Computational Physics

Vinit P. Doke[1]

[1] Department of Physics, Indian Institute of Technology, Powai, Mumbai 400076 ,
Email: vinitdoke@gmail.com , 190260018@iitb.ac.in .

Mentor: Chaitanya Kumar

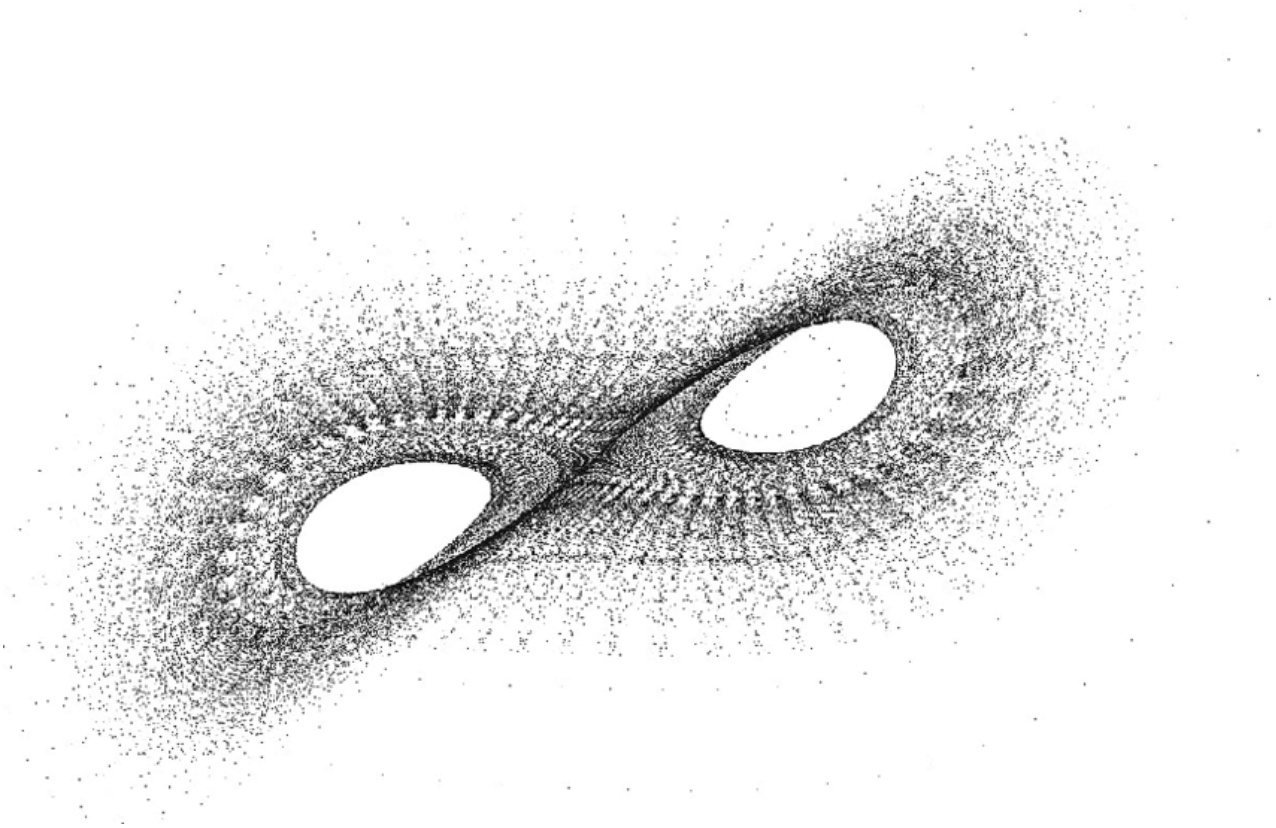


Figure 1: Lorrenz Attractor

Table Of Contents

List of Figures

1 Introduction

With greater strides being made in Experimental Physics, we are witnessing data volumes that were never seen before. For instance, the Large Hadron Collider generates petabytes of data per second. This warrants the use of computers to analyse, reformat and simulate various tasks in Physics. This is the field of study represented by Computational Sciences, Computational Physics being the application in Physics.

Most numerical calculations in physics fall into one of several general categories, based on the mathematical operations that their solution requires. Examples of common operations include calculation of integrals and derivatives, linear algebra tasks such as matrix inversion or the calculation of eigenvalues, and the solution of differential equations, including both ordinary and partial differential equations. If we know how to perform each of the basic operation types then we can solve most problems we are likely to encounter as physicists. I shall study individually the computational techniques used to perform various operations and then apply the solution to wide range of physics problems.

1.1 Using Python

Being an interpreted language, Python provides a much more natural coding experience. The plethora of support libraries available for it make it an ideal choice for scientific computing purposes. Some of the libraries required for this project include **Numpy, Pandas, Matplotlib, VPython, SciPy** etc.

Though many of these libraries provide in-built functionalities for carrying out various numerical operations, we shall follow a ground up approach towards these operations, exploring the methods followed to port them to a computer.

1.2 Numerical Methods

We shall see various methods of numerical evaluations of maths operations and then deal with the ways of calculating the errors produced due to either approximations or precision value errors (inherent to Python due to way the data types are stored). I shall be simultaneously solving problems pertaining to varied areas of Physics related to the mathematical methods which are linked here as follows:

1. Data Visualisation
2. Integration
3. Random Processes and Monte-Carlo Methods
4. Linear Equations
5. Non-Linear Equations

2 Data Visualisation

To make sense of the vast amounts of data, to study trends, patterns and various other characteristics, Visualisation serves as an indispensable tool. Data Viz. in itself is another vast domain of study, yet it has its uses in Physics. Graphs are fundamental in Physics followed by images and animations. Here, I deal with **Graphs, Density Plots, Scatter Plots and Images**.

2.1 Graphs

2D Graphs require datasets of points to be represented on a coordinate plane. One often encounters such datasets in Physics, having data regarding a dependent variable changing with an independent one. However, to plot functions, we need to evaluate them at a linear space of points between the required range to produce illusion of continuity. The computer draws straight-lines between the points. So the end-result is not smooth but a set of line-segments. However, sampling the functions at hundreds of such points enables the creation of smooth-looking curves.

2.1.1 Polar Plot

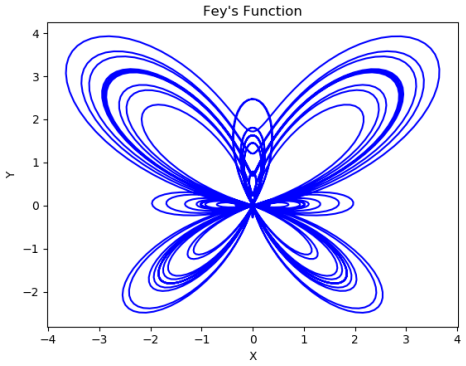
The underlying code plots functions of form $r = f(\theta)$, i.e. explicit in r and θ.

Listing 1: Polar Plot

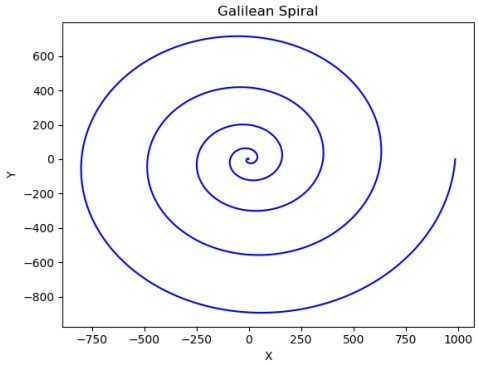
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib qt5 #for jupyter notebooks only


thetas=np.linspace(#range of theta, #number of sample points)
def cart(theta, r):
        x=r*np.cos(theta)
        y=r*np.sin(theta)
        return (x,y)
def func(theta):
        return #function here

X=[]
Y=[]
for theta in thetas:
        x,y=cart(theta, func(theta))
        X.append(x)
        Y.append(y)
plt.plot(Y,X, 'b')
plt.xlabel('X')
plt.ylabel('Y')
plt.title(# Function Name)
```



(a) $f(\theta) = e^{\cos\theta} - 2\cos 4\theta + \sin^5 \frac{\theta}{12}$



(b) $f(\theta) = \theta^2$

Figure 2: Polar Plots

2.2 Density Plots and Images

A **Density Plot** uses colours or brightness to represent magnitude on a 2D-Grid. In Python, we can pass an array of data, which can be 3D too. In this case, the values are interpreted as RGB values for each pixel in the grid. Consequently, many image formats are imported to Python in **3D-*arrays***. For instance, **FITS** (Flexible Image Transport System) is a widely used data format for storing astronomical observations by telescopes.

2.2.1 Interference Pattern

The underlying code displays the wave interference pattern for 2 plane waves produced in a medium at a particular separation.

Listing 2: Interfernce Pattern

```
amp=1
wavelength=5
d=20
k=2*np.pi/wavelength
pond=np.zeros((500,500)) # two waves in a 1m*1m pond on a 500*500 grid
def dist(p1,p2):
        x1,y1=p1
        x2,y2=p2
        x1=conv_units(x1)
        x2=conv_units(x2)
        y1=conv_units(y1)
        y2=conv_units(y2)
        return ((x2-x1)**2+(y2-y1)**2)**(0.5)
def height1(p):
        return amp*np.sin(k*dist(p1,p))
def height2(p):
        return amp*np.sin(k*dist(p2,p))
def conv_units(point):
        return point/5
def conv_units2(point):
        return point*5
```

```
def height(p):
        return height1(p)+height2(p)
p1=(250,200)
p2=(250,200+conv_units2(20))
for i in range(500):
        for j in range(500):
                pond[i][j]=height((i,j))
pond=pond/2
plt.imshow(pond,interpolation='bicubic')
```

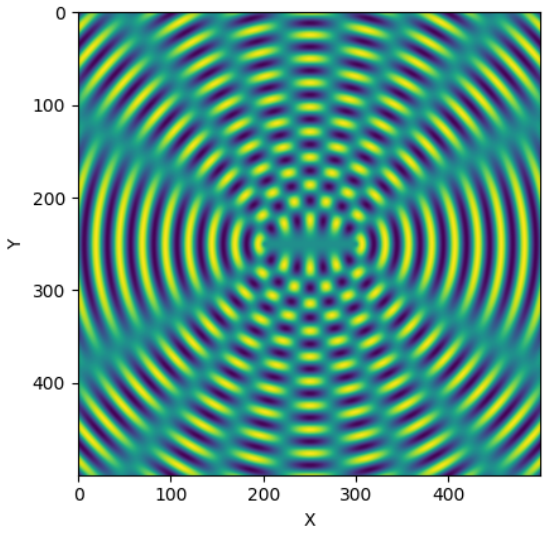


Figure 3: OUTPUT of Listing 2

3 Integration

3.1 Trapezoidal Rule

We evaluate the integral for $f(x)$ in range (a, b). One way would be to divide the range into N slices like rectangles or trapezoids. As the name suggests, we approximate the function between x_i and x_{i+1} slices as a trapezium of $l1 = f(x_i)$ and $l2 = f(x_{i+1})$ with separation $h = (b-a)/N$, which will be same for all the trapeziums. Let the true value of the Integral be represented by I. Then the approximated integral is represented by I_0, given by:

$$\begin{aligned} I_0 &= \sum_{i=1}^{N} \frac{h}{2}[f(a+(i-1)h)) + f(a+ih)] \\ &= \frac{h}{2}[f(a)+f(b)+2(f(a+h))....2f(a+(N-1)h)] \\ &= \frac{h}{2}[f(a)+f(b)+2\sum_{i=1}^{N-1} f(a+ih)] \\ &\approx I \end{aligned}$$

Naturally, we will have greater precision for higher values of N. Generally, we use Adaptive Integral Method (discussed here), to obtain results within the desired level of accuracy.

Listing 3: Trapezoidal Rule

```
def f(x):
        return x**4-2*x+1 #Function here
N= #Number Of Divisions
a= #start range
b= #end range
h=(b-a)/N
integ=f(a)/2+f(b)/2
for i in range(1,N):
        integ+=f(a+i*h)
integ*=h
print(integ) #Final Solution
```

3.2 Simpson's Rule

In Trapezoidal Rule, we approximate the curve by trapeziums, however we could fit a higher-order polynomial to our sliced function for better accuracy. Simpson's Rule does exactly that with polynomials of degree 2, i.e. Quadratic Expressions. We divide the range of $f(x)$, (a, b) into N equally spaced slices of width $h = (b-a)/N$. Let us consider 3 points, $\{-h, 0, h\}$. The Quadratic polynomial passing through these points would be:

$$\begin{aligned} f(x) &= ax^2 + bx + c \\ f(-h) &= ah^2 - bh + c \\ f(0) &= c \\ f(h) &= ah^2 + bh + c \end{aligned}$$

Solving for the coefficients gives us the constants a, b, c in terms of h. Now, we integrate over the slice $[-h, h]$ as follows:

$$\int_{-h}^{h} (ax^2 + bx + c)dx = \frac{h}{3}[f(-h) + 4f(0) + f(h)]$$

Performing this integral for all slices of form x_{i-1}, x_i, x_{i+1} returns the general approximate integral I_0 as:

$$I_0 = \frac{h}{3}[f(a) + f(b) + 4\sum_{k=1}^{Odd} f(a+kh) + 2\sum_{k=2}^{Even} f(a+kh)]$$
$$\approx I$$

Due to better order approximation, Simpson's rule generally yields more accurate results than the Trapezoidal rule in lesser steps (N). To obtain results within a specified degree of accuracy, we use the Adaptive Integral Method for Simpson's Rule (discussed here).

Listing 4: Simpson's Rule

```
def f(x):
        return x**4-2*x+1 #function here
a= #start range
b= #end range
N= #Number of Divisions
h=(b-a)/N
s=f(a)+f(b)
for i in range(1,N,2):
        s+=4*f(a+i*h)
for i in range(2,N-1,2):
        s+=2*f(a+i*h)
s=s*h/3
print(s)#Final Value
```

3.2.1 Diffraction Limit Of Telescope

Our ability to resolve detail in astronomical observations is limited by the diffraction of light in our telescopes. Light from stars can be treated effectively as coming from a point source at infinity. When such light, with wavelength λ, passes through the circular aperture of a telescope (which we'll assume to have unit radius) and is focused by the telescope in the focal plane, it produces not a single dot, but a circular diffraction pattern consisting of central spot surrounded by a series of concentric rings. The intensity of the light in this diffraction pattern is given by

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

where r is the distance in the focal plane from the center of the diffraction pattern, $k = 2\pi/\lambda$, and $J_1(x)$ is a Bessel function. The Bessel functions $J_m(x)$ are given by

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

where m is a nonnegative integer and $x \geq 0$.

Here, I work out the image of the focal plane for light of $\lambda = 500nm$

Listing 5: Bessel Function

```
def J(m,x):
        def f(t):
                return np.cos(m*t-x*np.sin(t))
        a=0
        b=np.pi
```

```
        N=1000
        h=(b-a)/N
        s=f(a)+f(b)
        for i in range(1,N,2):
                s+=4*f(a+i*h)
        for j in range(2,N-1,2):
                s+=2*f(a+j*h)
        s*=h/3
        return (1/(np.pi)*s)
```

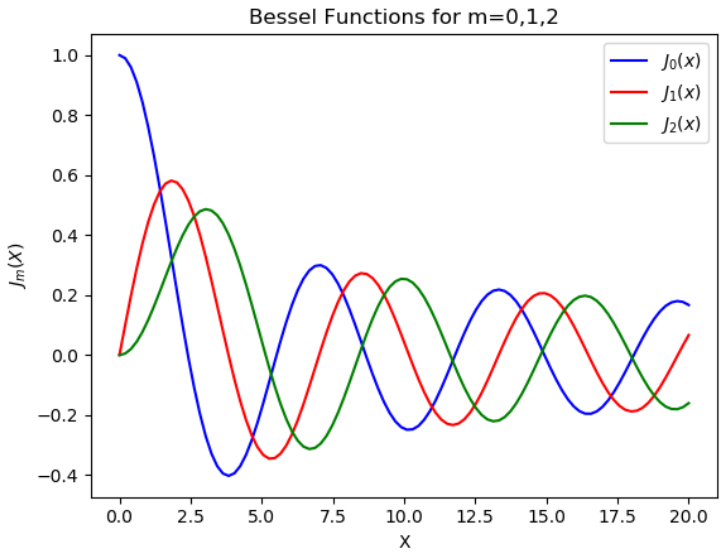


Figure 4: Plot of Bessel Functions

Listing 6: Main Function

```
focal_plane=np.array([[0 for i in range(500)] for i in range(500)])
center=(250,250)
wavelength=62.5
k=2*np.pi/wavelength
def dist(point):
        x0,y0=center
        x,y=point
        return np.sqrt((x0-x)**2+(y0-y)**2)
def I(r):
        return ((J(1,k*r))/(k*r))**2
for i in range(500):
        for j in range(500):
                focal_plane[i][j]=dist((i,j))
fp2=I(focal_plane)
mean=fp2.mean()
std=fp2.std()
plt.imshow(fp2,'cividis',vmax=mean+0.5*std, interpolation='bicubic')
```

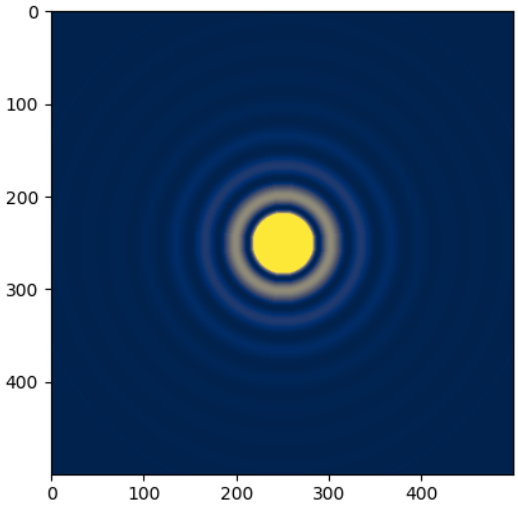


Figure 5: OUTPUT of Listing 6

3.3 Errors on Integrals

The numerical integrals mentioned before were only approximations. Therefore, we are able to derive their approximation errors to get an idea of the precision involved.

We define $x_k = a + kh$, a point to evaluate $f(x)$ at. Now Taylor expansion at $x_{k-1}$is given by:

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

Integrating $f(x)$ in range $[x_{k-1}, x_k]$ gives:

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

Substituting $u = x - x_{k-1}$:

$$\begin{aligned} \int_{x_{k-1}}^{x_k} f(x) \cdot dx &= f(x_{k-1}) \int_0^h du + f^{'}(x_{k-1}) \int_0^h u \cdot du + \frac{f^{''}(x_{k-1})}{2} \int_0^h u^2 \cdot du + ... \\ &= hf(x_{k-1}) + \frac{h^2}{2} f^{'}(x_{k-1}) + \frac{h^3}{6} f^{''}(x_{k-1}) + O(h^4) \end{aligned}$$

where, $O(h^4)$ denotes the rest of the terms in the series, those in h^4 and higher.

We can do similar expansion around $x = x_k$ and again integrate within the given range. Taking the average of the integrals in range $[x_{k-1}, x_k]$ with Taylor expansions at both $x_{k-1} and x_k$ gives us:

$$\int_{x_{k-1}}^{x_k} f(x) \cdot dx = \frac{h}{2}[f(x_{k-1}) + f(x_k)] + \frac{h^2}{4}[f^{'}(x_{k-1}) - f^{'}(x_k)] + \frac{h^3}{12}[f^{''}(x_{k-1}) + f^{''}(x_k)] + O(h^4)$$

Finally, we sum this expression over all slices k to get the full integral that we want:

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

Making the changes necessary to fit the above equation to the **Trapezoidal Rule** we have:

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

Thus, to leading order in h, the value of the terms dropped when we use the trapezoidal rule, which equals the approximation error ϵ on the integral is:

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

This is the *Euler-Maclaurin formula* for the error on the trapezoidal rule. Similar operations for the **Simpson's Rule** yield:

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

For programming purposes, we use the numerical approach of calculating ϵ which is given as follows:

$$\textbf{Trapezoidal Rule} : \epsilon = \frac{1}{3}[I_2 - I_1]$$

$$\textbf{Simpson's Rule} : \epsilon = \frac{1}{15}[I_2 - I_1]$$

3.4 Adaptive Integral Method

This method utilises the numerical approach of calculating the errors. We begin by calculating an approximate integral I_1 with N steps. Similarly I_2 is calculated with $2N$ steps. Then, the error will be given by:

$$\epsilon = \frac{1}{3}[I_2 - I_1]$$

Furthermore, If $\epsilon > \epsilon_0$, where ϵ_0 is the degree of accuracy, we double the number of steps until $\epsilon < \epsilon_0$. This is implemented for the **Trapezoidal Rule** as follows:

Listing 7: Adaptive Rule

```
def f(x):
        return #Function here
a= #Start Range
b= #End Range
thresh= #threshold
error=1 #Seed Error
N=1 #Repetitions
I1=(f(a)+f(b))/2
while error>thresh:
        N=2*N
        h=(b-a)/N
        s=1/2*(f(a)+f(b))
        for i in range(1,N):
                s+=f(a+i*h)
        I2=h*s
        error=abs(1/3*(I2-I1))
        I1=I2
print(f'The value of integral is {I1} with error {error}')
```

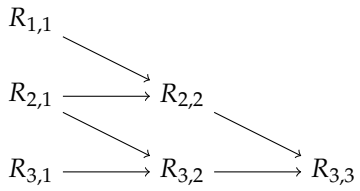
3.5 Romberg Integration

The **Romberg Integration Formula** is :

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

which is accurate to order h^{2m+1} with an error of order h^{2m+2}. The procedure is given as follows:

1. Calculate first two estimates using **Trapezoidal Rule** :
   $I_1 \equiv R_{1,1}$ and $I_2 \equiv R_{2,1}$
2. From these two, calculate the more accurate estimate: $R_{2,2}$
3. Thus. for each estimate we also calculate the error to meet some desired accuracy.



3.6 Gaussian Quadrature

Until now, we had been using numerical methods that had evenly spaced point within the range. However, for functions with high variation in slope, we require higher number of points of evaluation in regions of higher variations of slope. Conversely, we do not require a lot of points for functions with lesser variation in slope. This dilemma is partly solved by the **Gaussian Quadrature** method.

To calculate an approximation to our integral, we have:

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

Thus, we have weights w_k given by:

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

Often, we are given these weights for $a = -1$ and $b = +1$. These can be mapped to our new variables by the following equations:

$$x'_k = \frac{1}{2}(b-a)x_k + \frac{1}{2}(b+a)$$
$$w'_k = \frac{1}{2}(b-a)w_k$$

3.6.1 Sample Points

The sample points x_k should be chosen to coincide with the zeros of the N^{th} **Legendre Polynomial** $P_N(x)$, rescaled if necessary. The corresponding weights w_k are:

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

To obtain such points, I shall use the python program discussed in Appendix.

3.6.2 Gaussian Quadrature Implementation

Listing 8: Gaussian Quadrature

```
from gaussxw import gaussxwab #to obtain sample points and weights
def f(x):
        return #function here
a= #Start Range
b= #End Range
N= #Iterations
x,w=gaussxwab(N,a,b)
integ=0
for i in range(N):
        integ+=f(x[i])*w[i]
print(integ)
```

3.6.3 Heat Capacity Of A Solid

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

Listing 9: Heat Capacity

```
V=10**(-3)
ndensity=6.022*10**28
debyetemp=428
N=50
k=1.38065852*10**(-23)
x,w=gaussxw(N)
def f(x):
        return (x**4)*(np.exp(x))/(np.exp(x)-1)**2
def cv(T):
        a=0
        b=debyetemp/T
        integ=0
        wk=(b-a)/2*w
        xk=(b-a)/2*x+(b+a)/2
        for i in range(N):
                integ+=f(xk[i])*wk[i]
        return 9*V*ndensity*k*((T/debyetemp)**3)*integ
T=np.arange(5,501)
C_val=[]
for i in range(len(T)):
        C_val.append(cv(T[i]))
plt.figure(figsize=(10,5))
plt.plot(T,C_val)
plt.xlabel('Temp, K')
plt.ylabel('C_V')
plt.title('Heat Capacity Of Solid Al v/s Temp')
```

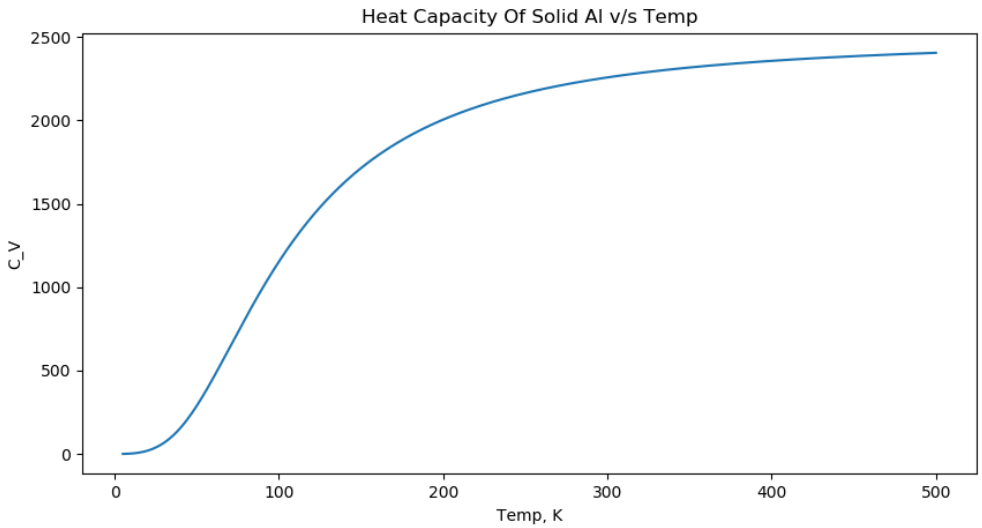


Figure 6: OUTPUT for Listing 9

3.7 Integrals over Infinite Ranges

We cannot possibly converge on an acceptable solution for integrals with infinite ranges by a numerical approach, since there are infinite slices. Therefore, what we require is a **change of variables**. To evaluate $\int_0^\infty f(x)dx$, we substitute

$$z = \frac{x}{1+x}$$

Then,

$$dx = \frac{dz}{(1-z)^2}$$

and

$$\int_0^\infty f(x)dx = \int_0^1 \frac{1}{(1-z)^2} f\left(\frac{z}{1-z}\right) dz$$

Similar substitution for ranges like $(-\infty, 0)$,(a, ∞) etc. can be made. To evaluate in range $(-\infty, \infty)$, we divide the range at 0 and evaluate accordingly.

3.8 Multiple Integrals

To evaluate

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

We can re-write this by defining a function $F(y)$, thus

$$F(y) = \int_0^1 f(x,y)dx$$

Then,

$$I = \int_0^1 F(y)dy$$

Similarly, for using the **Gaussian Quadrature**, we have the ***Gauss-Legendre product formula***:

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

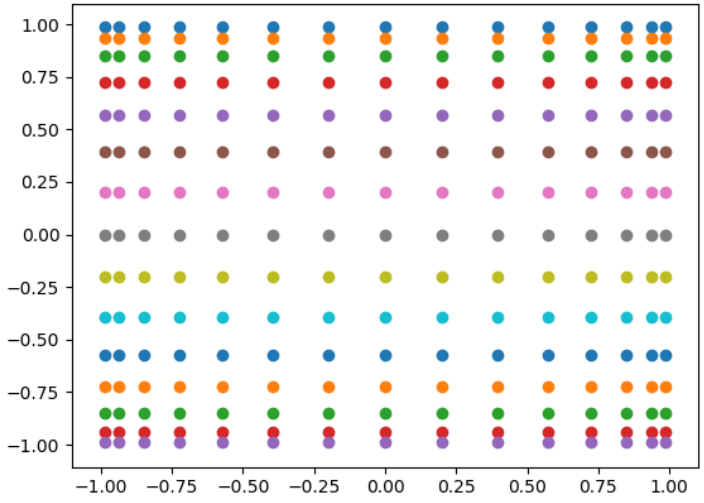


Figure 7: Points of Evaluation for Gaussian Quadrature

4 Random Processes and Monte-Carlo Methods

4.1 Random Number Genarators

The discussed equations produce seemingly random numbers given a seed number. However, these aren't truly random as they are deterministic and can be re-discovered given the seed number is known. Hence, they are referred to as **pseudorandom numbers.**

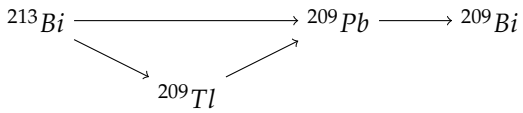
Consider the equation:

$$x' = (ax + c)mod(m),$$

where a, c, m are integer constants and x is an integer variable. Given a value of x (seed), x' is the resultant integer which can be re-fed to the equation as x. This is a *linear congruential random number genarator.*

4.2 Radioactive Decay Chain

^{213}Bi deacays to stable ^{209}Bi via two different routes given as:



Here, we begin with $10,000$ atoms of ^{213}Bi and for each such atom, we generate a random number between 0 and 1. We, compare the number with the probability of the decay of such an atom in time t which is given by the equation:

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

where τ is the half-life for ^{213}Bi. We calculate the probability for $t = 1sec$ with $t_{max} = 20,000seconds$. This is done for all decaying entities, namely, ^{213}Bi, ^{209}Tl, and ^{209}Pb

Listing 10: Decay Chain

```
Bi213=10000
Tl=0
Pb=0
Bi209=0
tau1=46*60
tau2=2.2*60
tau3=3.3*60
h=1
p1=1-2**(-h/tau1)
p2=1-2**(-h/tau2)
p3=1-2**(-h/tau3)
tmax=20000
b2p=0.9791
b2t=1-b2p
NTl=[]
NPb=[]
NBi213=[]
NBi209=[]
tpoints=np.arange(0,tmax,h)
for t in tpoints:
        Pbdecay=0
        Bi213decay=0
```

```
        Tldecay=0
        for p in range(Pb):
                if np.random.rand()<p3:
                        Pbdecay+=1
        for t in range(Tl):
                if np.random.rand()<p2:
                        Tldecay+=1
        for b in range(Bi213):
                if np.random.rand()<p1:
                        Bi213decay+=1

        Pb-=Pbdecay
        Bi209+=Pbdecay
        Pb+=Tldecay
        Tl-=Tldecay
        Bi213-=Bi213decay
        b2pdecay=0
        b2tdecay=0
        for _ in range(Bi213decay):
                if np.random.rand()<b2p:
                        b2pdecay+=1
                else:
                        b2tdecay+=1
        Pb+=b2pdecay
        Tl+=b2tdecay

        NTl.append(Tl)
        NPb.append(Pb)
        NBi213.append(Bi213)
        NBi209.append(Bi209)
```

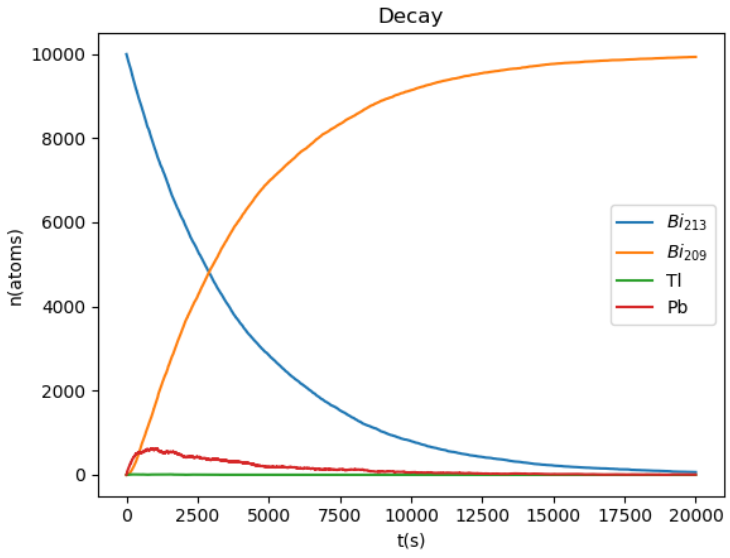


Figure 8: Output for Above Code

4.3 Brownian Motion

To simulate a random walk, I assume a Canvas of size $500 * 500$ pixels. We select at random, one of 4 directions : Up, Down, Left, and Right. Running the code for some prescribed number of iterations N, we obtain random paths. The code is as follows:

Listing 11: Random Walk

```
N=100000*3
current_pos=np.array([250,250])
canvas=np.zeros([501,501])
def fill(Dir, current_pos):
        x,y=current_pos
        if Dir==0:
                dx,dy=-1,0
        if Dir==1:
                dx,dy=+1,0
        if Dir==2:
                dx,dy=0,+1
        if Dir==3:
                dx,dy=0,-1
        if (x+dx)<0 or (x+dx)>500:
                dx=0
        if (y+dy)<0 or (y+dy)>500:
                dy=0
        canvas[x+dx,y+dy]+=1
        return np.array([x+dx,y+dy])
for i in range(N):
        Dir=np.random.randint(4)
        current_pos=fill(Dir, current_pos)
plt.imshow(canvas)
```

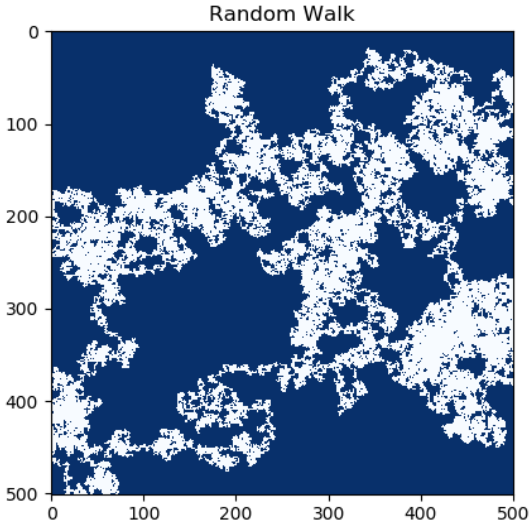


Figure 9: Random Walk: Output for Above Code

4.4 Monte-Carlo Integration