \frac{hc}{\lambda k_B T} - 5 = 0$$

And making the substitution: $x = \frac{hc}{\lambda k_B T}$, we can show that the Wien's constant satisfies the equation:

$$5e^{-x} + x - 5 = 0$$

Here is a program to find the solution :

```
from math import exp
from scipy.constants import h,k,c
def f(x):
        return 5*exp(-x)+x-5
x1=4
x2=100
accuracy=1e-6
error=abs(x2-x1)
while error>accuracy:
        xm=0.5*(x1+x2)
        if f(xm)>0:
                x2=xm
        elif f(xm)==0:
                break
        else:
                x1=xm
        error=abs(x2-x1)
xs=0.5*(x2+x1)
print("Solution is: ", xs)
wien=h*c/(k*xs)
print("wien's constant: ",wien)
```

And unsurprisingly, the output is:

```
Solution is:  4.965114235877991
wien's constant:  0.0028977719527726272
```

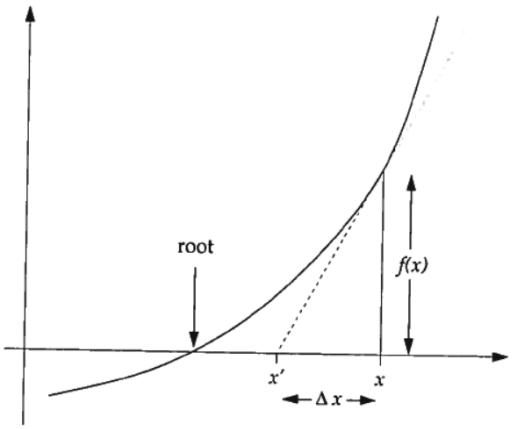


Figure 6.6: Newton's method. Newton's method takes a single estimate x of the root of a function and uses the slope of the function at that point to extrapolate a better estimate x'.

Figure 6.2: Newton's method

6.3 Newton's Method

Also called the Newton-Ralphson method, this method is slightly more robust than the previous two. It uses the slope of a function to find it's root. From the fig: 6.2, it's clear that:

$$f'(x) = \frac{f(x)}{\Delta x}$$

And hence:

$$x' = x - \Delta x = x - \frac{f(x)}{f'(x)}$$

If we know the derivative of a function, the method is straightforward, start with a guess, and iterate. For multiple roots, the method converges to the closest one. Suppose $x*$ is the true solution, and let's perform a taylor expansion of $f(x)$ about our estimate x:

$$f(x*) = f(x) + (x* - x)f'(x) + \frac{1}{2}(x* - x)^2 f''(x) + ...$$

Note that $f(x*) = 0$, then we can rearrange the equation to conclude that the errors on successive iterations follow the following relation:

$$\epsilon' = \left[\frac{-f''(x)}{2f'(x)}\right]]\epsilon^2$$

Hence it has a quadratic convergence, and it is much better than what we have seen yet. However, for small differences between successive guess, the error can be approximated to their difference. The disadvantages for this method is, firstly we must know the derivative of the function. We can get over this by using the numerical derivatives, but the more serious problem is, if $f'(x)$ is very small, error can grow larger and method could diverge.

6.3.1 Example: Roots of polynomial

Consider the following code to find the roots of the sixth degree polynomial:

```
 from numpy import *
from pylab import *

accuracy=1e-15
def P(x):
        return 924*x**6-2772*x**5+3150*x**4-1680*x**3+420*x**2-42*x+1
def Q(x):
        return 6*924*x**5-5*2772*x**4+4*3150*x**3-3*1680*x**2+2*420*x-42
x=linspace(0,1,100)
y=P(x)
plot(x,y)
show()
def find(f,x0,x1):
        delta=1.0
        while abs(delta)>accuracy:
                fp=(f(x1)-f(x0))/(x1-x0)
                delta=f(x1)/fp
                x0=x1
                x1-=delta
        return x1
roots=[]
N=arange(0,1.2,0.2)
for x in N:
        roots.append(find(P,x,x+0.05))
print(roots)
```

The ouput is a plot of the function (6.3) is the plot of the polynomial (which happens to be the 6th legendre polynomial) with the roots:

```
[0.033765242898423996, 0.1693953067668677, 0.3806904069583992, 0.61930959304160
```

6.4 Secant Method

This is similar to Newton's Method, but we will deal with two points at a time. Start with two points $x_1 and x_2$, they don't need to bracket the root. In the final

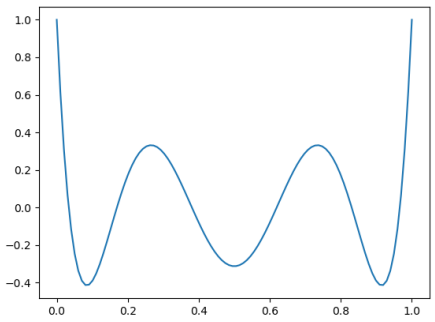


Figure 6.3: This is the sixth Legendre polynomial

equation of Newton's method, for $f'(x)$ substitute:

$$f'(x_2) \approx \frac{f(x_2) - f(x_1))}{x_2 - x_1}$$

Then, we get:

$$x_3 = x_2 - f(x_2)\frac{x_2 - x_1}{f(x_2) - f(x_1))}$$

This new guess is based on two previous values.

6.4.1 Example: Lagrange points

Lagrange point between Earth and Moon is the point at which, a satellite will orbit the earth in perfect synchrony with the moon. The following program computes the lagrange point (L1 point) by employing the secant method:

```
 from numpy import *
#constants
G=6.674e-11
M=5.974e24
m=7.348e22
R=3.844e8
w=2.662e-6
accuracy=1e-3
def P(r):
        return G*(M*(R-r)**2-m*r**2)-w**2*r**3*(R-r)**2
def find(f,r0,r1):
        delta=10
        while abs(delta)>accuracy:
                fp=(f(r1)-f(r0))/(r1-r0)
                delta=f(r1)/fp
```

```
                r0=r1
                r1-=delta
        return r1
print(find(P,2*R,5*R))
```

And the output is :

```
326045071.66535544
```

which is accurate to 4 sf, as good as the values of constants we assumed!

6.4.2 Higher dimensions

Suppose we have the set of equations:

$$f_1(x_1, ..., x_n) = 0$$

$$\vdots$$

$$f_n(x_1, ..., x_n) = 0$$

Then the equivalent of taylor expansion is:

$$f_i(x_1^*, ..., x_n^*) = f_i(x_1, ..., x_n) + \sum_j (x_j^* - x_j) \frac{\delta f_i}{\delta x_j} + ...$$

Or, using vector notation:

$$\mathbf{f}(\mathbf{x}*) = \mathbf{f}(\mathbf{x}) + \mathbf{J} \cdot (\mathbf{x} * -\mathbf{x}) + ...$$

Where $\mathbf{J}$ is the Jacobian matrix, with $J_{ij} = \frac{\delta f_i}{\delta x_j}$ Since $\mathbf{f}(\mathbf{x}*) = \mathbf{0}$, we have:

$$\mathbf{J} \cdot \mathbf{\Delta x} = \mathbf{f}(\mathbf{x})$$

Guess what, this equation if of the type $\mathbf{Ax} = \mathbf{v}$! Hence,we solve for $\mathbf{\Delta x}$, then calculate our new estimate by:

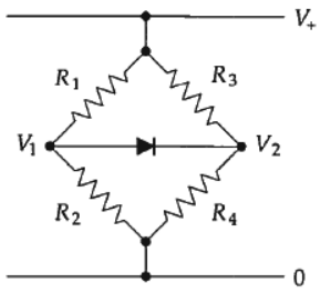
$$\mathbf{x}' = \mathbf{x} - \mathbf{\Delta x}$$

6.4.3 Example: Non-linear circuits

Let's solve the following circuit (ref: 6.4) The following code solves for V_1 and V_2 it, using Krichoff's laws and Newton's method in multiple dimensions:

```
 from numpy import *
from numpy.linalg import solve
from math import exp
#constants
vplus=5
```

Consider the following simple circuit, a variation on the classic Wheatstone bridge:



The resistors obey the normal Ohm law, but the diode obeys the diode equation:

$$I = I_0(e^{V/V_T} - 1),$$

Figure 6.4: Non-linear circuit

```
R1=1000
I0=3*10**-9
R2=4000
Vt=0.05
R3=3000
R4=2000
def f1(v1,v2):
        return (v1-vplus)/R1+v1/R2+I0*(exp((v1-v2)/Vt)-1)
def f2(v1,v2):
        return (-v2+vplus)/R3-v2/R4+I0*(exp((v1-v2)/Vt)-1)
def pder(v1,v2):
        h=1e-7
        J=empty([2,2],float)
        '''
        J[0,0]=(f1(v1+h,v2)-f1(v1,v2))/h
        J[1,0]=(f1(v1,v2+h)-f1(v1,v2))/h
        J[0,1]=(f2(v1+h,v2)-f2(v1,v2))/h
        J[1,1]=(f2(v1,v2+h)-f2(v1,v2))/h
        '''
        k=exp((v1-v2)/Vt)
        J[0,0]=1/R1+1/R2+I0/Vt*k
        J[1,0]=I0/Vt*k
        J[1,1]=-1/R3-1/R4-I0/Vt*k
        J[0,1]=-I0/Vt*k
        return J
```

```
def f(v1,v2):
        F=empty([2,1],float)
        F[0,0]=f1(v1,v2)
        F[1,0]=f2(v1,v2)
        #print(F)
        return F
def X(v1,v2):
        X=empty([2,1],float)
        X[0,0]=v1
        X[1,0]=v2
        return X
def inv(J):
        Jin=array([[J[1,1],-J[0,1]],
                           [-J[1,0],J[0,0]]
                           ])
        det=J[0,0]*J[1,1]-J[1,0]*J[0,1]
        Jin/=det
        return Jin

def Newt(v10,v20):
        x=X(v10,v20)
        for i in range(50):
                #print(x)
                J=pder(x[0,0],x[1,0])
                print(J)
                fx=f(x[0,0],x[1,0])
                delx=solve(J,fx)
                x-=delx
        return x
ans=Newt(3.4,2.8)
print("The ans is: ",ans)
```

With the ouput:

```
The ans is:  [[3.44695462],[2.82956807]]
```

Notice that the difference between V_1 and V_2 is nearly 0.617, and we know that the voltage across a forward biased diode should've been just that much!

6.5 Maxima and Minima

This is really close to just finding the roots of the first (and sometimes the second) derivatives of a function.

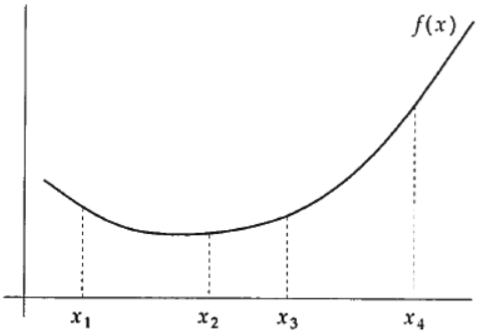


Figure 6.9: Golden ratio search. Suppose there is a minimum of the function $f(x)$ between x_1 and x_4. If the function is lower at x_2 than at x_3, as shown here, then we know the minimum must lie between x_1 and x_3; otherwise it lies between x_2 and x_4. Either way, we have narrowed the range in which it lies.

Figure 6.5: The Golden Ratio Search

6.5.1 The trivial method

Given a function $f(x_1, ..., x_n)$, to minimise it wrt all variables, all we do is set it's derivative wrt to each variable zero, and get a set of n equations, which we are now good at solving. However most of the times, this method fails, as we don't know the derivative of the function (maybe because it was the output of another function, or because it was a collection of experimental data), and also, this method can't differentiate between local and global derivatives!

6.5.2 Golden Ratio Search

This method is similar to the binary search we saw earlier. Here, we bracket the the minimum with four points, as shown in fig 6.5. Suppose that at least one of $f(x_2) and f(x_3)$ is less than the values at the outer points of f (at $x_1 and x_4$). This confirms that at least one minimum exists between the outer points. Now, we narrow down the interval by comparing values at $x_2 and x_3$. If $f(x_2)$ is smaller, then minimum must lie in between $x_1 and x_3$, and vice versa. Like this, we first narrow our search to three points, then add a fourth point to repeat the process. Now, we must decide the distribution of the four points. It turns out, that when they are optimally distributed, the ratio between the width of the interval before and after the search is:

$$z = \frac{x_4 - x_1}{x_3 - x_1} = \frac{x_2 - x_1}{x_3 - x_1} + 1 = \phi$$

where ϕ is the golden ratio (hence the name). The complete search can be summarised as:

- Choose the outer points and calculate the interior ones, making sure that atleast one of their outputs is less than that at the outer points.
- If $f(x_2) < f(x_3)$, then the minimum lies between $x_1 and x_3$, so x_3 becomes the new x_4, x_2 becomes the new x_3 and we find a new x_2. Otherwise, similar arguments are applied for the other case.
- If $x_4 - x_1$ is greater than the target accuracy, repeat the above step, else the final answer is $\frac{1}{2}(x_2 + x_3)$.

Example: Light Bulb

Let's find the optimum temperature for the functioning of a light bulb, one where it's efficiency is maximum. Defining efficiency as the fraction of radiation that falls in the visible range of light. Power of the bulb follows:

$$I(\lambda) = A\frac{2\pi hc^2\lambda^{-5}}{e^{hc/\lambda k_b T} - 1}$$

where A is the area of the filament. Let $\lambda_1 = 390nm$ and $\lambda_2 = 750nm$, so the total energy radiated in the visibe region is: $|\int_{\lambda_1}^{\lambda_2} I\lambda d\lambda$ and similar for the total energy, with the limits $0, \infty$. Then with the substitution, $x = \frac{hc}{\lambda k_B T}$, the efficiency becomes:

$$\eta = \frac{15}{\pi^4}\int_{hc/\lambda_2 k_B T}^{hc/\lambda_1 k_B T} \frac{x^3}{e^x - 1}dx$$

The program below plots the graph of the efficiency, and pin-points it's maxima:

```
 from math import exp
from numpy import *
from pylab import *
from gaussxw import gaussxw
#constant
hc=1.98644568e-25
w1=390e-9
w2=750e-9
kb=1.380649e-23
N=100
k=15/pi**4
x,w=gaussxw(N)
accuracy=1


def f(x):
        a=x**3
        b=exp(x)-1
        return a/b
```

```
T=linspace(300,10000,7000)
y=[]
def eta(t):
        a=hc/(w2*kb*t)
        b=hc/(w1*kb*t)
        xd=(b-a)/2*x+(b+a)/2
        wd=w*(b-a)/2
        s=0
        for i in range(N):
                s+=wd[i]*f(xd[i])
        return k*s

for t in T:
        y.append(eta(t))
plot(T,y,"b-")
xlabel("Temperature")
ylabel("Efficiency")
show()

z=1.6180339887
#initial 4 points
x1=5000
x4=8000
x2=x4-(x4-x1)/z
x3=x1+(x4-x1)/z
print(x2,x3)

#functional value of the initial points
f1=eta(x1)
f2=eta(x2)
f3=eta(x3)
f4=eta(x4)

#the loop
while x4-x1>accuracy:
        if f2>f3:
                x4,f4=x3,f3
                x2=x4-(x4-x1)/z
                x3=x1+(x4-x1)/z
                f2=eta(x2)
                f3=eta(x3)
        else:
                x1,f1=x2,f2
                x2=x4-(x4-x1)/z
                x3=x1+(x4-x1)/z
                f2=eta(x2)
```

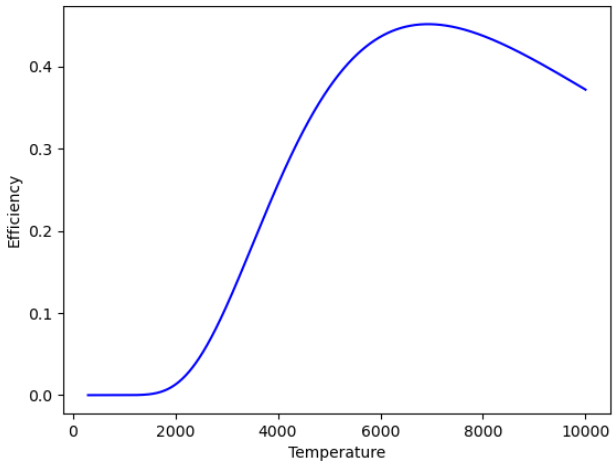


Figure 6.6: Efficiency of a light bulb

```
f3=eta(x3)

print("The efficiency is maximum at: ",0.5*(x2+x3)," K")
```

The plot is shown in fig: 6.6 and the output is :

```
The efficiency is maximum at:  6928.53768615682  K
```

Although you can imagine why we don't operate bulb at this temperature!

6.5.3 Newton-Gauss method and Gradient Descent

The Newton-Guass method is simply finding the root of the equation :

$$f'(x) = 0$$

We can use the Newton's method to solve this, as follows:

$$x' = x - \frac{f'(x)}{f''(x)}$$

Although, we might not always know all the derivatives. For this, we have the method of Gradient Descent:

$$x' = x - \gamma f'(x)$$

Here, we have replace the second derivative by a constant, which need not be quite accurate, the correct order is more than sufficient.A positive value of γ signifies that the method will move downhill from x to x' and converge to a

minimum, while a negative γ means the method converges to a maximum Trial and error is generally the path to follow. If in case, we don't know the first derivative either, switch to the equivalent of the secant method:

$$f'(x_2) \approx \frac{f(x_2) - f(x_1))}{x_2 - x_1}$$

$$x_3 = x_2 - \gamma \frac{f(x_2) - f(x_1))}{x_2 - x_1}$$

This mehtod also gives us a rate of convergence similar to that of Newton's Method.
We haven't yet seen a method to find the global minimum/ maximum of a function. Turns out, this is a very big problem in computational physics, the solution to which was given by M. Pincus in 1970 and is called Simulated Annealing. This method is described in the last chapter.

Chapter 7

Ordinary Differential Equations

7.1 Introduction

We have reached arguably the most important topic in computational physics. Most of the laws in physics are written in differential form, and solving them is of utmost importance.

7.1.1 First order one-variable equation

The general way to represent such equation is:

$$\frac{dx}{dt} = f(x,t)$$

To calculate the full solution to this eqauation, we need an initial condition or a boundary value. Let's go through some methods to do the same:

7.2 Euler's method

Let's say we are given the above mentioned equation, and an initial condition that fixes x for a particular t. Then, we can write:

$$x(t+h) = x(t) + h\frac{dx}{dt} + \frac{1}{2}h^2\frac{d^2x}{dt^2} + ...$$

For x after a small time interval h OR, ignoring higher order terms of h:

$$x(t+h) = x(t) + hf(x,t)$$

Thus, given a pair of (x,t) we can find x at evenly spaced intervals later. Here is an examle:

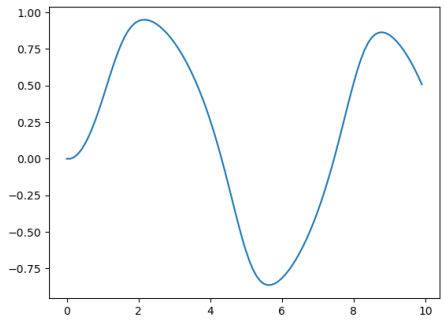


Figure 7.1: Euler's method

```
from numpy import *
from numpy import arange
from pylab import *
from math import sin
def f(x,t):
    return -x**3+sin(t)
#constants
a=0.0
b=10.0
N=1000
h=(b-a)/N
s=0.0
x=0.0
tp=arange(a,b,h)
xp=[]
for t in tp:
    xp.append(x)
    x+=h*f(x,t)
plot(tp,xp)
show()
```

And the output is shown in fig: 7.1 This method is good, but not used very often, because it introduces an error proportional to h^2 at each step. Thus is can be shown that the cumulative error is proportional to h which is not very good in terms of accuracy.

7.3 Runge-Kutta Method

This is actually a set of methods. Technically, Euler's method is just the first order Runge-Kutta method! The second order method, also called the midpoint

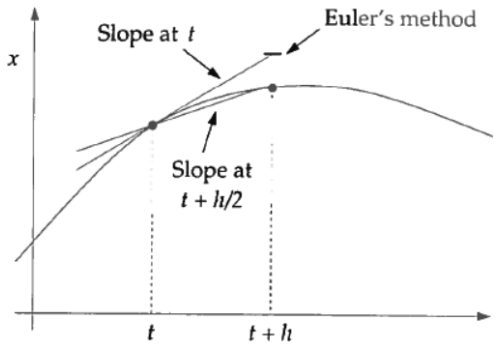


Figure 8.2: Euler's method and the second-order Runge–Kutta method. Euler's method is equivalent to taking the slope dx/dt at time t and extrapolating it into the future to time $t + h$. A better approximation is to perform the extrapolation using the slope at time $t + \frac{1}{2}h$.

Figure 7.2: Euler's method vs RG-2

method is as follows:

7.3.1 Midpoint Method

The following image (fig: 7.2) summarises this method: Here, unlike Euler's method, we will use the derivative of x at $t+\frac{1}{2}h$ to predict the value of $x(t+h)$. In mathematical terms: Perform taylor expansion of $x(t+h)$ around $t+\frac{1}{2}h$ to get :

$$x(t+h) = x(t+\frac{1}{2}) + \frac{1}{2}h\left(\frac{dx}{dt}\right) + \frac{1}{8}h^2\frac{d^2x}{dt^2} + ...$$

*where the derivatives are evaluated at $t+\frac{1}{2}$. Similarly, for $x(t)$:

$$x(t) = x(t+\frac{1}{2}) - \frac{1}{2}h\left(\frac{dx}{dt}\right) + \frac{1}{8}h^2\frac{d^2x}{dt^2} + ...$$

Subtracting the second from the first and rearranging:

$$x(t+h) = x(t) + h\frac{dx}{dt} + O(h^3) = x(t) + hf(x(t+\frac{1}{2}h), t+\frac{1}{2}h)$$

Now, notice that the error on each term is of the third order! However, we don't know the value of the function at $t+\frac{1}{2}h)$. We get around this problem by writing $x(t+\frac{1}{2}h)) = x(t) + \frac{1}{2}hf(x,t)$ by Euler's method. The complete calculation for a single step is:

$$k_1 = hf(x,t)$$

$$k_2 = hf(x + \frac{1}{2}k_1, t + \frac{1}{2}h)$$

$$x(t + h) = x(t) + k_2$$

We can show that the error introduced due to the approximation of $f(x(t+\frac{1}{2}h))$ is also of the order h^3, and hence, overall this method has an error of $\mathrm{O}(h^3)$ per step.
To see the power of this method, let's solve the same ODE, with RG-2

```
from numpy import *
from numpy import arange
from pylab import *
from math import sin
def f(x,t):
    return -x**3+sin(t)
#constants
a=0.0
b=10.0
N=80
h=(b-a)/N
x=0.0
tp=arange(a,b,h)
xp=[]
for t in tp:
    xp.append(x)
    k1=h*f(x,t)
    k2=h*f(x+k1/2,t+h/2)
    x+=k2
plot(tp,xp)
xlabel(N)
show()
```

And the ouput is in figure 7.3 We managed to get the same result by using only 80 points, as opposed to Euler's method, where we used 1000 points!

7.3.2 Fourth Order RG

In theory, we can extend this method as many times as we like, but the most famous one is this. This variant gives very good accuracy with very little extra efforts.
The summary of this method is:

$$k_1 = hf(x, t)$$

$$k_2 = hf(x + \frac{1}{2}k_1, t + \frac{1}{2}h)$$

$$k_3 = hf(x + \frac{1}{2}k_2, t + \frac{1}{2}h)$$

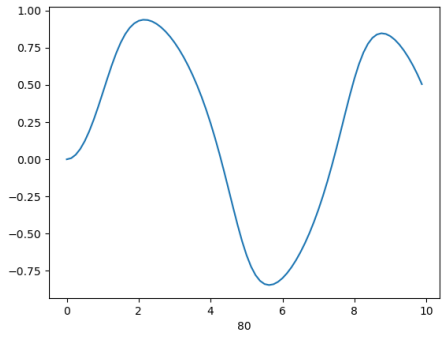


Figure 7.3: RG-2

$$k_4 = hf(x + k_3, t + h)$$

$$x(t + h) = x(t) + \frac{1}{6}(k_1 + k_4 + 2(k_2 + k_3))$$

It gives an error of the order h^5 per step.

Low pass filter

The figure 7.4 shows the low pass filter circuit. It can be easily shown that the output voltage follows:

$$\frac{dV_{out}}{dt} = \frac{1}{RC}(V_{in} - V_{out})$$

The following program solves the ODE for RC=0.05, initial condition $V_{out}(0) = 0$ and V_{in} as specified in the code:

```
from numpy import *
from pylab import *
#constants
RC=0.05
def f(V,t):
    td=int(2*t)
    if td%2==0:
        return 1/RC*(1-V)
    else:
        return 1/RC*(-1-V)
a=0
b=10
N=1000
h=(b-a)/N
tp=arange(a,b,h)
```

```
v=0.0
V=[]
for t in tp:
    V.append(v)
    k1=h*f(v,t)
    k2=h*f(v+k1/2,t+h/2)
    k3=h*f(v+k2/2,t+h/2)
    k4=h*f(v+k3,t+h)
    v+=(k1+k4+2*(k2+k3))/6
plot(tp,V)
xlabel("time")
ylabel("Vout")
show()
```

The ouput is shown in fig: 7.5

7.3.3 Infinite ranges

Problems which require us to solve the ODE over infinite range can again be tackled by a simple substitution of variables. Let's say $t \to \infty$ Write:

$$u = \frac{t}{1+t}, t = \frac{u}{1-u}$$

now as $t \to \infty, u \to 1$. We can use chain rule to re-write:

$$\frac{dx}{du} = (1-u)^{-2} f\left(x, \frac{u}{1-u}\right)$$

7.4 DE with multiple variables

The derivative of a variable can depend on other dependent variables also, in addition to the independent variables (t). Such equations are called simultaneous differential equations and are represented as:

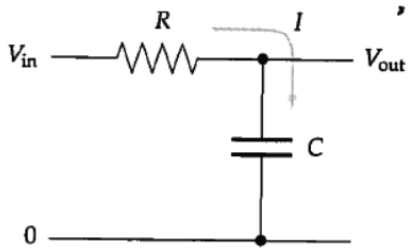
$$\frac{dx}{dt} = f_x(x,y,t), \frac{dy}{dt} = f_y(x,y,t)$$

Where f_x and f_y are general (maybe non-linear) functions. For arbitrary number of variables:

$$\frac{d\mathbf{r}}{dt} = \mathbf{f}(\mathbf{r}, t)$$

where $\mathbf{r}$=(x,y,z,...) and fis the vector function $(f_x(\mathbf{r},t),..)$ For a computer, scalar are similar to vectors, and so the methods don't change. In the method of RG-4, we can simply substitute x with $\mathbf{r}$

Here is a simple electronic circuit with one resistor and one capacitor:



This circuit acts as a low-pass filter: you send a signal in on the left and it comes out filtered on the right.

Figure 7.4: A low pass filter

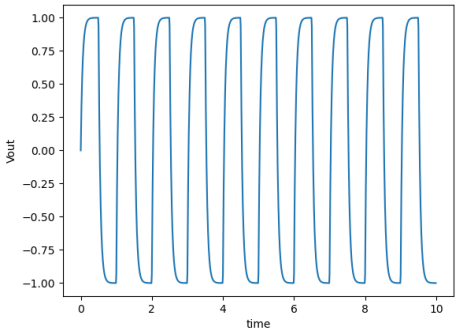


Figure 7.5: Output of the low pass filter

7.4.1 The Lotka Volterra Equation

These are the mathematical models describing predator prey realtions in wild! Let two variables x, y be proportional to the sizes of two species (rabbits,foxes). The Lotka-Volterra equations satisfy:

$$\frac{dx}{dt} = \alpha x - \beta xy$$

$$\frac{dy}{dt} = \gamma yx - \delta y$$

The following program solves these equations, with the initial condition $x = y = 2$.

```
from numpy import *
from pylab import *
#constants
alpha=1
beta=0.5
gamma=0.5
delta=2

def f(r,t):
    x=r[0]
    y=r[1]
    fx=alpha*x-beta*x*y
    fy=gamma*x*y-delta*y
    return array([fx,fy],float)

a=0.0
b=30.0
N=1000
h=(b-a)/N
tp=arange(a,b,h)
xp=[]
yp=[]
r=array([2.0,2.0],float)
for t in tp:
    xp.append(r[0])
    yp.append(r[1])
    k1=h*f(r,t)
    k2=h*f(r+k1/2,t+h/2)
    k3=h*f(r+k2/2,t+h/2)
    k4=h*f(r+k3,t+h)
    r+=(k1+k4+2*(k2+k3))/6
plot(tp,xp,label="rabbits")
plot(tp,yp,label="foxes")
```

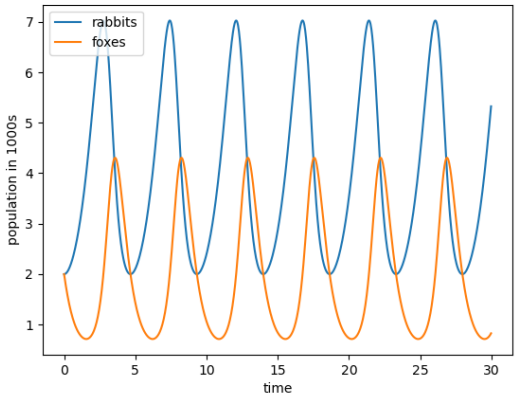


Figure 7.6: Population of foxes and rabbits

```
xlabel("time")
ylabel("population in 1000s")
legend()
show()
plot(xp,yp)
xlabel("rabbits")
ylabel("foxes")
show()
```

And the ouput is shown in fig: 7.6

7.4.2 The Lorenz Attractor

One of the most famous DE in physics are:

$$\frac{dx}{dt} = \sigma(y - x), \frac{dy}{dt} = rx - y - xz, \frac{dz}{dt} = xy - bz$$

These are one of the first examples of deterministic chaos. The following code makes it clear:

```
from numpy import *
from pylab import *
#constants
q=10    #sigma
w=28     #r
e=8/3    #b
def f(r,t):
    x=r[0]
    y=r[1]
```

```
    z=r[2]
    fx=q*(y-x)
    fy=w*x-y-x*z
    fz=x*y-e*z
    return array([fx,fy,fz],float)
a=0.0
b=50.0
N=100000
h=(b-a)/N
r=array([0,1,0],float)
tp=arange(a,b,h)
xp=[]
yp=[]
zp=[]

for t in tp:
    xp.append(r[0])
    yp.append(r[1])
    zp.append(r[2])
    k1=h*f(r,t)
    k2=h*f(r+k1/2,t+h/2)
    k3=h*f(r+k2/2,t+h/2)
    k4=h*f(r+k3,t+h)
    r+=(k1+k4+2*(k2+k3))/6
plot(tp,yp,label="y vs t")
legend()
show()
plot(xp,zp,label="lorenz attractor")
legend()
show()
```

And the (very attractive) output is shown in fig: 7.7

7.5 Second order DE

Most common types of differential equations in physics are these. Luckily, we already have all the tools to solve them. Consider the simple case:

$$\frac{d^2x}{dt^2} = f\left(x, \frac{dx}{dt}, t\right)$$

The trick is:

$$\frac{dx}{dt} = y$$

Hence:

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

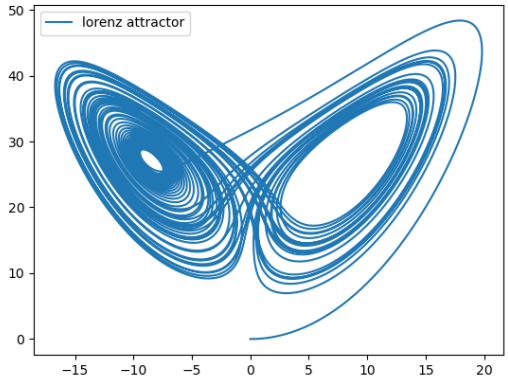


Figure 7.7: The famous Lorenz attractor

And we have reduced the problem to one which has already been solved! This is just a sysytem of simultaneous DE. Such a trick can be generalised to any ordered equation. Again, if you've got simultaneous set of 2nd order DE, first convert them to single variable equivalent by writing **r**=(x,y,z,..) and then applying the above mentioned trick.

7.5.1 The non-linear pendulum

For a pendulum, with an angular displacement of θ, the tangential acceleration is $l\frac{d^2\theta}{dt^2}$, while mg acts downwards. Thus we get the equation:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin\theta$$

This is a second order DE, which is solved by the following program:

```
from numpy import *
from pylab import *
#constants
g=9.81
l=0.1
def f(r,t):
    theta=r[0]
    omega=r[1]
    ftheta=omega
    fomega=-g/l*sin(theta)
    return array([ftheta,fomega],float)
a=0.0
```

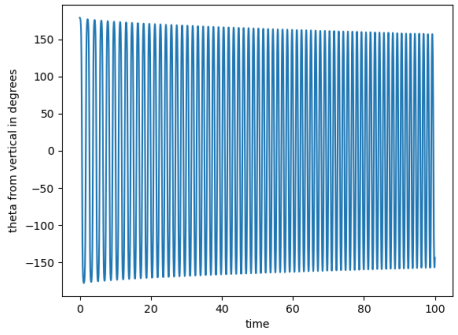


Figure 7.8: Note that the amplitude falls with time. This wasn't coded for! This is actually a fault in RG-4 method. We will later see that for such programs Leap-frog method is better

```
b=100.0
N=2500
h=(b-a)/N
tp=arange(a,b,h)
thetap=[]
timep=[]
r=array([(pi/180)*179,0])
for t in tp:
    thetap.append((180/pi)*r[0])
    timep.append(2*pi/r[1])
    k1=h*f(r,t)
    k2=h*f(r+k1/2,t+h/2)
    k3=h*f(r+k2/2,t+h/2)
    k4=h*f(r+k3,t+h)
    r+=(k1+k4+2*(k2+k3))/6
plot(tp,thetap)
xlabel("time")
ylabel("theta from vertical in degrees")
show()
```

And the output is in fig: 7.8

7.5.2 Trajectory with air resistance

We have always ignored air resistance , as it made calculations cumbersome. But no more! For a sphere moving in air, the resistive force is:

$$F = \frac{1}{2}\pi R^2 \rho C v^2$$

Using Newton's second law, we can show :

$$x'' = -kx'\sqrt{x'^2 + y'^2}, y'' = -g - ky'\sqrt{x'^2 + y'^2}$$

Here is the code to solve it:

```
from numpy import *
from scipy.constants import g
from pylab import *
#constants
m=1
r=0.08
theta=30*pi/180
v0=100
rho=1.22
C=0.47
k=pi*r**2*rho*C/(2*m)
def f_v(r):
    vx=r[0]
    vy=r[1]
    fvx=-k*vx*sqrt(vx**2+vy**2)
    fvy=-g-k*vy*sqrt(vx**2+vy**2)
    return array([fvx,fvy],float)

def f_x(rx,rv):
    x=rx[0]
    y=rx[1]
    fx=rv[0]
    fy=rv[1]
    return array([fx,fy],float)
x=[]
y=[]
h=0.0001
r_x=array([0,0],float)
r_v=array([v0*sqrt(3/4),v0*0.5],float)

while r_x[1]>=0:
    x.append(r_x[0])
    y.append(r_x[1])
    k1=h*f_v(r_v)
    k2=h*f_v(r_v+k1/2)
    k3=h*f_v(r_v+k2/2)
    k4=h*f_v(r_v+k3)
    r_v+=(k1+k4+2*(k2+k3))/6
    k1x=h*f_x(r_x,r_v)
    k2x=h*f_x(r_x+k1x/2,r_v)
    k3x=h*f_x(r_x+k2x/2,r_v)
```

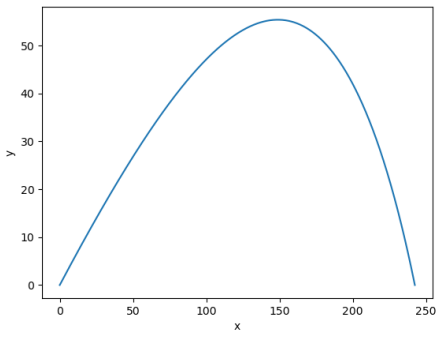


Figure 7.9: Trajectory with air resistance

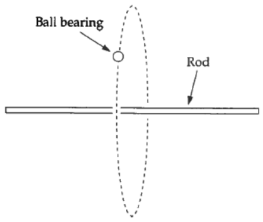


Figure 7.10: Space Garbage

```
    k4x=h*f_x(r_x+k3x,r_v)
    r_x+=(k1x+k4x+2*(k2x+k3x))/6
plot(x,y)
xlabel("x")
ylabel("y")
show()
```

And the output is in fig:7.9

7.5.3 Space Garbage

Let's say that a ball was left in space, and it started orbiting a rod (fig:7.10) After applying Newton's Law of Gravitation, it can be shown that the motion follows the differential equations:

$$\frac{d^2x}{dt^2} = -GM\frac{x}{r^2\sqrt{r^2+L^2/4}}, \frac{d^2y}{dt^2} = -GM\frac{y}{r^2\sqrt{r^2+L^2/4}}$$

Where $r = \sqrt{x^2+y^2}$ These equations are solve by the following:

```
from numpy import *
from math import sqrt
```

```
from pylab import *
from tqdm import tqdm
#constants
G=1
M=10
L=2

def fv(rv,rx):
    rsqr=float(rx[0]**2+rx[1]**2)
    fax=-G*M*rx[0]/(rsqr*sqrt(rsqr+L**2/4))
    fay=-G*M*rx[1]/(rsqr*sqrt(rsqr+L**2/4))
    return array([fax,fay],float)

def fx(rv):
    fvx=rv[0]
    fvy=rv[1]
    return array([fvx,fvy],float)

N=10000
time=linspace(0,100,N)
h=100/N
rx=array([1,0],float)
rv=array([0,1],float)
x=[]
y=[]
for t in tqdm(time):
    x.append(rx[0])
    y.append(rx[1])
    k1=h*fv(rv,rx)
    k2=h*fv(rv+k1/2,rx)
    k3=h*fv(rv+k2/2,rx)
    k4=h*fv(rv+k3,rx)
    rv+=(k1+k4+2*(k2+k3))/6
    k1x=h*fx(rv)
    k2x=h*fx(rv+k1x/2)
    k3x=h*fx(rv+k2x/2)
    k4x=h*fx(rv+k3x)
    rx+=(k1x+k4x+2*(k2x+k3x))/6
plot(x,y)
xlabel("x")
ylabel("y")
show()
```

And the output is shown in fig: 7.11

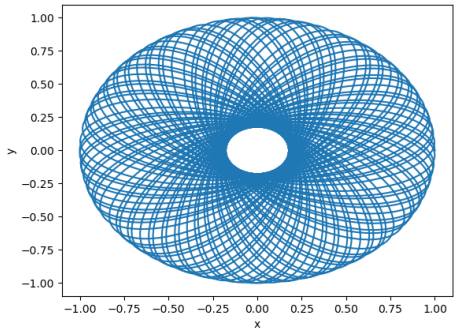


Figure 7.11: Motion of Space junk. Beautiful, isn't it!

7.6 Other methods for DE

7.6.1 The Leapfrog method/ The modified midpoint method

This method starts similar to the RG method, with a half-step to the midpoint, folloed by the full step to calculate $x(t+h)$. But then, instead of using $x(t+h)$ to calculate the next midpoint, we instead calculate it from the previous midpoint values $x(t+\frac{1}{2}h)$:
Given values of $x(t)$ and $x(t+h/2)$

$$x(t+h) = x(t) + hf(x(t+\frac{1}{2}h), t+\frac{1}{2}h)$$

$$x(t+\frac{3}{2}h) = x(t+\frac{1}{2}h) + hf(x(t+h), t+h)$$

*Note, again simultaneous DE can be solved by replacing x with $\mathbf{r}$. The major advantage of this method is that this method is time-reversal symmetric. This means given $x(t_1)$ and $x(t_1+h/2)$, the complete solution is determinable, but if at a later time t_2, we decide to apply this method backward, by using step size of -h, then we will retrace our steps and reach $x(t_1)$ again. This fact is not true for the RG methods! So why is this important? Consider the pendulum. When we use the RG method to solve it's equation, we found that energy drifted at every oscillation (in our example, it drifted down). Let us suppose that such drift happened in leap-frog method and the energy drifted down. Now let's apply leap-frog method backward. Since we must retrace our steps and reach the starting point, the energy must now drift upward. But here is the thing, the physics of pendulum is itself time reversal symmetric! It means that the motion is identical in forward manner as it is in backward direction. Hence, it is not possible for the variables θ, ω, to change from what they were in the forward run! Thus we can conclude that they **cannot change** in this method. This method takes care of energy conservation inherently! Now that is a reason to use it.

7.6.2 The Bulirsch-Stoer Method

This method combines two ideas i.e., the modified midpoint method and Richardson's Extrapolation. We are given an initial condition at time t and are interested in finding the solution at time $t + H$. We start by calculating a solution using the modified midpoint method and in the begining we will use just a single step for the whole solution, from t to $t + H$ i.e. our step size $h_1 = H$. This gives us an estimate we will call $R_{1,1}$.After this, we again go back to time t and repeat our calculation for $h_2 = \frac{1}{2}H$ and call this second estimate $R_{2,1}$. It is a known fact that the error on modified midpoint approach is an even function of h.

$$x(t + H) = R_{1,1} + c_1 h_1^2 + O(h_1^4) = R_{1,1} + 4c_1 h_2^2 + O(h_2^4)$$

Thus we get:

$$c_1 h_2^2 = \frac{1}{3}(R_{2,1} - R_{1,1})$$

Hence:

$$x(t + H) = R_{2,1} + \frac{1}{3}(R_{2,1} - R_{1,1}) + O(h_2^4)$$

Now we have found an estimate of $x(t + H)$, upto an accuracy of h^4. Let's call this new estimate $R_{2,2}$:

$$R_{2,2} = R_{2,1} + \frac{1}{3}(R_{2,1} - R_{1,1})$$

Taking this approach further, if we increase the steps to three, $h_3 = \frac{1}{3}H$, and solve from t to $t + H$, using modified midpoint method, call that estimate $R_{3,1}$. Then we can find:

$$R_{3,2} = R_{3,1} + \frac{4}{5}(R_{3,1} - R_{2,1})$$

With error of the order h_3^4. And once again, using logic as above:

$$x(t + H) = R_{2,2} + c_2 h_2^2 + O(h_2^6) = R_{2,2} + \frac{81}{16} c_2 h_3^2 + O(h_2^6)$$

Thus, solving for $c_2 h_3^4$ and substituting it back , we get:

$$x(t + H) = R_{3,3} + O(h_3^6)$$

with $R_{3,3} = R_{3,2} + \frac{16}{65}(R_{3,2} - R_{2,2})$ And now the error is of the order 6. We can generalise this method to a series of estimates $R_{n,2}, R_{n,3}, ..R_{n,m}$ carries an error of order h^{2m}.

$$x(t + H) = R_{n,m} + c_m h_n^{2m} + O(h_n^{2m+2})$$

Now the $R_{n-1,m}$ made with one less step satisfies:

$$x(t + H) = R_{n-1,m} + c_m h_{n-1}^{2m} + O(h_{n-1}^{2m+2})$$

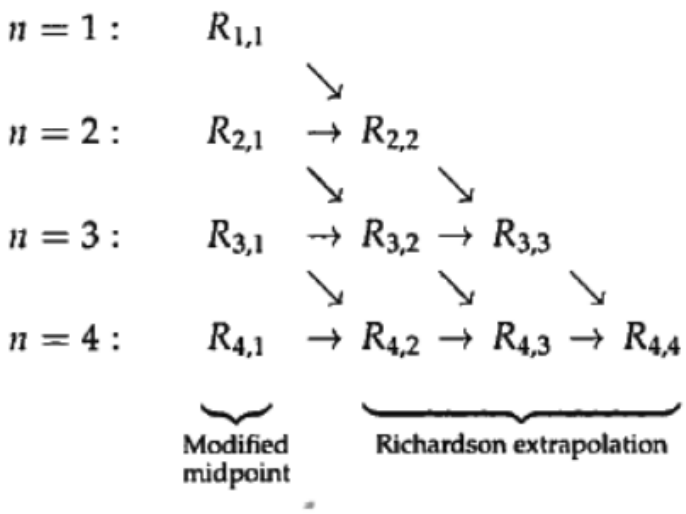


Figure 7.12: The Bulirsch-Stoer methos

But $h_n = H/n$ and $h_{n-1} = H/(n-1)$. Hence we get

$$c_m h_n^{2m} = \frac{R_{n,m} - R_{n-1,m}}{[n/(n-1)^{2m}] - 1}$$

And :

$$R_{n,m+1} = R_{n,m} + \frac{R_{n,m} - R_{n-1,m}}{[n/(n-1)^{2m}] - 1}$$

This is the heart of Bulirsch-Stoer method. fig 7.12 This method is kind of similar to the Romberg Integration method.

7.6.3 Pendulum revisited

Let's solve the non-linear pendulum again, using the Bulirsch-Stoer method:

```
from numpy import *
from pylab import *
g=9.81
l=0.1
theta0=179*pi/180
a=0.0
b=10.0
N=200  #Number of big steps
H=(b-a)/N  #Size of big steps
delta=1e-8  #required accuracy per unit time

def f(r):
```

```
    theta=r[0]
    omega=r[1]
    ftheta=omega
    fomega=-g/l*sin(theta)
    return array([ftheta,fomega],float)

#divide domain in step of big size
tp=arange(a,b,H)
thetap=[]
#r is vector
r=array([theta0,0.0],float)

#Repeat N times..
for t in tp:

    thetap.append(r[0])

    #for calc func in time range t,t+H
    #start with most basic interval, n=1
    n=1
    #Calc f(t+h) and store in R1,1
    r1=r+0.5*H*f(r)
    r2=r+H*f(r1)

    #R1 dim is [1,2] because r is vector
    R1=empty([1,2],float)
    R1[0]=0.5*(r1+r2+0.5*H*f(r2))

    #declare initial error
    error=2*H*delta

    #our initial estimate might not be good enough, so repeat with smaller step
    while error>H*delta:

        #increase step size, modify h
        n+=1
        h=H/n

        #modified midpoint for updated n
        '''
        x0=x(t)
        y1=x0+0.5*h*f(x0,t)  -->y1 is x(t+h/2)
        x1=x0+h*f(y1,t+h/2)  -->x1 is x(t+h)
        y2=y1+h*f(x1,t+h)    -->y2 is x(t+3h/2)
        x2=x1+h*f(y1,t+3h/2) -->x2 is x(t+2h)
        .
```

```
        .
        .

        All xi and yi are evaluate using derivative at the respective midpoints(
        '''
        r1=r+0.5*h*f(r)
        r2=r+h*f(r1)
        for i in range(n-1):
            r1+=h*f(r2)
            r2+=h*f(r1)

        #storing R1 for further use
        R2=R1
        #defining R1 with one more row than previous
        R1=empty([n,2],float)

        #first element is just modified midpoint method
        R1[0]=0.5*(r1+r2+0.5*h*f(r2))

        #for subsequent, use the trick..
        for m in range(1,n):
            epsilon=(R1[m-1]-R2[m-1])/((n/(n-1))**(2*m)-1)
            R1[m]=R1[m-1]+epsilon

        #epsilon[0] gives error on theta
        error=abs(epsilon[0])

    #once req accuracy reached, get out of loop, finalise r as the last element
    r=R1[n-1]

    #repeat for other t's

plot(tp,thetap)
show()
```

And the output is shown in figure 7.13

7.7 Boundary value problems

There can be differential equations, where the given information is a boundary value. Eg. consider the equation governing the height of the ball above the ground in free fall:

$$\frac{d^2x}{dt^2} = -g$$

The two initial conditions could be given as (i) The start height and velocity or (ii) one initial and one ending condition, eg. $x = 0, t = 0$ and $x = 0, t = t1$.

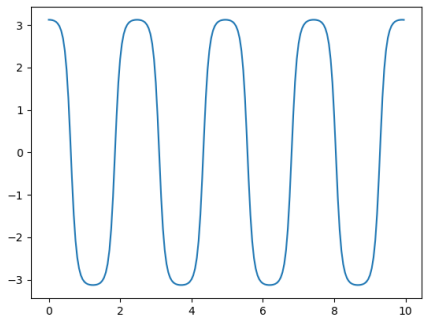


Figure 7.13: Solving the pendulum using Bulirsch-Stoer method

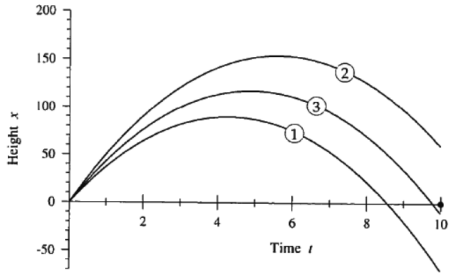


Figure 8.11: The shooting method. The shooting method allows us to match boundary conditions at the beginning and end of a solution. In this example we are solving for the trajectory of a thrown ball and require that its height x be zero—i.e., that it lands—at time $t = 10$. The shooting method involves making a guess as to the initial conditions that will achieve this. In this example, we undershoot our target (represented by the dot on the right) on our first guess (trajectory 1). On the second guess we overshoot. On the third we are closer, but still not perfect.

Figure 7.14: The shooting method

This is a boundary value problem.

7.7.1 Shooting method

Solving a DE using boundary value problems is tougher, hence here, we will try to guess the intial condition, which satisfies the boundary values provided. Eg. we will guess an initial velocity of the ball, and solve the equation to find the height at time t_1. If it doesn't satisfy the boundary condition, we modify our guess as follows: We have (in principle) a function $x = f(v)$ which gives us the final height given an initial velocity, and we have to find it's root!. This should be easy. The method is described in fig:7.14

Chapter 8

Partial Differential Equations

Most of the interesting equations in physics are Partial Differential Equations (PDE). For PDEs, it turns out it is easier to solve Boundary Value Problems. A classic example of this is the problem of solving electric potential in electrostatics. e.g. we have an empty box, whose one wall is at voltage V and the others at zero. We want to determine electrostatic potential within the box. We know from the maxwell's equations :

$$\Delta^2\phi = 0$$

This is Laplace's equation. We want to solve this equation with the boundary conditions $\phi = V$ on the top wall and zero otherwise.

8.1 BVP the relaxation method

The fundamental technique for solving PDE is the method of finite differences. Consider the Laplace's equation in 2 dimensions. In the method of finite differences we divide the space into a grid of discrete points (most common being a square grid). Let the spacing between the points be a, then :

$$\frac{\delta^2\phi}{\delta x^2} = \frac{\phi(x+a,y) + \phi(x-a,y) - 2\phi(x,y)}{a^2}$$

$$\frac{\delta^2\phi}{\delta y^2} = \frac{\phi(x,y+a) + \phi(x,y-a) - 2\phi(x,y)}{a^2}$$

And the laplacian operator in 2d becomes:

$$\frac{\delta^2\phi}{\delta x^2} + \frac{\delta^2\phi}{\delta y^2} = \frac{\phi(x+a,y) + \phi(x-a,y) - 2\phi(x,y) + \phi(x,y+a) + \phi(x,y-a) - 2\phi(x,y)}{a^2}$$

In other words, we add the values of ϕ adjacent to (x, y) and subtract 4 times the values at (x, y) and divide by a^2. Hence to solve this, we create an array which stores the values of ϕ.

8.2 Gauss-Seidal Method

We initialize the array using the boundary conditions given to us, and then continuously update the array using the rule:

$$\phi(x, y) = \frac{1}{4}[\phi(x + a, y) + \phi(x - a, y) + \phi(x, y + a) + \phi(x, y - a)]$$

8.2.1 Over-relaxation

Generally, we use this method, together with Gauss-Siedel method to get to the answer quickly.

$$\phi'(x, y) = \phi(x, y) + \Delta\phi(x, y)$$

We define the over-relaxed value $\phi_\omega(x, y)$ by:

$$\phi_\omega(x, y) = \phi(x, y) + (1 + \omega)\Delta\phi(x, y)$$

Hence, when clubbed with Gauss-Siedel method:

$$\phi(x, y) = \frac{1 + \omega}{4}[\phi(x + a, y) + \phi(x - a, y) + \phi(x, y + a) + \phi(x, y - a)] - \omega\phi(x, y)$$

Moreover, Gauss-Siedel method is numerically stable, so it will surely get you an answer. Usually, a value of $\omega = 0.9$ is optimal, for fastest calculations.

8.2.2 Example

Let's solve the potential in the empty box:

```
from numpy import *
from pylab import imshow,gray,show
#constants
L=1
M=100
V=1
a=L/M
delta=1e-6
w=0.91

def compare(a,b):
    if a>=b:
        return a
    else:
```

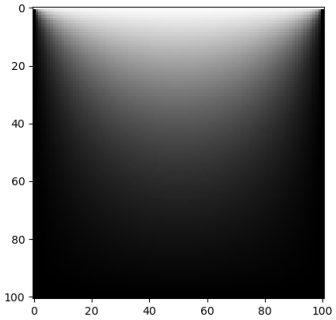


Figure 8.1: Electric box

```
        return b

phi=zeros([M+1,M+1],float)
phi[0,:]=V
max_diff=2*delta
while max_diff>delta:
    print(max_diff)
    max_diff=0.0
    for i in range(0,M+1):
        for j in range(0,M+1):
            if i==0 or i==M or j==0 or j==M:
                continue
            else:
                old=phi[i,j]
                new=(1+w)/4*(phi[i+1,j]+phi[i,j+1]+phi[i-1,j]+phi[i,j-1])-w*phi[
                phi[i,j]=new
                max_diff=compare(max_diff,abs(new-old))

imshow(phi)
gray()
show()
```

And the output is shown in fig: 8.1

8.2.3 Capacitors in a box:

Consider the arrangement shown in fig 8.2 And we are interested in finding the potential within the box. The following code does it:

```
from numpy import *
from pylab import *
```

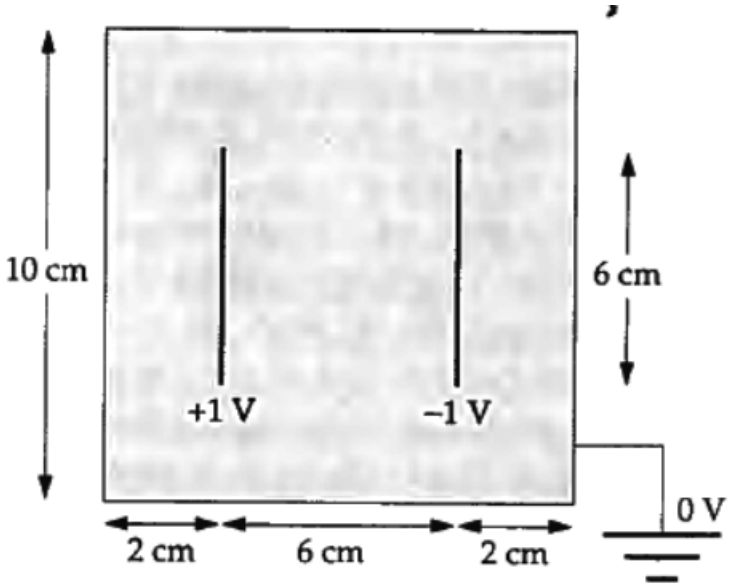


Figure 8.2: Capacitors are placed in a metal box

```
#constants
L=0.1
l=0.06
M=200
a=L/M
x1=0.02  #x coord of capacitior 1
x2=0.08
y10=0.02 #start of capacitor
y11=0.08 #end of capacitor
y20=0.02
y21=0.08

phi=zeros([M+1,M+1],float)
i1=int(x1/a)
i2=int(x2/a)
j10=int(y10//a)
j11=int(y11//a)
j20=int(y20//a)
j21=int(y21//a)
phi[i1,j10:j11]=1
phi[i2,j20:j21]=-1
w=0.9
delta=1e-6
```

```
max_diff=2*delta
def compare(a,b):
    if a>=b:
        return a

    else:
        return b

while max_diff>delta:
    print(max_diff)
    max_diff=0.0
    for i in range(1,M):
        for j in range(1,M):
            if i==i1 and j>j10 and j<j11:
                continue
            elif i==i2 and j>j20 and j<j21:
                continue
            else:
                old=phi[i,j]
                phi[i,j]=(1+w)/4*(phi[i+1,j]+phi[i,j+1]+phi[i-1,j]+phi[i,j-1])-w
                max_diff=compare(max_diff,abs(phi[i,j]-old))

imshow(phi)
gray()
show()
```

And the output is shown in fig 8.3

8.3 Initial Value Problems

In this type of problems, we are given an initial condition and are required to predict the future. A simple example is the diffusion equation:

$$\frac{\delta\phi}{\delta t} = D\frac{\delta^2\phi}{\delta x^2}$$

$\phi(x,t)$ depends on both x,t and so this is a partial differential equation. The previously applied approach doesn't work here, as we generally have a boundary condition in x, and an initial condition in t(we know where the value of x starts but not where it ends).

8.3.1 FTCS method

In the Forward Integration Method, we start by dividing the spatial dimension(s) into a grid. In the case of one spatial dimension, it reduces to a line. We divide it into evely spaced points with spacing a.Then, in the diffusion equation,

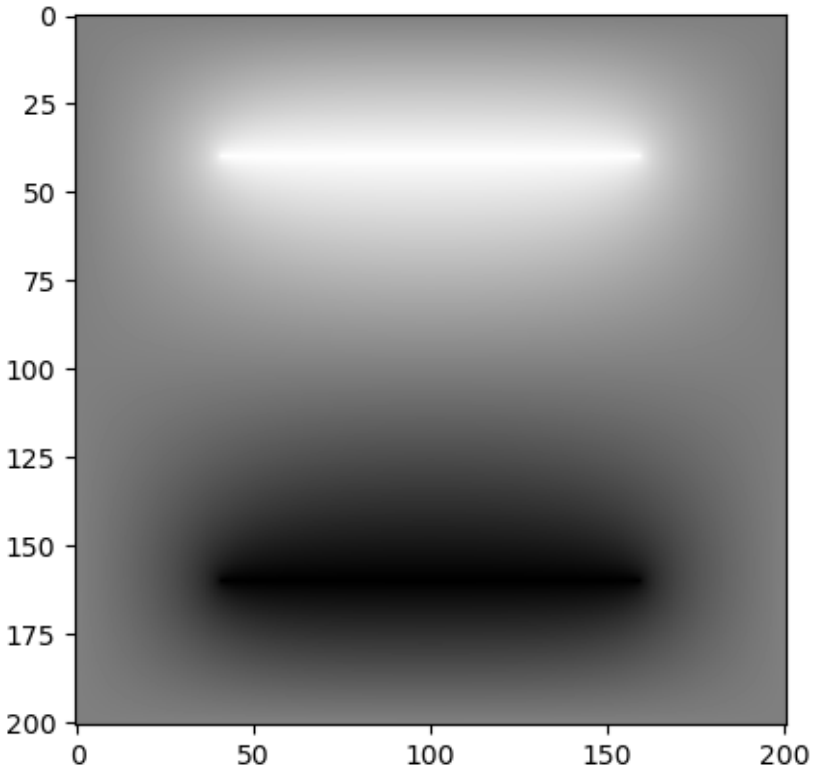


Figure 8.3: Potential of the system

the RHS becomes:

$$\frac{\delta^2\phi}{\delta x^2} = \frac{\phi(x+a,t) + \phi(x-a,t) - 2\phi(x,t)}{a^2}$$

And the whole equation becomes:

$$\frac{\delta\phi}{\delta t} = \frac{D}{a^2}[\phi(x+a,t) + \phi(x-a,t) - 2\phi(x,t)]$$

If we think of ϕ at different points as different variables, then we have a set of simultaneous ODEs, with the number of equations ranging from a thousand to a few millions, depending on how fine a grid we use.
To solve these equations, we will use Euler's method. YES, Euler's method. This is because, when we approximate the second derivative, we already introduce an error of order h^2, so now it's of no use to apply more accurate methods like Runge-Kutta method. Hence, we have:

$$\phi(x,t+h) = \phi(x,t) + h\frac{D}{a^2}[\phi(x+a,t) + \phi(x-a,t) - 2\phi(x,t)]$$

Thus, if we know the value of every grid point at some time t, we can calculate it at any later time. This method is call the *forward-time centered-space method*.

Thermal Diffusion in Earth's crust

This is an example of a diffusion problem with time varying boundary condition. The mean daily temperature at the Earth's surface is:

$$T_0(t) = A + B\sin\frac{2\pi t}{\tau}$$

where τ = 365 days, A=10, B=12 degrees Celsius. Given that the temperature at a depth of 20 m is a constant 11 degrees, and the diffusivity of Earth is=$0.1m^2day^-1$. Following is the program to solve:

```
from numpy import *
from pylab import *

#constants
A=10
B=12
T=365
L=20
D=0.1
M=100  #no. of points
a=L/M  #grid spacing
def T0(t):
    return A+B*sin(2*pi*t/T)
```

```
#plot times
t1=31*3
t2=31*6
t3=31*9
t4=364

Temp=empty(M+1,float)
Temp[M]=11
Temp[1:M]=10
Tempp=empty(M+1,float)
Tempp[M]=11
h=0.1
epsilon=h/1000
c=h*D/a**2
time=arange(0,365*9,h)
timeplot=arange(0,365,h)
for t in time:
    Temp[0]=T0(t)
    Tempp[0]=T0(t)
    Tempp[1:M]=Temp[1:M]+c*(Temp[2:M+1]+Temp[0:M-1]-2*Temp[1:M])

    Temp,Tempp=Tempp,Temp

for t in timeplot:
    Temp[0]=T0(t)
    Tempp[0]=T0(t)
    Tempp[1:M]=Temp[1:M]+c*(Temp[2:M+1]+Temp[0:M-1]-2*Temp[1:M])

    Temp,Tempp=Tempp,Temp
    if abs(t-t1)<epsilon:
        plot(Temp,label="3m")
    if abs(t-t2)<epsilon:
        plot(Temp,label="6m")
    if abs(t-t3)<epsilon:
        plot(Temp,label="9m")
    if abs(t-t4)<epsilon:
        plot(Temp,label="12m")
xlabel("x")
ylabel("temp")
legend()
show()
```

And, this program outputs the variation of temperature with depth at intervals of 3 months (fig. 8.4)

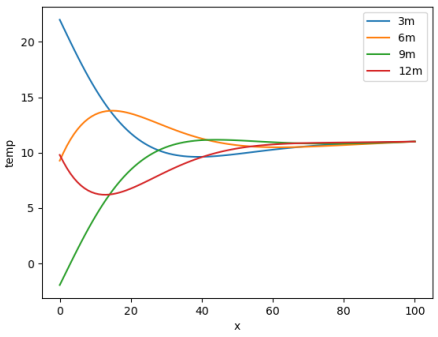


Figure 8.4: Temperature variation in Earth's crust at intervals of 4 months

8.3.2 Numerical Stability

Sadly, the FTCS doesn't work on one of the most useful DE in physics, the wave equation! The 1 d wave equation is:

$$\frac{\delta^2\phi}{\delta t^2} = v^2\frac{\delta^2\phi}{\delta x^2}$$

If we try to apply the method to this equation, we end up with a second order ODE, which, when split to 2 simultaneous ODEs:

$$\phi(x, t+h) = \phi(x,t) + h\psi(x,t)$$

$$\psi(x, t+h) = \psi(x,t) + h\frac{v^2}{a^2}[\phi(x+a,t) + \phi(x-a,t) - 2\phi(x,t)]$$

These can be solved, but in practice, when we do so, the solution becomes chaotic, and loses meaning! This calculation is called numerically unstable. Let's return to the basic form of FTCS for diffusion:

$$\phi(x, t+h) = \phi(x,t) + h\frac{D}{a^2}[\phi(x+a,t) + \phi(x-a,t) - 2\phi(x,t)]$$

We can perform von Newman analysis on this equation. This analysis judges whether an equation is numerically stable or not. In this analysis, we write the spatial variation $\phi(x,t)$ as a Fourier Series:

$$\phi(x,t) = \sum_k c_k(t)\exp ikx$$

for some wave vector k and coefficients $c_k(t)$.Using the linearity of the DE, we study a single term of this expansion. By plugging it in and manipulating it, we find:

$$c_k(t+h) = \left[1 - h\frac{4D}{a^2}\sin^2 ka/2\right]c_k(t)$$

The coefficients just get multiplied by the brackets. Now, for stability, the terms in the bracket must be less than unity for any k. Luckily in this case, it is always less than 1. However, the magnitude could exceed 1 if it fell below -1.

$$h \leq \frac{a^2}{2D}$$

when h is larger than this, solution can diverge. We can make sure to avoid this to maintain stability. Such analysis can only be performed if the DE is linear. If we apply the von Newman analysis to the wave-equation for FTCS, it turns out that the coefficients are always greater than 1. Hence the unstability.

8.3.3 The implcit and Crank-Nicholson Method

In the implicit method, we make the substitution h $\rightarrow -h, in the equations of FTCS for waves, giving:$ $\phi(x, t-h) = \phi(x,t) - h\psi(x,t)$

$$\psi(x, t-h) = \psi(x,t) - h\frac{v^2}{a^2}[\phi(x+a,t) + \phi(x-a,t) - 2\phi(x,t)]$$

Simiarly, if we make a second substitution, t $\rightarrow t+h, and rearrange:$ $\phi(x, t+h) - h\psi(x,t+h) = \phi(x,t)$

$$\psi(x, t+h) - h\frac{v^2}{a^2}[\phi(x+a,t+h) + \phi(x-a,t+h) - 2\phi(x,t+h)] = \psi(x,t)$$

These equations give us value of ϕ and ψ at t+h but implicitly. The advantage is that, these solutions are numerically stable, but turns out, this method creates an exponential decay for the waves, all the solutions of this methods tend to zero at large values of t. A common ground between this method and FTCS is the Crank-Nicolson method: All we do is take average of the equations given by the two methods!

$$\phi(x, t+h) - \frac{1}{2}\psi(x,t+h) = \phi(x,t) + \frac{1}{2}\psi(x,t)$$

$$\psi(x,t+h) - h\frac{v^2}{2a^2}[\phi(x+a,t+h) + \phi(x-a,t+h) - 2\phi(x,t+h)] = \psi(x,t) + h\frac{v^2}{2a^2}[\phi(x+a,t) + \phi(x-a,t) - 2\phi(x,t)]$$

Analysing this shows that the coefficients turn out to be exactly 1. This is also numerically stable in a sense.

8.3.4 Spectral methods

Apart from finite difference methods, some methods exist which are faster, more stable, and more accurate, but at the cost of increased complexity in terms of programming. One such is the finite element method. However the vastness of this method can't be contained in this little report. However, there exists another such method, called the Spectral method (or the fourier transform

method), which we can explore right now.
Consider the equation of a wave on a string, fixed at both ends. Consider the trial solution:

$$\phi_k(x,t) = \sin\frac{\pi kx}{L}e^{i\omega t}$$

And we take the real part of the RHS, assuming ϕ to be real. So long as k is an integer,the equation satisfies the boundary conditions. Substituting this into the wave equation, we find:

$$\omega = \frac{\pi vk}{L}$$

Now, we break the string into N equal intervals, bounded by N+1 grid points, both of which have $\phi = 0$. The position of points are:

$$x_n = \frac{n}{N}L$$

and the solutions at these points are:

$$\phi_k(x_n,t) = \sin\frac{\pi kx}{L}e^{i\frac{\pi vkt}{L}}$$

Since the wave equation is linear, any linear combination of these solutions is also a solution at a point:

$$\phi(x_n,t) = \frac{1}{N}\sum_{k=1}^{N-1} b_k \sin\frac{\pi kx}{L}e^{i\frac{\pi vkt}{L}}$$

This is a solution for any choice of coefficients, which may be complex. At t=0, the real part of the equation is:

$$\phi(x_n,0) = \frac{1}{N}\sum_{k=1}^{N-1} \alpha_k \sin\frac{\pi kx}{L}$$

where α is the real part of b_k. It is clear that this equation (and it's time derivative) is the standard Fourier sine series, which can represent any set of samples $\phi(x_n)$. Hence, given the value of $\phi(x,0)$ and it's time derivative at $t = 0$, the real and imaginary parts of the coefficients b_k are fixed, hence the entire solution can be determined. The nice feature about this is that, for calculating the coefficient, special numerical methods exists, like the fast fourier transform and the discrete fourier transforms. Moreover, notice that in this method, we don't need to step through every time interval to reach the required one. We can directly calculate it! This saves time and efforts. This method only works for the problems with simple boundary conditions like $\phi = 0$ at the ends or box shaped conditions. Moreover, they only work for the DEs which are linear.

8.4 Examples

8.4.1 Relaxation for ODE

Since ODE is just a special case of PDE, we can apply the relaxation methods to ODE. Consider the boundary value problem of the ball thrown in the sky, while we are givem $x = 0$ at $t = 0$ and $t = 10$. The differential equation was:

$$\frac{d^2x}{dt^2} = -g$$

Here is the code to solve for it:

```
from numpy import *
from pylab import plot,show,xlabel,ylabel
#constants
g=9.81
a=0.0
b=10.0
#divide time in 100 steps
N=100
h=(b-a)/N
#array to store 101 points,start and end included
x=zeros([N+1],float)
time=linspace(a,b,N+1)
delta=1e-6
max_diff= 2*delta
def max_val(a,b):
    if a>=b:
        return a
    else:
        return b

#over relaxation factor
'''
x'=x+delta(x)
=>x''=x+(1+w)*delta(x)
=>x''=x+(1+w)*(x'-x)
'''
w=0.9

while max_diff>delta:
    print(max_diff)
    #reset max diff everytime
    max_diff=0.0
    for i in range(N):
        #maintain boundary value
        if i==0:
```

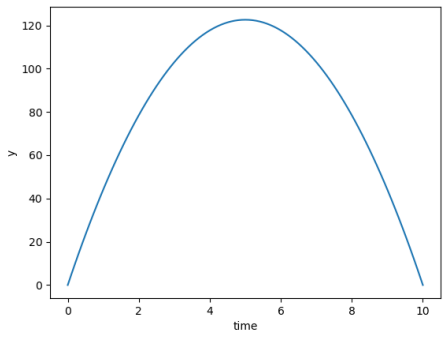


Figure 8.5: Trajectory of a ball, given the boundary conditions

```
            continue
        else:
            old=x[i]
            #update grid point using over-relaxation
            x[i]=(1+w)*(g*h**2+x[i-1]+x[i+1])/2-w*x[i]
            #At each i, ensure max_diff countains value of largest difference be
            max_diff=max_val(max_diff,abs(x[i]-old))

plot(time,x)
xlabel("time")
ylabel("y")
show()
```

And the output is shown in fig: 8.5

8.4.2 Schrodinger equation and the Crank-Nicholson method

The schrodinger equation in 1D is:

$$-\frac{\hbar}{2M}\frac{\delta^2\psi}{\delta x} = i\hbar\frac{\delta\psi}{\delta t}$$

Consider the boundary conditions that the particle is in an impenetrable box of length L. We divide the line from $x = 0$ to $x = L$ into N points and conduct the method of Crank and Nicholson:

```
from numpy import *
from cmath import exp
from pylab import *
from scipy.constants import hbar,m_e
from numpy.linalg import solve
#constants
```

```
L=1e-8
N=1000
a=L/N
h=1e-18
a1=1+h*(1j*hbar/(2*m_e*a**2))
a2=-h*(1j*hbar/(4*m_e*a**2))
b1=1-h*(1j*hbar/(2*m_e*a**2))
b2=-a2

B=zeros([N,N],complex)
for i in range(N):
    if i==0:
        B[i,i]=b1
        B[i,i+1]=b2
    elif i==N-1:
        B[i-1,i]=b2
        B[i,i]=b1
    else:
        B[i,i-1]=b2
        B[i,i]=b1
        B[i,i+1]=b2

A=zeros([N,N],complex)
for i in range(N):
    if i==0:
        A[i,i]=a1
        A[i,i+1]=a2
    elif i==N-1:
        A[i-1,i]=a2
        A[i,i]=a1
    else:
        A[i,i-1]=a2
        A[i,i]=a1
        A[i,i+1]=a2

def psi0(x):
    x0=L/2
    sigma=1e-10
    k=5e10
    z=-(x-x0)**2/(2*sigma**2)
    iz=1j*k*x
    return exp(z)*exp(iz)

psi=empty(N,float)
x=linspace(0,L,N)
psi[:]=psi0(x)
```

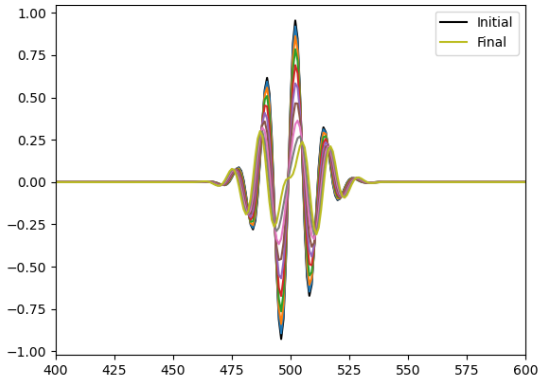


Figure 8.6: Time-evolution of wavefunction of particle in a box

```
def Bcrosspsi(psi):
    v=empty(N,complex)
    for i in range(N):
        if i==0:
            v[i]=b1*psi[i]+b2*psi[i+1]
        elif i==N-1:
            v[i]=b2*psi[i-1]+b1*psi[i]
        else:
            v[i]=b1*psi[i]+b2*(psi[i+1]+psi[i-1])
    return v
for i in range(10):
    v=Bcrosspsi(psi)
    psi=solve(A,v)
    if i==0:
        plot(psi,"k",label="Initial")
    elif i==9:
        plot(psi,label="Final")
    else:
        plot(psi)
xlim((400,600))
legend()
show()
```

This program plots the wavefunction for 10 times steps. The output is in figure 8.6

Chapter 9

Random numbers And Monte-Carlo methods

In nature, we often encounter random events, like the decay of radioactive elements, and also some events like Brownian motion, which aren't actually random. Given the position and velocities of each particle, the entire motion is deterministic (classically). But this motion can be considered random due to the large number of variables and equations. For such analysis with randomness, we can use Monte-Carlo methods.

9.1 Decay of isotopes

On average, we know the number N of atoms in a sample will follow:

$$N(t) = N(0)2^{-t/\tau}$$

Or, the probability that an atom decays in time t is:

$$p(t) = 1 - 2^{-t/\tau}$$

Now consider the chain reaction in the fig 9.1 We will start with a sample of 10000 atoms of ^{213}Bi and simulate the decay

```
from random import random
from numpy import *
from math import exp
from pylab import *
from tqdm import tqdm
#constant
NBi=10000
NPb=0
NTl=0
```

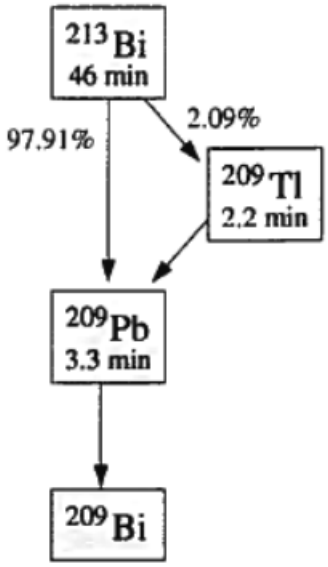


Figure 9.1: Chain reaction of radioactive decay

```
NBif=0
Bi=[]
Pb=[]
Tl=[]
Bif=[]
dt=1
pdBi=1-exp(-dt/(46*60))
pBitoPb=0.9791
pBitoTl=1-pBitoPb
pdPb=1-exp(-dt/(3.3*60))
pdTl=1-exp(-dt/(2.2*60))

tfinal=20000
time=arange(0,tfinal,dt)
for t in tqdm(time):
    #update lists
    Bi.append(NBi)
    Bif.append(NBif)
    Pb.append(NPb)
    Tl.append(NTl)

    #Pb decay
    decay=0
```

```
    for i in range(NPb):
        if random()<pdPb:
            decay+=1
    NPb-=decay
    NBif+=decay

    #Tl
    decay=0
    for i in range(NTl):
        if random()<pdTl:
            decay+=1
    NTl-=decay
    NPb+=decay

    #Bi
    decay=0
    decaypb=0
    decaytl=0
    for i in range(NBi):
        if random()<pdBi:
            decay+=1
    NBi-=decay
    for j in range(decay):
        if random()<pBitoPb:
            decaypb+=1
        else:
            decaytl+=1
    NPb+=decaypb
    NTl+=decaytl
plot(time,Bi,label="Bismuth 213 remaining")
plot(time,Pb,label="Lead")
plot(time,Tl,label="Thallium")
plot(time,Bif,label="Bismuth 209 formed")
ylabel("Number of atoms")
xlabel("time in seconds")
legend()
show()
```

And the output is shown in fig 9.2

9.1.1 Brownian Motion

Following is the code to animate the brownian motion of a particle in a box in 2d:

```
from numpy import empty, array,zeros
from pylab import imshow,show,gray
```

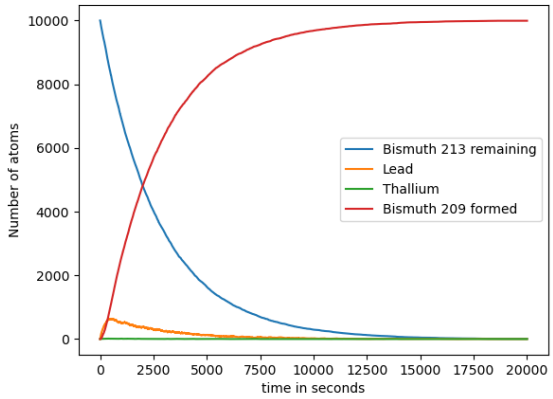


Figure 9.2: Radioactive decay

```
from vpython import simple_sphere,vector,rate,color,canvas
from random import random
#const
L=101
scene2=canvas(width=L,height=L,title="Brownian Motion")

s=simple_sphere(pos=vector(50,50,0),radius=1,color=color.red)
lattice=zeros([L,L],int)
i=50
j=50
N=100000
for t in range(N):
    lattice[i,j]=1
    s.pos=vector(i-50,j-50,0)
    rate(20)
    p=random()
    if p<0.25:
        if i!=L-1:
            i+=1
    elif p<0.5:
        if i!=0:
            i-=1
    elif p<0.75:
        if j!=L-1:
            j+=1
    else:
        if j!=0:
            j-=1
```

```
imshow(lattice)
gray()
show()
```

I can't show the output, but hey, you can try out this code in your terminal.

9.2 Non-uniform Random numbers

The probability that a particle decays in a small time interval dt is:

$$1 - 2^{-dt/\tau} \approx \frac{ln2}{\tau} dt$$

And that it doesn't decay until t is $2^{-t/\tau}$ Hence the probability that it decays between t and t+dt is:

$$P(t)dt = 2^{-t/\tau} \frac{ln2}{\tau} dt$$

A way to simulate this decay is to draw random numbers from this non-uniform distribution to represent the time at which each atom decays. Now, we will use the transformation method to choose a random number from this distribution. Suppose we have a random number z from the distribution $q(z)$, meaning the probability of generating the random number between z and z+dz is q(z)dz. Suppose you have a function x=x(z) then x is also a random number. Now if we want x to follow the distribution p(x), then we do the following:
The probability of generating value of x between x and x+dx is by definition equal to probability of generating value of z between z and z+dz :

$$p(x)dx = q(z)dz$$

If q(z)=1:

$$\int_{-\infty}^{x(z)} p(x')dx' = \int_{0}^{z} dz' = z$$

If we solve this equation for the decay case, we get:

$$x = -\frac{1}{\mu} ln(1 - z)$$

Now all we need to do is generate uniform random numbers, and feed them into this formula.

9.3 Monte Carlo Integration

Consider a pond in a farm. Given that we know the area of the pond and the area of the farm, we can calculate the probability that a rock thrown randomly in the farm falls in the pond. Now consider this: given the probability that a rock falls in the pond, and the area of the farm, we can calculate the area of the

pond! This is a very powerful result, and the basis of Monte-Carlo Integration. Let's say that we want to evaluate the integral:

$$I = \int_0^2 \sin^2 \frac{1}{x(2-x)} dx$$

We will pick 2 random numbers in the ranges (0,2) and (0,1) respectively, and call them the point (x,y). Then we will find the probability that the point is below the curve in the grid, and hence evaluate the integral:

```
import random
from numpy import *
def f(x):
    return sin(1/(x*(2-x)))**2


N=10000
count=0
for i in range(N):
    x=2*random.random()
    y=random.random()
    if y<f(x):
        count+=1
I=2*count/N
print("Integral is: ",I)
error=sqrt(I*(2-I)/N)
print("Error is: ",error)
```

Output is:

```
Integral is:  1.435
Error is:  0.00900430452616969
```

Where we have used the formula:

$$\sigma = \sqrt{vark}\frac{A}{N} = \sqrt{\frac{I(A-I)}{N}}$$

9.4 Mean Value Method

The average value of $f(x)$ is defined by:

$$avg(f) = \frac{1}{b-1}\int_a^b f(x)dx = \frac{I}{b-a}$$

Hence we can estimate I by calculating the average of the function .

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

It can be shown that this method is always more accurate than the previous method, for a given number of sample points.
Another advantage is that this methods can be easily extended to multiple dimensions:

$$I \approx \frac{V}{N} \sum_{i=1}^{N} f(\mathbf{r_i})$$

where $\mathbf{r_i}$ are picked randomly from the volume V.

Volume of a Hypersphere

The following program calculates the volume of a sphere in 10D, using the formula:

$$I \approx \frac{2^n}{N} \sum_{1}^{N} f(\mathbf{r_i})$$

```
from numpy import *
from numpy.random import random
from tqdm import tqdm
d=10
def f(r):
    if dot(r,r)<=1:
        return 1
    else:
        return 0

N=300000
s=0.0
for i in tqdm(range(N)):
    r=random(d)
    r=2*r-1
    s+=f(r)

I=2**d/N*s
print(I)
```

And the output is: 2.409813333333333 Notice that using the traditional techniques, if we only used 100 points per axis, we would require a total of 100^10 points, which is at all a feasible calculation. This method does it reasonably well with just a million points or so!

9.5 Importance Sampling

This method gives us a problem in cases when the integrand diverges, because those points might come up in the random selection and the value of the integrand at them is very large, thus causing the sum to diverge(in some runs and

not in others). To solve this problem, we choose the points x_i non-uniformly from the interval.
For any function $g(x)$, the weighted average is:

$$\langle g \rangle_w = \frac{\int_a^b w(x)g(x)dx}{\int_a^b w(x)dx}$$

Consider the integral:

$$I = \int_a^b f(x)dx$$

setting $g(x) = f(x)/w(x)$:

$$\left\langle \frac{f(x)}{w(x)} \right\rangle_w = \frac{\int_a^b w(x)g(x)dx}{\int_a^b w(x)dx} = \frac{\int_a^b f(x)dx}{\int_a^b w(x)dx}$$

Or:

$$I = \left\langle \frac{f(x)}{w(x)} \right\rangle_w \int_a^b w(x)dx$$

This formula allows us to calculate the integral from a weighted, rather than the standard, average. To calculate the weighted average, let's define the probability distribution:

$$p(x) = \frac{w(x)}{\int_a^b w(x)dx}$$

This is like a normalized weight function. If we sample N random points nonuniformly with this density, the probability of generating a value in the interval between x and $x + dx$ is $p(x)dx$ and the average number of points in this range is $Np(x)dx$. Hence:

$$\sum_{i=1}^{N} g(x_i) \approx \int_a^b Np(x)g(x)dx$$

Now, the weighted average becomes:

$$\langle g \rangle_w = \frac{\int_a^b w(x)g(x)dx}{\int_a^b w(x)dx} = \int_a^b p(x)g(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} g(x_i)$$

Where x_i are chosen from the distribution. Now, we have found the weighted average, and substituting into the origianl equation:

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

By choosing the $w(x_i)$, we can get rid of the pathologies.

9.6 Monte Carlo Simulation

These are any simulations run on computers, which use random numbers to simulate a random physical process. Here is an analysis in statistical mechanics:

9.6.1 Importance sampling and statistical mechanics

The fundamental problem of statistical mechanics is to calculate the average value of a quantity of interest in a physical system in thermal equilibrium at temperature T. All we know is that the system will pass through a succession of states of energy E_i such that :

$$P(E_i) = \frac{e^{-\beta E_i}}{Z}, Z = \sum_i e^{-\beta E_i}$$

where $\beta = \frac{1}{k_B T}$ And the average value of the quantity X is :

$$\langle X \rangle = \sum_i X_i P(E_i)$$

Usually, we can't calculate the sum directly, as the number of states within a system is too large(consider a gas with 10^{23} molecules, and each molecule can be in 2 states!). We will do the sum, by choosing N terms randomly from the sum, and calculating:

$$\langle X \rangle \approx \frac{\sum_{k=1}^{N} X_k P(E_k)}{\sum_{k=1}^{N} P(E_k)}$$

However, notice that for most of the states $E_i >> k_B T$, so they will make negligible contributions to the sum. If most of such points are chosen in the random selection, they lead lead to erroneous results. To solve this, we use importance sampling. We will select points from a distribution which will let us choose maximum points from the region which is important for the sum. To do this, we will set $g_i = X_i P(E_i)/w_i$ which will give (using the definition of weighted average):

$$\langle X \rangle = \left\langle \frac{X_i P(E_i)}{w_i} \right\rangle_w \sum_i w_i$$

Once again, we have to find the weighted average, and reusing arguments similar to above, we will define the probability distribution:

$$p_i = \frac{w_i}{\sum_j w_j}$$

and choose N random points from this distribution, then :

$$\langle g \rangle_w \approx \frac{1}{N} \sum_{k=1}^{N} g(x_k)$$

Then we get finally:

$$\langle X \rangle \approx \frac{1}{N} \sum_{k=1}^{N} \frac{X_k P(E_k)}{w_k} \sum_i w_i$$

Here the first sum is over states k while the second is over all the states i (this is calculated analytically). We have to choose a w_i such that we pick up the big terms and ignore the small ones, but also the sum $\sum_i w_i$ is easy to perform. Luckily, we know such a distribution! Putting $w_i = P(E_i)$, the sum reduces to 1, by definition! Also, the equation mentioned above reduces to:

$$\langle X \rangle \approx \frac{1}{N} \sum_{k=1}^{N} X_k$$

We just chose the N states in proportion to their Boltzman probabilities, and the problems were solved! A more physical way to interpret this result is: Our system evolves through states with probabilities $P(E_i)$ and the average of X is defined by the average values of the states the system passes through. We just did that in the more rigorous analysis which preceded!

9.6.2 Markov Chain method

Unfortunately, we are not done.If we try to pick states with probability $P(E_i)$, in it's formula, we encounter the denominator which includes Z (the partition function) which is precisely the sum we swore to avoid! We can do it without the partition funtion, by using the Markov chain method as follows.
We want to choose a set of states for the sum in our final expression:

$$\langle X \rangle \approx \frac{1}{N} \sum_{k=1}^{N} X_k$$

To do so, we generate states based on the previous chain. Let the previous state be denoted by i. For the new state new, instead of choosing randomly from all the states, let's make a small change to the current state (for gases, maybe change the quantum state of one atom). Now the probability of the new state is determined by a set of *transition probabilities* T_{ij} that denotes the probability of changing from state i to state j. Choosing the right T_{ij} will allow is to make the probability of visiting any state $E_i = P(E_i)$(the boltzman probability). Then all we will have to do is move through states, measure the quantity X in each state, and take the average.
Since we must always end up in some state at each move:

$$\sum_j T_{ij} = 1$$

They must also satisfy:

$$\frac{T_{ij}}{T_{ji}} = \frac{P(E_j)}{P(E_i)} = e^{-\beta(E_j - E_i)}$$

Assuming that we found $T'_{ij}s$ that satisfy the above mentioned constraints for the state i, then the probability of being in state j on the next state is the sum of getting there from every other possible state i:

$$\sum_i T_{ij} P(E_i) = \sum_j T_{ji} P(E_j) = P(E_j)$$

So, if we have correct probabilities for being in every state on one step, then we have the correct probabilities for every step! It can be proven that, no matter which distribution you start with, if run the markov chain long enough, you will end up with the boltzman distribution! To find T_{ij}, we will use the *Metropolis Algorithm*. Note that we are allowed to revisit a state in the chain, and also allowed to state for consecutive steps, i.e. $T_{ii} \neq 0$. Suppose again that the previous state was i. Now we choose a change to this state uniformly at random from a set of possible actions, called the move set.Eg. in case of a gas, we will randomly choose a molecule and randomly increase or decrease it's energy by 1. However, this in itself doesn't satisfy the constraints but if we accept a move with an acceptance probability P_a given by:

$$P_a = \begin{cases} 1 & \text{if } E_j \leq E_i, \\ e^{-\beta(E_j - E_i)} & \text{if } E_j > E_i, \end{cases}$$

All this does is, if the energy is decreasing due to a move, we certainly accept it. If it increase energy, we accept it with the probability given above. Under this scheme, the probability of making a move from i to j, T_{ij} is

$$T_{ij} = \frac{1}{M} P_a$$

Where M is just the the total number of possibilities we have for choosing the move. Notice that this definition of T_{ij} satisfies both the constraints.
To summarise the complete Markov chain Monte Carlo simulation:

- Choose a random starting state.
- Choose a move uniformly at random from the set of available moves (like changing the energy of one molecule).
- Calculate the acceptance probability.
- With probability P_a accept the move (make the change), otherwise reject it (let the system be in the state it is currently in, for one more move).
- Measure the desired quantity X and add the value to a running sum.
- Repeat from step 2.

A great way to implement the part with acceptance probability is, to generate a random number z, and accept the move if $z < e^{-\beta(E_j - E_i)}$. Notice that if $E_j < E_i$, exponential term is greater than one, and the move is always accepted, otherwise, it is accepted with a probability of P_a.

Monte Carlo Simulation of an ideal gas

Quantum states of a particle in a box is given by:

$$E(n_x, n_y, n_z) = \frac{\pi^2 \hbar^2}{2mL^2}(n_x^2 + n_y^2 + n_z^2)$$

The total energy of the system is just the sum of individual energies of the particles. We define the move set as one of the six following possibilities, changing each quantum number for a particular (randomly chosen) atom by ± 1. We simulate a system of $N = 1000$ particles, with $K_B T = 10$, $m = \hbar = 1$, $L = 1$ and we start with all the particles in the ground state, i.e. all quantum numbers equal to 1. The code follows:

```
from random import randrange,random
from math import exp,pi
from numpy import ones
from pylab import plot, ylabel, show

T=10
N=1000
steps=250000

#2D array to store q no.s
n=ones([N,3],int)

#main loop
eplot=[]
E=3*N*pi*pi/2

for k in range(steps):
    #Choose random particle and move
    i=randrange(N)  #choose a random molecule
    j=randrange(3)  #choose a random q no.
    if random()<0.5:  #choose change with 0.5 chances
        dn=1
        dE=(2*n[i,j]+1)*pi*pi/2
    else:
        dn=-1
        dE=(-2*n[i,j]+1)*pi*pi/2

    #Decide whether to accept the move
    if n[i,j]>1 or dn==1: #can't go below ground state
        if random()<exp(-dE/T):
            n[i,j]+=dn
            E+=dE
    eplot.append(E)
```

```
#Make the graph
plot(eplot)
ylabel("Energy")
show()
```

And the corresponding output is shown in fig: 9.3

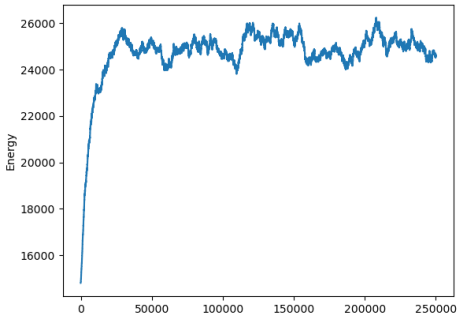


Figure 9.3: Internal Energy of 1000-particle system

9.7 Simulated Annealing

One of the most important/interesting uses of Monte-Carlo Simulations is Simulated Annealing. We have seen many methods to find the minimum/maximum of a function but none of them said anything about local or global extrema. An interesting and perplexing problem in optimization is to find the global extrema of a curve. This is precisely what simulated annealing. For a system in thermal equilibrium, the probability of occupancy of a state is given by the boltzman distribution. If we set the ground state as $E_i = 0$, then, if we allow the temperature to fall to absolute zero,

$$P(E_i) = \begin{cases} 1 & \text{for } E_i = 0, \\ 0 & \text{for } E_i > 0, \end{cases}$$

Equivalently, the system **will** attain the ground state as temperature falls to absolute zero. This suggests a way to find the global minimum (and thus maximum) of a function! Simulate the system using Markov chain Monte Carlo method, and start lowering the temperature, the system will find it's way to the ground state. Here, for minimizing the function $f(x, y, z)$, we treat the independent variables as the state of the system, and f being it's energy.However, notice that if we set $T = 0$, and the system finds itself in a local minimum, it will be trapped forever! However, we know a simple trick to get around this problem, and this trick is inspired by the physical process of annealing glass. When we

cool glass very quickly, it develops cracks weakness. This is because, first the molecules were in random motion, but due to rapid cooling, they found a local minimum of energy, and got stuck in it. Cooling the system sufficiently slowly will ensure the global minimum. We do the same for our analysis as well. All we have to do in the Markov Chain Monte Carlo simulation, is to keep reducing the temperature, from an initial temperature, until the change in energy of the state becomes very small. Most common rate of cooling is given as:

$$T = T_o e^{-t/\tau}$$

Some trial and error is necessary to find the time constant. This method is famously used in many areas of physics and beyond. For example, it can be used to, theoretically solve the Travelling salesman problem (though a perfectly guaranteed solution might require infinite time).