Python Lab Notebook

Soumya Majhi

Sem-4 2022

CC8 Practical Lab Notebook

Stream: B.Sc

Semester: 4

Shift: Day

Department: Physics

Subject Code: PHSA

College Roll: 2817

University Roll No.: 203224-21-0011

University Registration No.: 224-1111-0520-20

1 1^{st} Order ODE

THEORY: The general form of 1^{st} order ODE:

$$\frac{dx}{dt} = f(x,t) \tag{1}$$

For a first order ODE to solve one needs one initial condition. For example, $x = x - 0$ at $t = 0$. The `odeint()` function takes the function name (f) as argument variable (t) over which the solution is to seek.
The `odeint()` returns an array which contains a column of values of **`x`** at all points in the given array **`t`**. The function, **`odeint(f, x0, t)`** takes threee default arguments, where **`'f'`** is the name of the user definmed function tht contains the derivative dx/dt. Integration is done for x at all points of the array t where $x0$ is the the initial value.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
def f(x,t):
    dxdt = 5.0*x
    return dxdt
x0 =1.0
t=np.linspace(0,10,101)
s=odeint(f,x0,t)
print(s)
plt.plot(t,s)
plt.xlabel("Value of t")
plt.ylabel("Value of s (ode)")
plt.show()
```

```
OUTPUT
[[1.00000000e+00]
 [1.64872127e+00]
 [2.71828191e+00]
 ...
 [1.90734832e+21]
 [3.14468577e+21]
 [5.18471037e+21]]
```

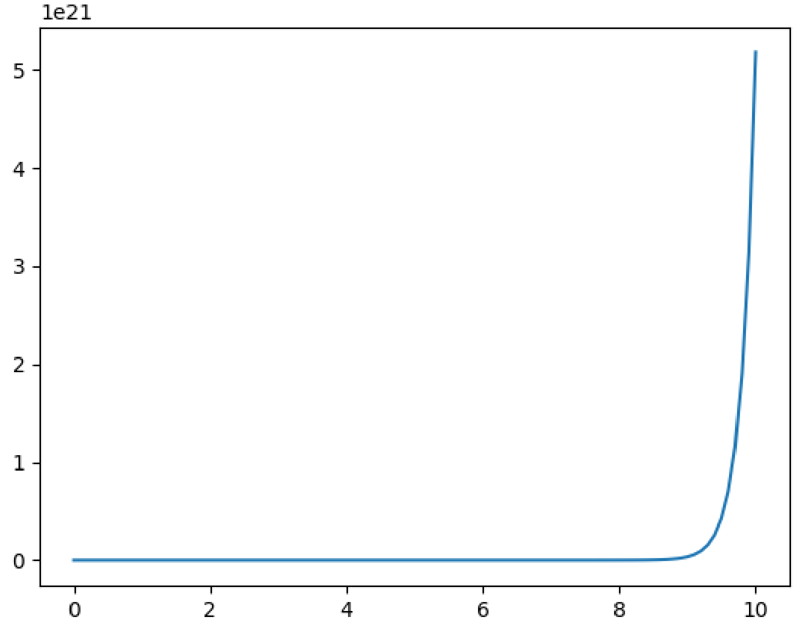

Figure 1: x vs. t plot

2 2^{nd} Order ODE (Classical Harmonic Oscillator - Damping)

THEORY: We can split a 2^{nd} order differential equation into two coupled 1^{st} order equations. For each of the first order equations we can use `odeint()` to solve.
To solve, we need two initial values of $\boldsymbol{x}$ and $\boldsymbol{y}$ and the array for the values of independent variable $\boldsymbol{t}$ (The domain over which we find out x and y at all points).

As the function **`odeint(func, y0, t)`** can take only three default arguments, we pack two functions **`dxdt`** and **`dydt`** into one to put in place of '`func`' and pack two initial values for dependent variables **`(x,y)`** into one name to put in place of `y0`.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

z=float(input('enter value of "z": '))          #damping factor
k=float(input('enter value of "k": '))          #spring factor
def f(u,t):
    x=u[0]
    y=u[1]
    dxdt=y
    dydt=-z*y-k*x
    return np.array([dxdt,dydt])
u0=[0,1]
t=np.linspace(0,500,10001)
s=odeint(f,u0,t)
print(s)

plt.plot(s[:,0],s[:,1])
plt.show()
plt.plot(t,s[:,0])
plt.show()
```

```
OUTPUT
enter value of "z": 0.02
enter value of "k": 0.49
[[ 0.          1.        ]
 [ 0.04996479  0.99838847]
 [ 0.09981849  0.99555626]
 ...
 [-0.00890059 -0.00249621]
 [-0.00901986 -0.00227428]
 [-0.00912797 -0.00204979]]
```

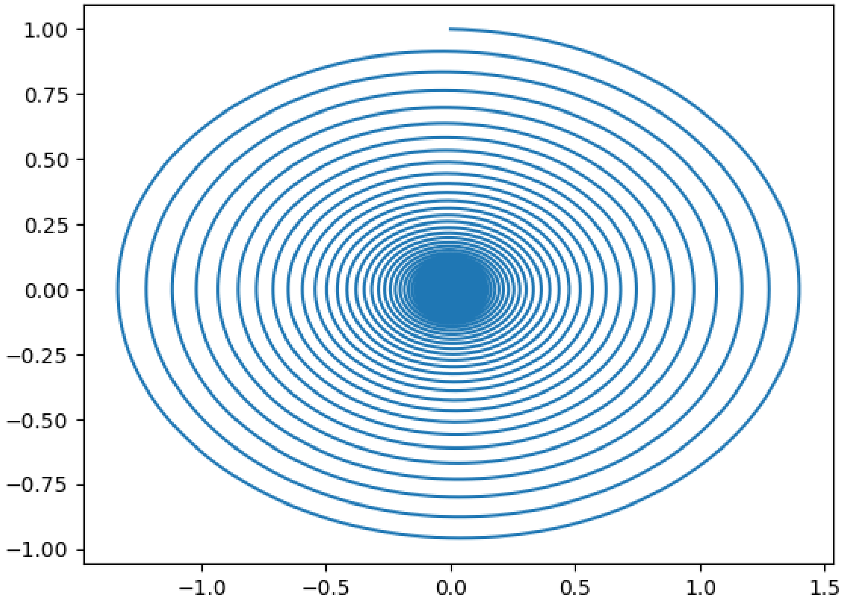

Figure 2: x-y plot

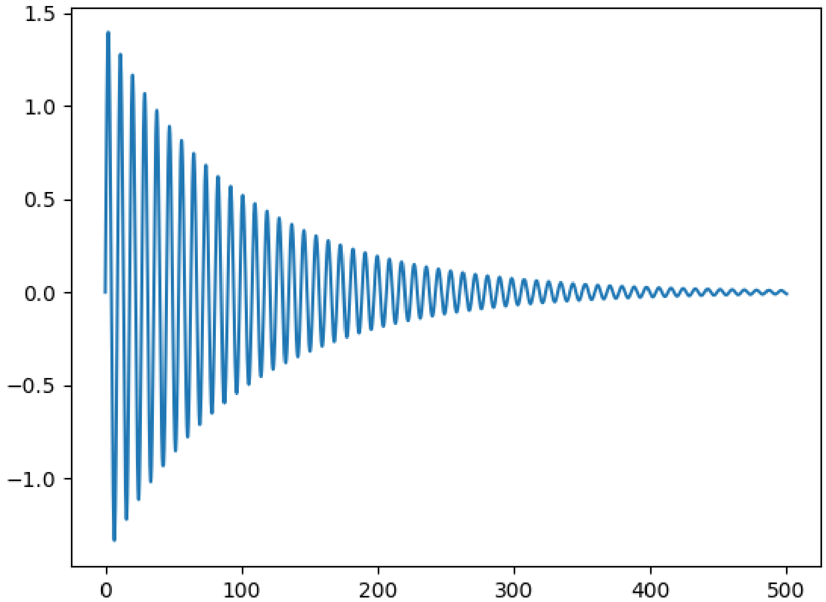

Figure 3: The solution: x vs. t plot

3 Lorentz Attractor Model - Non-Linear Plot

THEORY: Consider the following set of three coupled 1^{st} order Differential equations:

$$\frac{dx}{dt} = \sigma(y - x), \tag{2}$$
$$\frac{dy}{dt} = x(\rho - z) - y, \tag{3}$$
$$\frac{dz}{dt} = xy - \beta z \tag{4}$$

Solving these, we arrive at famous Lorentz curves(in the area of Chaos). The values of the parameters that Lorentz used: $\sigma = 10, \rho = 28, \beta = 8/3$

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def f(u,t):
    x=u[0]
    y=u[1]
    z=u[2]
    dxdt=10*(y-x)
    dydt=x*(28-z)-y
    dzdt=x*y-(8/3)*z
    return np.array([dxdt,dydt,dzdt])
uo=[1,0,0]

t=np.linspace(0,101,100001)
s=odeint(f,uo,t)
print(s)

#plt.plot(t,s[:,0],t,s[:,1],t,s[:,2])  #PLOT OF X(t),Y(t),Z(t)IN SAME GRAPH

plt.plot(s[:,0],s[:,2])                #PLOT OF X(t)vs Z(t)
plt.xlabel("Value of X(t)")
plt.ylabel("Value of Z(t)")
plt.show()
```

```
OUTPUT
[[ 1.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 9.90092652e-01  2.81247606e-02  1.41229057e-05]
 [ 9.80565956e-01  5.59464679e-02  5.58693144e-05]
 ...
 [-1.03134805e+01 -1.62501880e+01  2.01840013e+01]
 [-1.03734653e+01 -1.63147924e+01  2.02995863e+01]
 [-1.04334924e+01 -1.63785807e+01  2.04165197e+01]]
```

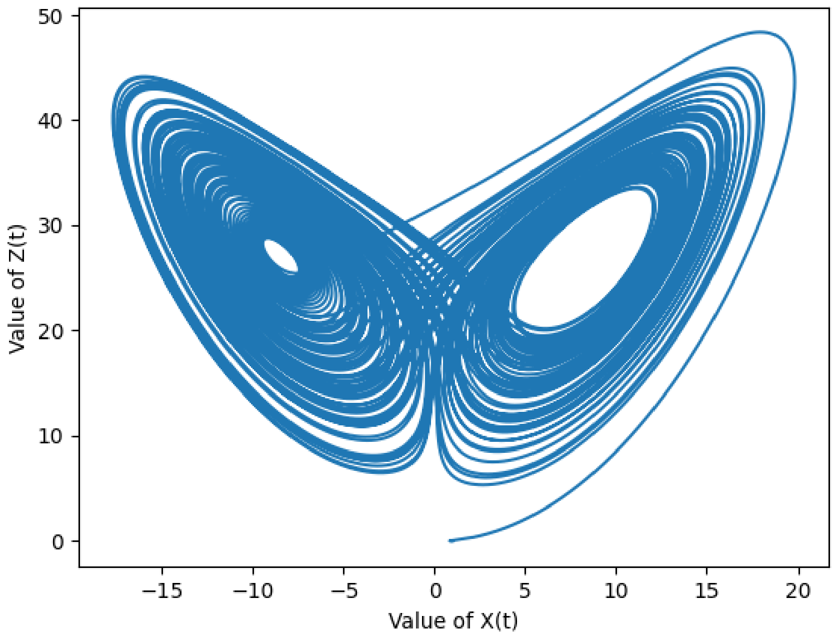

Figure 4: Lorentz Attractor

4 Gaussian Function

THEORY: Consider the Gaussian Integral,

$$\frac{N}{\sigma\sqrt{2\pi}}\int_{-\infty}^{+\infty} e^{-(x-\mu)^2/2\sigma^2}dx = 1 \tag{5}$$

If we want to check this integral, we may choose the integration limits to be as large as possible, which we may essentially call 'infinity'.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

x=np.linspace(-10,10,100)

N=float(input("Enter value of N: "))
sig=float(input("Enter value of sigma: "))
mu=float(input("Enter value of mu: "))

f=lambda x:(N/sig*np.sqrt(2*np.pi))*np.exp((-(x-mu)**2)/(2.0*sig**2))

#print(np.array([x,f(x)]))
plt.plot(x,f(x))
plt.show()
```

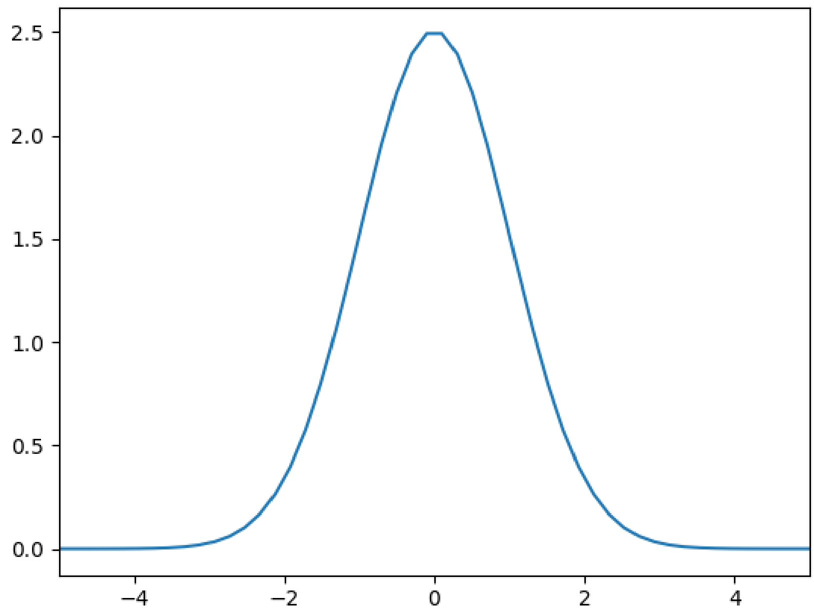

Figure 5: Gaussian Curve

5 Area Under The Curve Of Gaussian Function

THEORY: In NumPy, the constant, 'inf' is a floating-point representation of positive infinity. [We can check NumPy/SciPy documentation for all the available constants, pi, nan, inf etc.]. Essentially, positive, or negative 'infinity' is the largest or smallest number to represent. Python does it dynamically. We input any number; infinity is larger than that.
Here, we use quad(), simps(), trapz() functions to integrate the Gaussian function.

```
import numpy as np
from scipy.integrate import quad,simps,trapz

import matplotlib.pyplot as plt

x=np.linspace(-2,2,200)

N=float(input("Enter value of N: ")) #NORMALISATION CONSTANT
sig=float(input("Enter value of sigma: ")) #FWMH WIDTH
mu=float(input("Enter value of mu: "))#POSITION OF PEAK

f=lambda x: (N/sig*np.sqrt(2*np.pi))*np.exp((-(x-mu)**2)/(2.0*sig**2))

s1=quad(f,-np.inf,np.inf)
x1=np.linspace(-1,1,101)
s2=simps(f(x1),x1)
s3=trapz(f(x1),x1)
print('INTEGRATION BY QUAD: ',s1)
print('INTEGRATION BY SIMPSON: ',s2)
print('INTEGRATION BY TRAPZ: ',s3)

#print(np.array([x,f(x)]))

plt.plot(x,f(x))
plt.show()
```

```
OUTPUT
Enter value of N: 1
Enter value of sigma: 0.2
Enter value of mu: 0
INTEGRATION BY QUAD:  (6.283185307179587, 4.639297412243715e-08)
INTEGRATION BY SIMPSON:  6.283181703891748
INTEGRATION BY TRAPZ:  6.2831816274494745
```

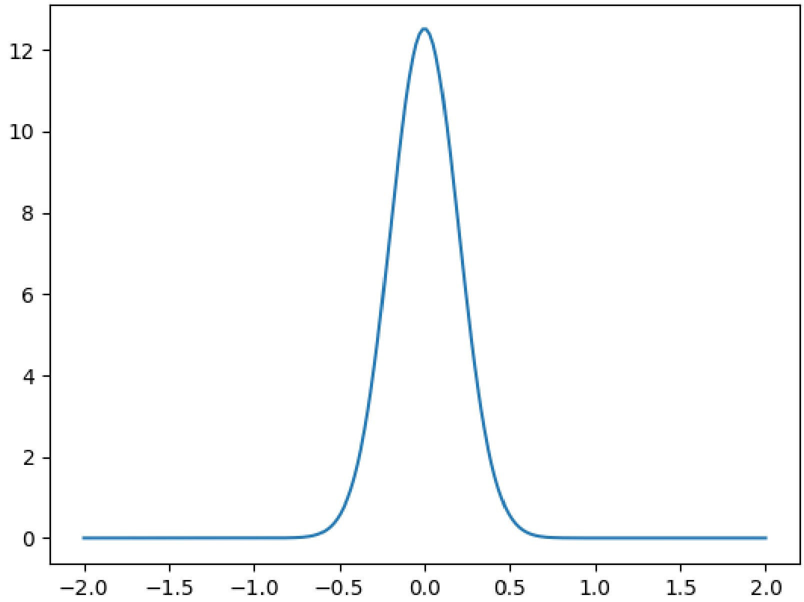

Figure 6: Another Gaussian Curve

6 Numerically Verifying A Given Gaussian Integral

THEORY: To numerically verify the integral,

$$\int_{-\infty}^{+\infty} e^{(-ax^2+bx+c)} = \sqrt{\frac{\pi}{a}}e^{(\frac{b^2}{4a}+c)} \tag{6}$$

we make use of '`inf`' from `NumPy`, `quad()` function (alongwith `simps()`, `trapz()`) to integrate the LHS of eq. 6 as '`l(x)`' and see if the result of the integration matches well with the analytical value, i.e, RHS of eq. 6 '`r(x)`'.

```
import numpy as np
from scipy.integrate import quad,simps,trapz
import matplotlib.pyplot as plt

a=float(input("Enter value of a: ",))
b=float(input("Enter value of b: ",))
c=float(input("Enter value of c: ",))
x=np.linspace(-1,1,101)

l=lambda x: np.exp(-a*x**2+b*x+c)              #LHS of Gaussian Integration
r=lambda x: np.sqrt(np.pi/a)*np.exp((b**2/4*a)+c) #RHS of Gaussian Integration
s1=quad(l,-np.inf,np.inf)
s2=simps(l(x),x)
s3=trapz(l(x),x)

print('VALUE OF LHS INTEGRATION BY QUAD: ',s1)
print('VALUE OF LHS INTEGRATION BY SIMPSON: ',s2)
print('VALUE OF LHS INTEGRATION BY TRAPZ: ',s3)
print('Value of RHS: ', r(x))
```

```
OUTPUT
Enter value of a: 1
Enter value of b: 1
Enter value of c: 1
VALUE OF LHS INTEGRATION BY QUAD:  (6.1864718159341905, 2.0214475437552896e-08)
VALUE OF LHS INTEGRATION BY SIMPSON:  4.598420051234154
VALUE OF LHS INTEGRATION BY TRAPZ:  4.598292642469942
Value of RHS:  6.186471815934188
```

7 Dynamical Integration Of Discrete Data

```
from scipy.integrate import simps,quad
import numpy as np
import matplotlib.pyplot as plt

x=[0.0,1.0,2.0,3.0,4.0] # x values
y=[0.0,1.0,4.0,9.0,16.0] # corrosponding y values

x1=[]
y1=[]
I=[]

for i in range(len(x)):
    x1.append(x[i])
    y1.append(y[i])
    print (x1,y1)
    I=simps(y1,x1)
    print(I)
plt.plot(x1,y1)  #ploting the Integration values performed manually

#now, original function definition instead of given points
r=np.linspace(0,5)
plt.plot(r,r**2)

plt.show ()
```

```
OUTPUT
[0.0] [0.0]
0.0
[0.0, 1.0] [0.0, 1.0]
0.5
[0.0, 1.0, 2.0] [0.0, 1.0, 4.0]
2.6666666666666665
[0.0, 1.0, 2.0, 3.0] [0.0, 1.0, 4.0, 9.0]
9.166666666666666
[0.0, 1.0, 2.0, 3.0, 4.0] [0.0, 1.0, 4.0, 9.0, 16.0]
21.333333333333332
```

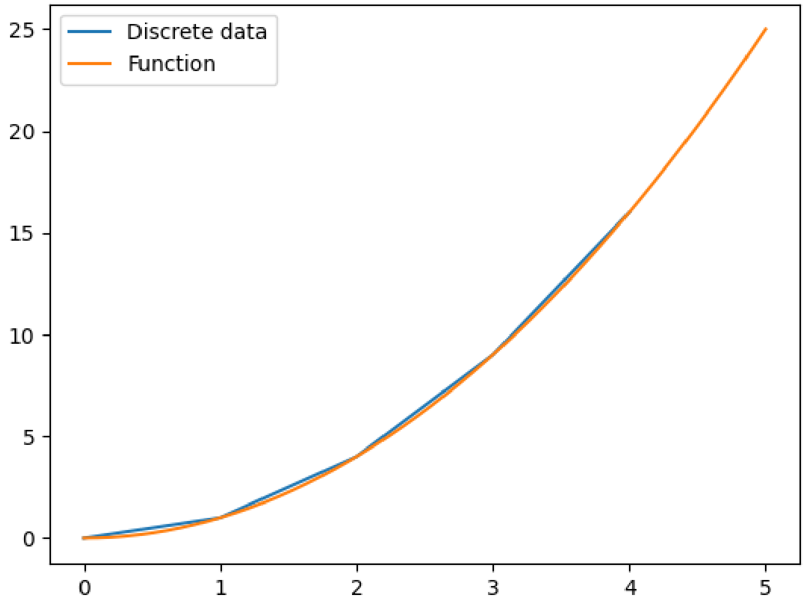

Figure 7: Plot of Integration of Discrete Data and its Function

8 Gaussian to Delta Function

THEORY:

$$\int_{-\infty}^{+\infty} g(x)G(x-\mu)dx = g(\mu) \tag{7}$$

Verifying the above integral by using different limiting representations of $G(x-\mu)$, where $G(x-\mu)$ is a Gaussian function.
We use the function:

$$f(x) = (x^2+3x)\frac{N}{\sigma\sqrt{2\pi}}\int_{-\infty}^{+\infty} e^{-(x-\mu)^2/2\sigma^2}dx \tag{8}$$

```
import numpy as np
from scipy.integrate import quad,simps,trapz
import matplotlib.pyplot as plt

a=float(input("Give the range of the X axis where the function is to be plotted: "))
x=np.linspace(-a,a,200)
N=float(input("Enter value of N: ")) #NORMALISATION CONSTANT
sig=float(input("Enter value of sigma(sig): ")) #FWHM WIDTH
mu=float(input("Enter value of mu: "))#POSITION OF PEAK

f=lambda x:(x**2+3.0*x)*(N/(sig*np.sqrt(2*np.pi)))*np.exp((-(x-mu)**2)/(2.0*sig**2))
#f=lambda x: (N/(sig*np.sqrt(2*np.pi)))*np.exp((-(x-mu)**2)/(2.0*sig**2))

s1=quad(f,-np.inf,np.inf)
print('INTEGRATION BY QUAD: ',s1)
plt.plot(x,f(x))
plt.show()
```

```
OUTPUT
Give the value of the X axis range where the function is to be plotted: 10
Enter value of N: 1
Enter value of sigma(sig): 0.2
Enter value of mu: 0
INTEGRATION BY QUAD:  (0.04000000000000008, 9.187619849545337e-09)
```

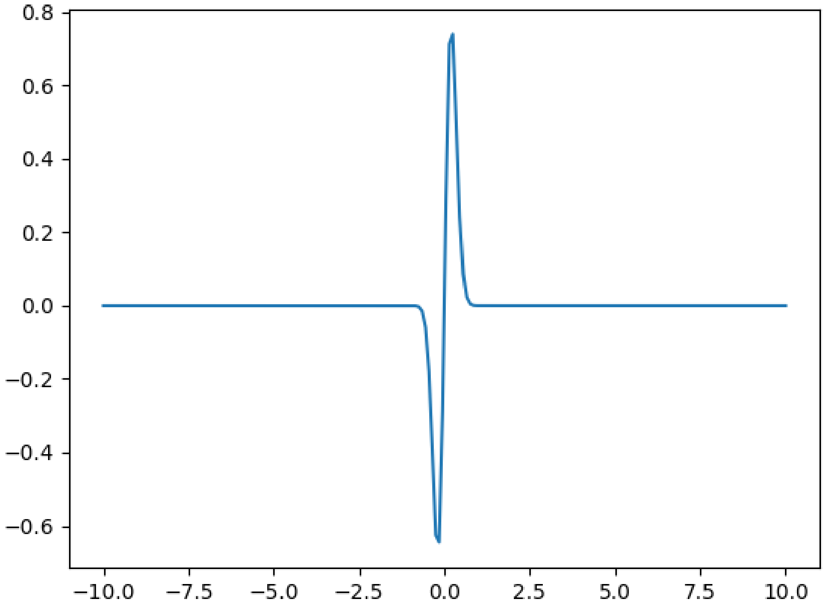

Figure 8: Plot of the function

9 Plotting of a Delta Function

THEORY: Dirac Delta function is type of a peak function which is defined as

$$\delta(x) = \begin{cases} +\infty, & x = 0, \\ 0, & x \neq 0, \end{cases} \tag{9}$$

Now, to plot the function, we use the same condition as above to define the function. And then `vectorize()` the function as an array. As we make `a` sufficiently small, and ϵ (`eps`) sufficiently large, the distribution starts to behave like dirac delta function

```
import numpy as np
from scipy.integrate import quad,simps,trapz
import matplotlib.pyplot as plt

eps=20     #epsilon
a=0.01
delta=lambda x : eps if abs(x)<(a/2) else 0
delta=np.vectorize(delta) #it is used when we plot a piecewise function
x=np.linspace(-2,+2,1000)

plt.plot(x,delta(x))
plt.show()
```

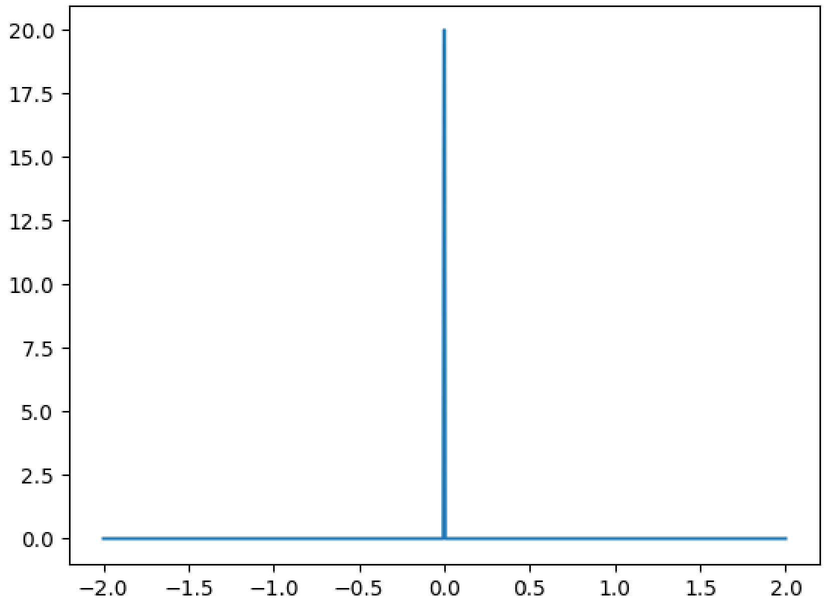

Figure 9: Plot of Delta function

10 Legendre Polynomial

THEORY: Legendre polynomials are a type of orthogonal polynomials. There are different ways to evaluate a Legendre polynomial, using generating functions, Rodrigues' formula, recurrence relation, Gram-Schmidt orthogonalization etc.
Legendre Polynomials (of different orders) satisfy the following differential equation:

$$\frac{d}{dx}[(1-x^2)\frac{d}{dx}P_n(x)] + n(n+1)P_n(x) = 0 \tag{10}$$

Rodrigues' formula:

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}[(x^2-1)^n] \tag{11}$$

To plot the legendre polynomials, we use the module `legendre` from `scipy.special`.

```
from scipy.special import legendre as l
import numpy as np
import matplotlib.pyplot as plt

x=np.linspace(-1,1,1000)
for i in range(10):
    plt.plot(x,l(i)(x),label="$P_{}(x)$".format({i}))
plt.legend()
plt.show()
```

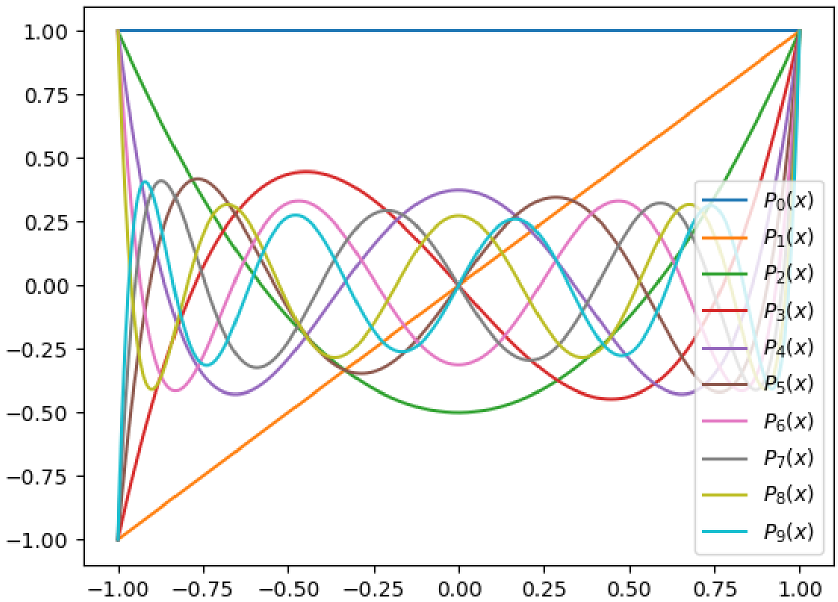

Figure 10: Legendre Polynomials

11 Legendre Recursion Formula-1

THEORY: To plot the given legendre recursion relation, we use the module `legendre` from `scipy.special`.

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - n + P_{n-1}(x) \tag{12}$$

```
from scipy.special import legendre as l
import numpy as np
import matplotlib.pyplot as plt

x=np.linspace(-1,1,1000)
n=int(input("Enter value of 'n': "))
LHS = (n+1)*l(n+1)(x)
RHS = (2*n+1)*x*l(n)(x)-n*l(n-1)(x)

plt.plot(x,LHS,'*',color="red",label="LHS")
plt.plot(x,RHS,'+',color="cyan",label="RHS")
plt.legend()
plt.show()
```

```
OUTPUT
Enter value of 'n': 4
```

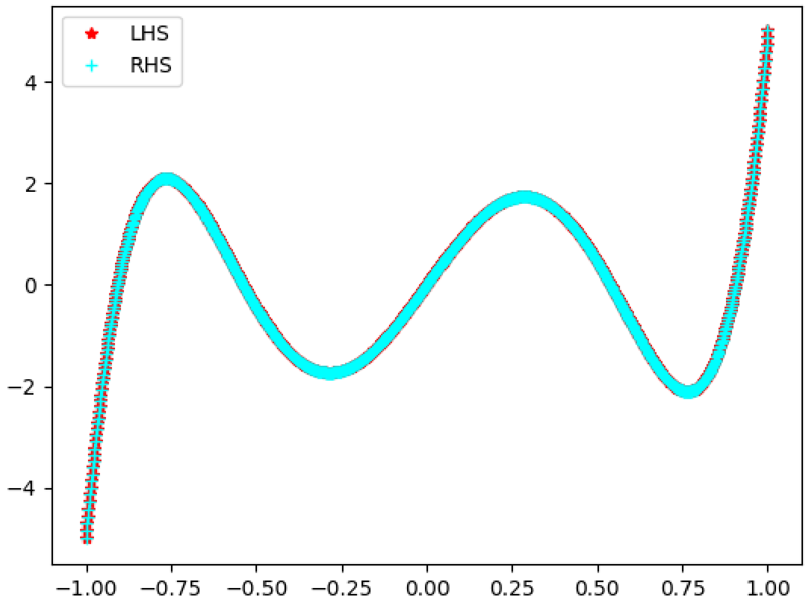

Figure 11: Plot of the relation

12 Legendre Recursion Formula-2

THEORY: To plot the given legendre recursion relation, we use the module `legendre` from `scipy.special`.

$$(1-x)^2 P\,'_n(x) = (n+1)xP_n(x) - (n+1)P_{n+1}(x) \qquad (13)$$

```
from scipy.special import legendre as p
import numpy as np
import matplotlib.pyplot as plt

x=np.linspace(-1,1,100)
n=5
LHS = lambda x : (1-x**2)*np.polyder(p(n))(x)
RHS = lambda x : (n+1)*x*p(n)(x)-(n+1)*p(n+1)(x)

plt.plot(x,LHS(x),'*',color="red",label="LHS")
plt.plot(x,RHS(x),'+',color="yellow",label="RHS")
plt.legend()
plt.show()
```

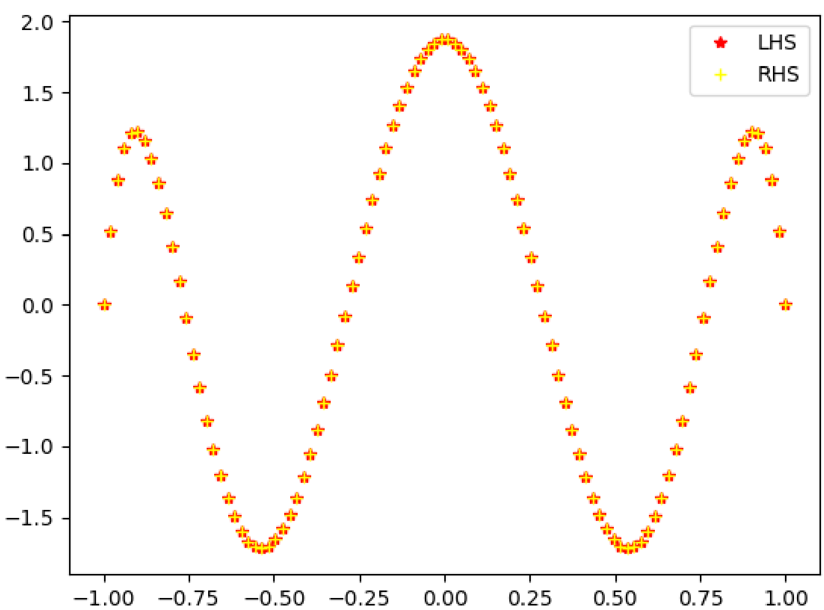

Figure 12: Plot of the relation

13 Orthogonality of Legendre Polynomial

THEORY: To check the Orthogonality relations of Legendre Polynomials, we use the module `legendre` from `scipy.special` and `simps()` from `scipy.integrate`.

$$\int_{-1}^{1} P_n(x)P_m(x)dx = \begin{cases} 0, & m \neq n, \\ \frac{2}{2n+1}, & m = n, \end{cases} \tag{14}$$

```
from scipy.special import legendre as P
import numpy as np
from scipy.integrate import simps as S

n=int(input("Give the value of n: "))
m=int(input("Give the value of m: "))
x=np.linspace(-1.0,1.0,1001)
y=(P(n)(x))*(P(m)(x))
I=S(y,x)
print(I)
```

```
OUTPUT-1
Give the value of n: 5
Give the value of m: 5
0.18181818364742836
```

```
OUTPUT-2
Give the value of n: 5
Give the value of m: 6
0.0
```

14 Convolution 1

THEORY: Verifying that the convolution of two Gaussian functions is also Gaussian.

$$f(x) = e^{-(x-2)^2/2} \tag{15}$$

$$g(x) = e^{-(x-1)^2/2} \tag{16}$$

Convolution Formula:

$$S = \int_{-x}^{+x} f(x-\tau)g(x)d\tau \tag{17}$$

```
from scipy.special import legendre as p
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import simps

x=np.linspace(-10,10,1000)

f = lambda x: np.exp((-(x-2)**2)/2)
g = lambda x: np.exp((-(x-1)**2)/2)

S = []
for i in range(len(x)):
    t = np.linspace(-x[i],x[i],1000)
    I= simps(f(x[i]-t)*g(x),t)
    S.append(I)
plt.plot(x,S)
plt.show()
```

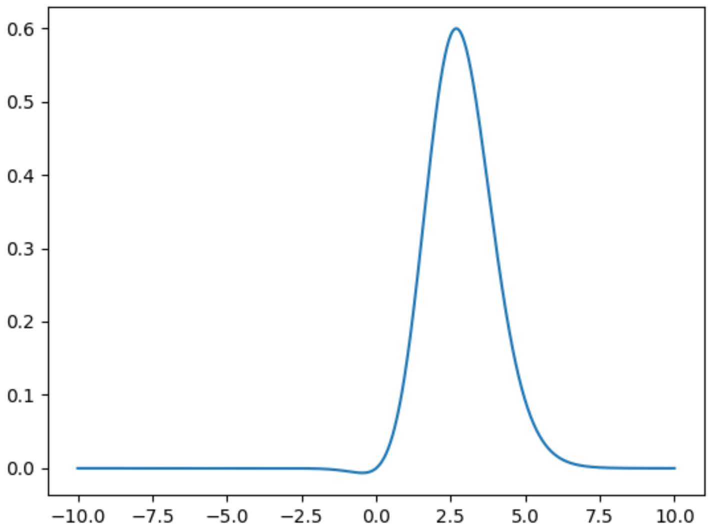

Figure 13: Plot of convolution of the functions

15 Convolution 2

THEORY: Checking the convolution of the two functions:

$$f(x) = e^{-x} \tag{18}$$
$$g(x) = sin(x) \tag{19}$$

Convolution Formula:

$$S = \int_{-x}^{+x} f(x-\tau)g(x)d\tau \tag{20}$$

```python
import numpy as np
from scipy.integrate import simps
import matplotlib.pyplot as plt
def f(x): return np.exp(-x)
def g(x): return np.sin(x)
x=np.linspace(0,20,101)
R=[]
for i in range(len(x)):
    t=np.linspace(-x[i],x[i],101)
    S=simps(f(x[i]-t)*g(t),t)
    R.append(S)
actual=(np.exp(-x)+np.sin(x)-np.cos(x))/2
plt.xlabel(r'$x$')
plt.ylabel(r'$R$')
plt.plot(x,R, label="convoluted")
plt.scatter(x,actual, label="actual")
plt.legend()
plt.show()
```

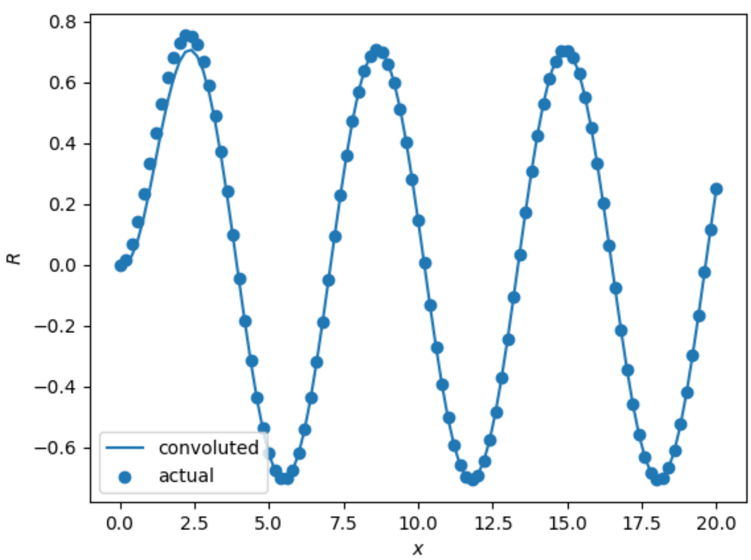

Figure 14: Plot of convolution of the functions

16 Bessel Polynomial

THEORY: Bessel functions are the solutions of the following second order differential equation,

$$x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - n^2)y = 0 \tag{21}$$

Bessel functions of the first kind,

$$J_n(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!\,\Gamma(m+n+1)}\left(\frac{x}{2}\right)^{2m+n} \tag{22}$$

In the above, $\Gamma(..)$ is a Gamma function.
$J_{-n}(x) = (-1)^n J_n(x)$
Bessel functions of the first kind,

$$Y_n(x) = \frac{J_n(x)cos\, n\pi - J_{-n}(x)}{sin\, n} \tag{23}$$

To plot the bessel polynomials, we use the module `jv` from `scipy.special`.

```
n=np.arange(0,6,1)
import numpy as np
from scipy.special import jv
import matplotlib.pyplot as plt
x=np.linspace(-0,11.0,101)
for i in n:
    plt.plot(x,jv(i,x), label='$J_{}(x)$'.format({i}))
    plt.xlabel('$x$')
    plt.ylabel('$J_n(x)$')
    plt.legend()
plt.show()
```

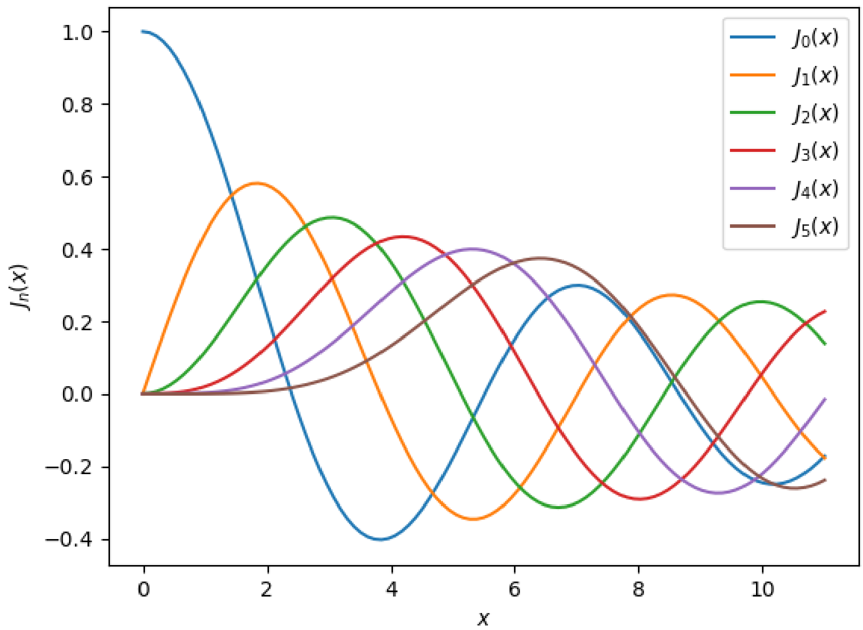

Figure 15: Plot of the Bessel Polynomials

17 Bessel Recursion Relation 1

THEORY: To plot the given bessel recursion relation, we use the module `jv` and `jvp` from `scipy.special`:

$$nJ_n(x) + xJ\,'_n(x) = xJ_{n-1}(x) \tag{24}$$

```
n=int(input("enter value of n: "))
import numpy as np
from scipy.special import jv,jvp
import matplotlib.pyplot as plt
x=np.linspace(-21.0,21.0,101)
L=n*jv(n,x)+x*jvp(n,x)
R=x*jv(n-1,x)
plt.plot(x,L,label="L")
plt.plot(x,R,'*', label="R")
plt.xlabel('$x$')
plt.ylabel('$L & R$')
plt.legend()
plt.show()
```

```
OUTPUT
enter value of n: 5
```

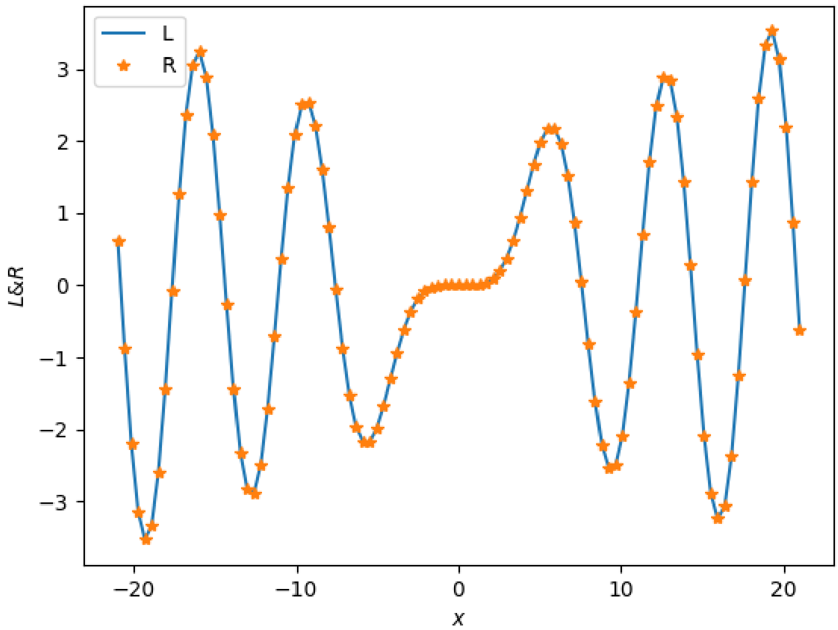

Figure 16: Plot of the function

18 Bessel Recursion Relation 2

THEORY: To plot the given bessel recursion relation, we use the module `jv` and `jvp` from `scipy.special`:

$$x^{-n}J\,'_n(x) - nx^{-n-1}J_n(x) = x^{-n}J_{n+1}(x) \tag{25}$$

```
n=int(input("enter value of n: "))
import numpy as np
from scipy.special import jv,jvp
import matplotlib.pyplot as plt
x=np.linspace(-21.0,21.0,101)
L=(x**(-n))*jvp(n,x)-n*x**(-n-1)*jv(n,x)
R=-x**(-n)*jv(n+1,x)
plt.plot(x,L,label="L")
plt.plot(x,R,'*', label="R")
plt.xlabel('$x$')
plt.ylabel('$L & R$')
plt.legend()
plt.show()
```

```
OUTPUT
enter value of n: 5
```

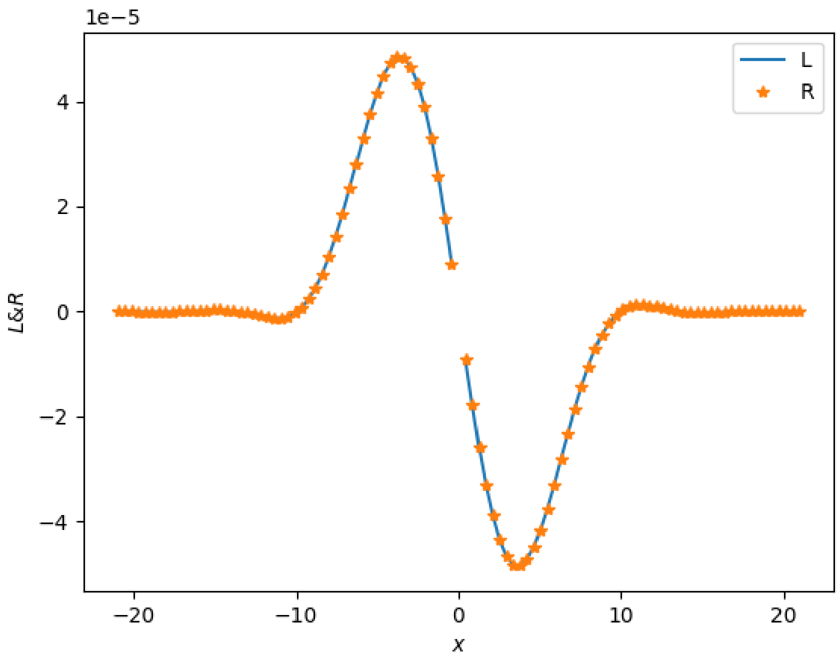

Figure 17: Plot of the function

19 Orthogonality of Bessel Polynomial

THEORY: To check the Orthogonality relations of Bessel Polynomials, we use the module `jv` from `scipy.special`, `root` from `scipy.optimize` to find the actual roots and `quad()` from `scipy.integrate` for integrating the orthogonality relation.

$$\int_0^1 xJ_n(ax)J_n(bx)dx = \begin{cases} 0, & a \neq b, \\ \frac{(J_{n+1}(a))^2}{2}, & a = b, \end{cases} \tag{26}$$

```
import numpy as np
from scipy.optimize import root
from scipy.special import jv
import matplotlib.pyplot as plt
from scipy.integrate import quad
x=np.linspace (0,20,1001)
n=int(input('enter the value of n='))
def f(x): return jv(n,x)
plt.hlines(0,0,20)
plt.grid()
plt.plot(x,f(x))
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.show()
a=float(input('enter guessroot1: '))
b=float(input('enter guessroot2: '))
c=float(input('enter guessroot3: '))
S=root(f,np.array([a,b,c])).x
print("actual roots are: ", S)

a=float(input('enter value of a: '))
b=float(input('enter value of b: '))
L= lambda x: x*jv(n,a*x)*jv(n,b*x)
R= (jv(n+1,a)**2)/2
I=quad(L,0,1)[0]
print("LHS: ",I,"RHS: ",R if a==b else 0)
```

```
OUTPUT
enter the value of n=5
```

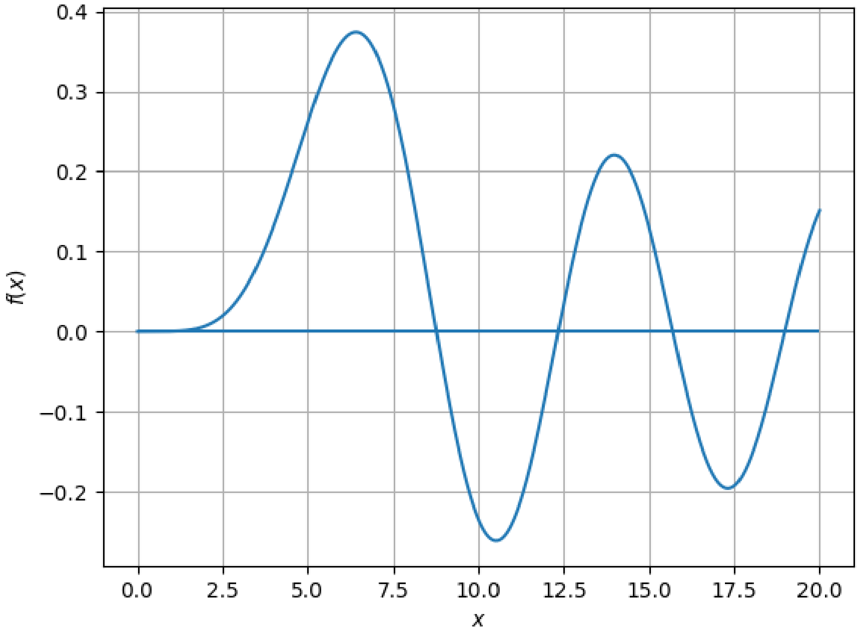

Figure 18: Plot of the function

```
enter guessroot1: 8
enter guessroot2: 12
enter guessroot3: 15
actual roots are:  [ 8.77148382 12.3386042  15.70017408]
enter value of a: 8.77
enter value of b: 8.77
LHS:  0.03012916904096064 RHS:  0.03018011768723782
```

20 Fourier Series

THEORY: Fourier expansion of a function $f(x'$ is given by:

$$f(x') = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n cos(nx') + \sum_{n=1}^{\infty} b_n sin(nx') \tag{27}$$

The Fourier coefficients are given by the integrals:

$$a_0 = \frac{1}{\pi}\int_{-\pi}^{+\pi} f(x)dx \tag{28}$$

$$a_n = \frac{1}{\pi}\int_{-\pi}^{+\pi} f(x)cos(nx)dx \tag{29}$$

$$b_n = \frac{1}{\pi}\int_{-\pi}^{+\pi} f(x)sin(nx)dx \tag{30}$$

```python
import numpy as np
from scipy.integrate import simps
import matplotlib.pyplot as plt

def f(x): return x**2

n=1
xp=np.linspace(0,20,1001)
x=np.linspace(-np.pi, np.pi, 100)
a0=(1/np.pi)*simps(f(x),x)

def a(n): return (1/np.pi)*simps(f(x)*np.cos(x),x)
def b(n): return (1/np.pi)*simps(f(x)*np.sin(x),x)

s=0.5*a0

R=[]
for i in xp:
    for n in range(1,100):
        s=s+a(n)*np.cos(n*i)+b(n)*np.sin(n*i)
    R.append(s)
plt.plot(xp,R)
plt.xlabel('$x$')
plt.ylabel('$R$')
plt.show()
```

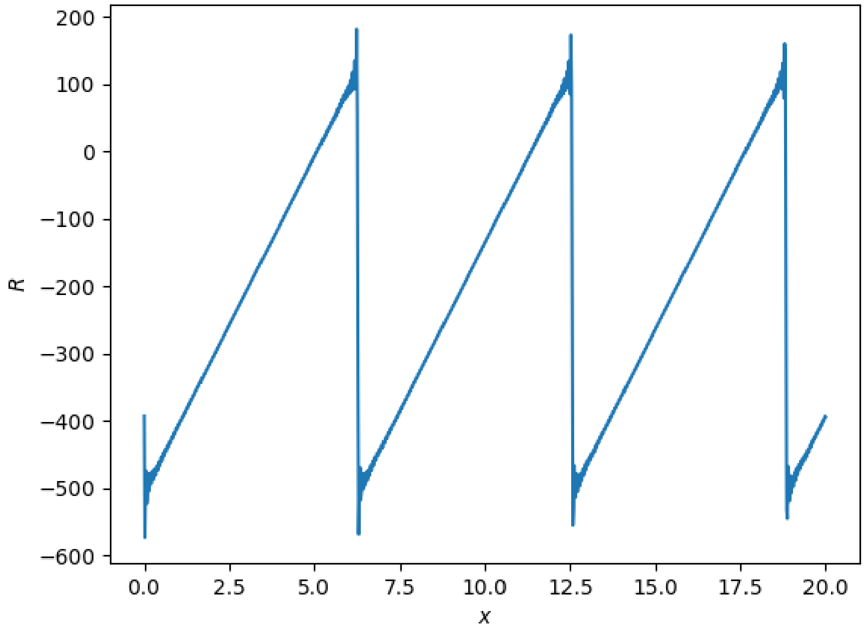

Figure 19: Plot of the function

21 Fourier Series of Square, Sawtooth and Triangular Waves

THEORY:
We demonstrate Fourier series over some piecewise continuous functions such as `square` wave, `sawtooth` wave and `triang` wave from the `scipy.signal` module.

```
from scipy.signal import square,sawtooth,triang
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import simps

L=100
f=5.0           #f=frequency, n=no. of terms
x=np.linspace(0,L,10000)

yarray=[square(2*np.pi*f*x/L), sawtooth(2*np.pi*f*x/L), triang(10000)]
i=1
labels=["Square", "Sawtooth", "Triangular"]
for y,j in zip(yarray,labels):
    a0=(2/L)*simps(y,x)
    def a(n): return (2/L)*simps(y*np.cos(2*np.pi*n*x/L),x)
    def b(n): return (2/L)*simps(y*np.sin(2*np.pi*n*x/L),x)
    s=0.5*a0
    s=s+sum([a(k)*np.cos(2*np.pi*k*x/L)+b(k)*np.sin(2*np.pi*k*x/L) for k in
    range(1,101)])
    plt.subplot(2,2,i)
    plt.plot(x,s)
    plt.plot(x,y)
    plt.title(j)
    i=i+1
plt.show()
```

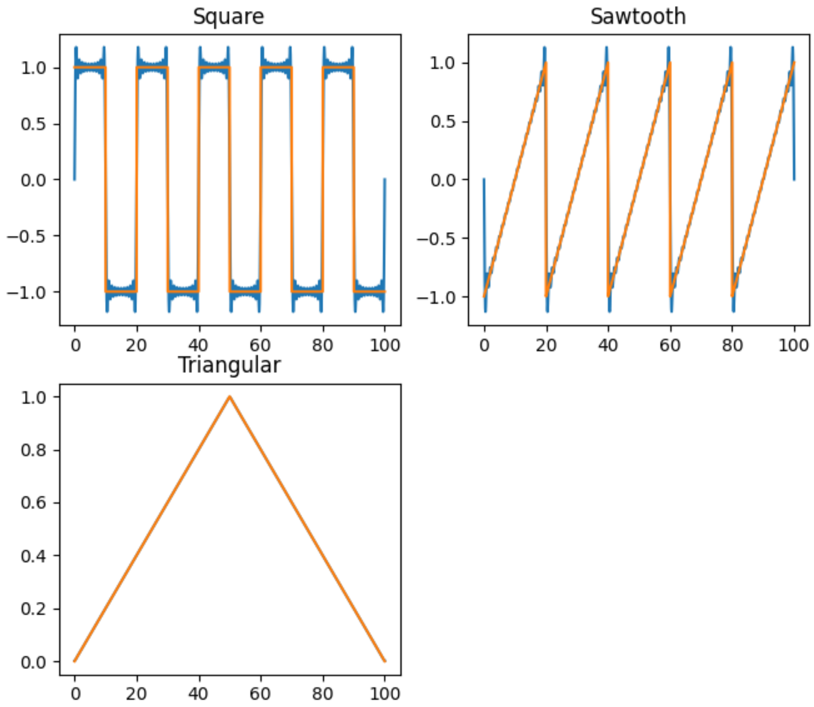

Figure 20: Plot of the function

22 1D Heat Equation

THEORY: 1D Diffusion equation or Heat equation is given by:

$$\frac{\partial u}{\partial t} = D^2 \frac{\partial^2 u}{\partial x^2} \tag{31}$$

It can reduced to the following form using *Schmidt method* (Eulerian Scheme) for simplifying the numerical calculation:

$$u_j^{i+1} = u_j^i + r(u_{j+1}^i - 2u_j^i + u_{j-1}^i) \tag{32}$$

where, $0 < r < 1/2$, for computer analysis, most often, the choice is, $r = 1/4$.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import simps

x=np.linspace(0,1,101)
u=np.zeros(101)
u[50]=1
for i in range(100):          #Time Loop
    for j in range(1,100):
        u[j] += (u[j-1] - 2*u[j] + u[j+1])/4.0
plt.plot(x,u)
plt.xlabel('length ($x$)')
plt.ylabel('amount of heat($u$)')
plt.show()
```

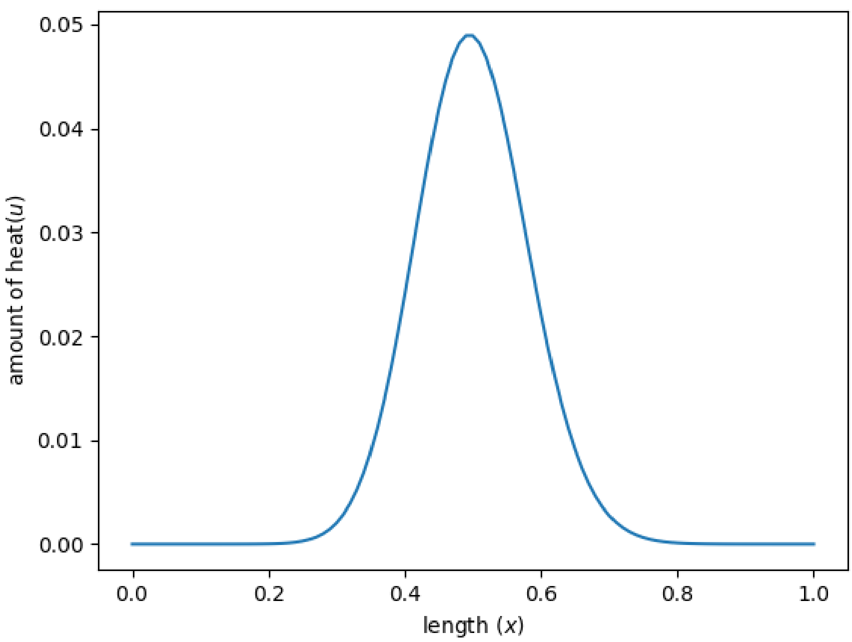

Figure 21: Plot of the function

23 3D Plot of 2D Heat Equation

THEORY: 2D Diffusion equation or Heat equation is given by:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{33}$$

It can reduced to the following recursion relation for simplifying the numerical calculation:

$$u_{i,j}^{t+1} = u_{i,j}^{t} + \frac{(u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t} - 4u_{i,j}^{t})}{4} \tag{34}$$

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import simps
from matplotlib import cm

x=np.linspace(0,1,101)
y=np.linspace(0,1,101)
u=np.zeros((101,101))
u[50,50]=1.0      #IC: Heating at middle

for t in range(1000):             #Time Loop
    for i in range(100):
        for j in range(1,100):
            u[i,j] += (u[i+1,j] + u[i-1,j] + u[i,j+1] + u[i,j-1] - 4*u[i,j])/4.0

X,Y=np.meshgrid(x,y)
plt.axes(projection="3d").plot_surface(X,Y,u, cmap=cm.jet, rstride=1, cstride=1)
plt.show()
```

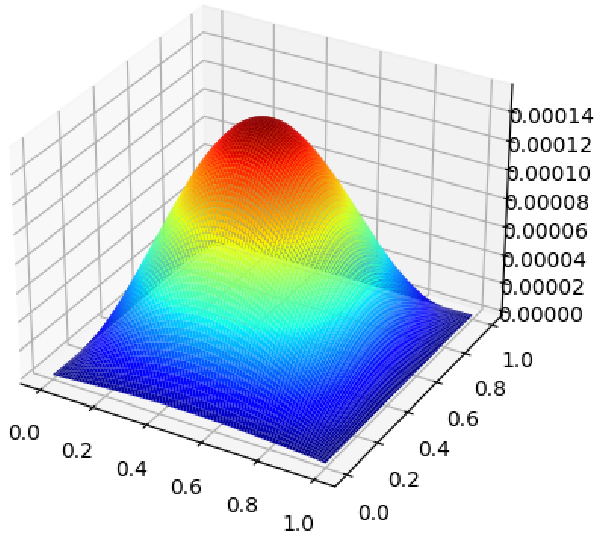

Figure 22: Plot of the function