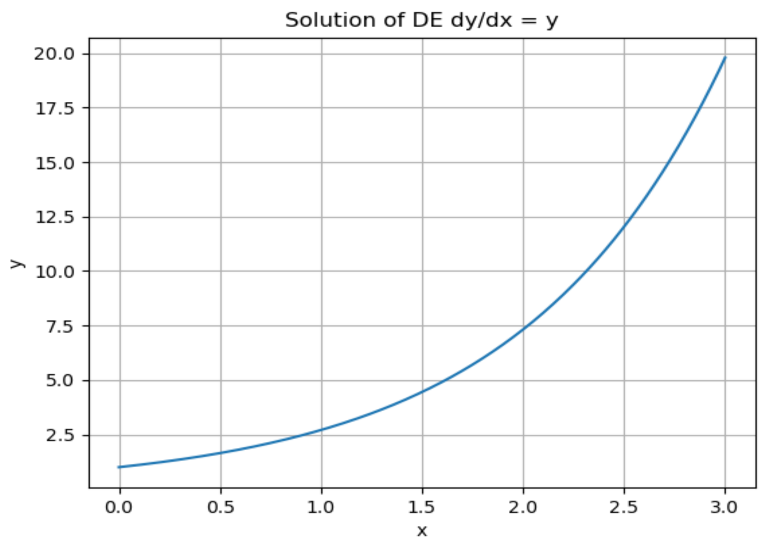
UID:2309111 ***DIFFERENTIAL EQUATIONS(ODE)*** **PRACTICAL:1**

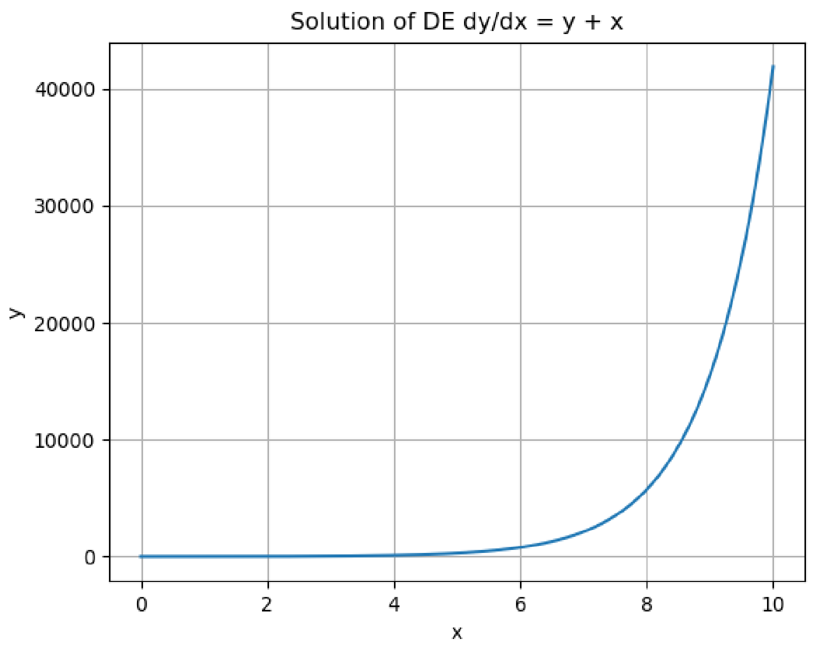
* **Example python code:**

```
import numpy as np
import matplotlib.pyplot as plt

def fxy(xf,yf):
    y = yf
    return y

a=0;b=3;h=0.01;i=0

n=(b-a)/h # width of each interval
x=[]
y =[]

intg =[] #initial value y(8) = 1
x.append(0)
y.append(1)

while i<n:
    x.append(a + h*(i+1))
    y.append(y[i]+h*fxy(x[i],y[i]))
    i+=1

print ('value of y(x) is', y[i], "at x =",x[i])
plt.plot(x,y)
plt.title('Solution of DE dy/dx = y')
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.show()
```



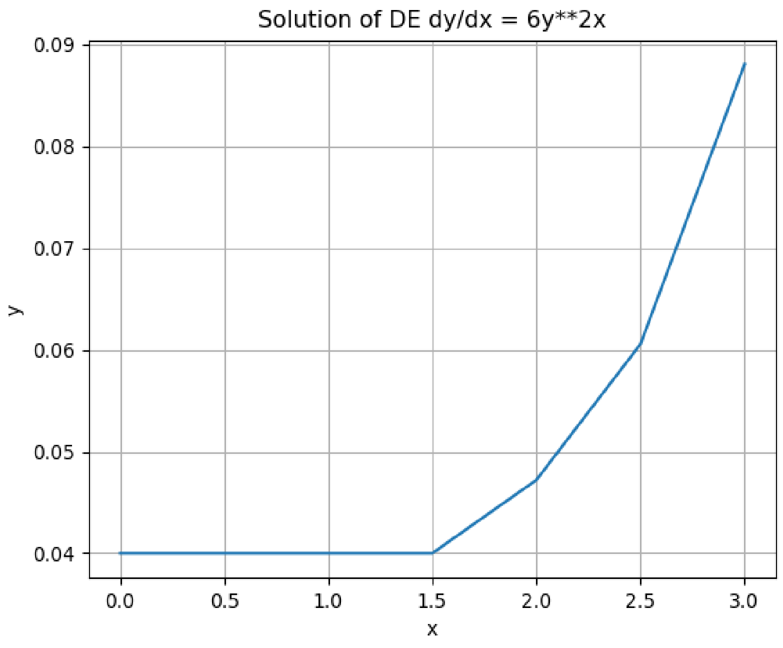
*** 1(a):For step size(n) = 0.001 equation is (dy/dx)=y+x**

```
import numpy as np
import matplotlib.pyplot as plt

def fxy(xf,yf):
    y = xf + yf
    return y

a =0;b = 10;h = 0.01;i =0
n = (b-a)/h # width of each interval
x =[]
y =[]
intg =[]
x.append(0)
y.append(1)

while i<n:
    x.append(a + h*(i+1))
    y.append(y[i]+h*fxy(x[i],y[i]))
    i+=1

print ('value of y(x) is', y[i], "at x =",x[i])
plt.plot(x,y)
plt.title("Solution of DE dy/dx = y + x")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.show()
```



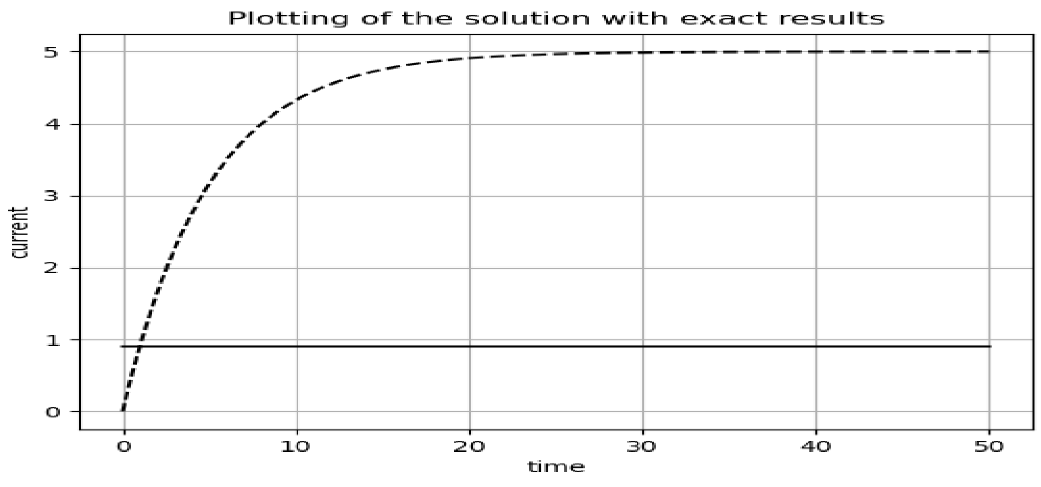
* 1(b): For step size(n)=0.001 and equation (dy/dx)=6y^{2}x

```
import numpy as np
import matplotlib.pyplot as plt

def fxy(xf,yf):
    y = 6*(yf**2)*xf
    return y

a =1;b = 3;h = 0.5;i =0
n = (b-a)/h # width of each interval
x =[]
y =[]
intg =[] #initial value y(8) = 1
x.append(0)
y.append(1/25)

while i<n:
    x.append(a + h*(i+1))
    y.append(y[i]+h*fxy(x[i],y[i]))
    i+=1

print ('value of y(x) is', y[i], "at x =",x[i])
plt.plot(x,y)
plt.title("Solution of DE dy/dx = 6y**2x")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.show()
```



* L.R Circuit

```
import numpy as np
import matplotlib.pyplot as plt

e = 1;r= 0.2;l= 1.;t= 0;i= 0  #intial values
tt= []
ii= []
tf= 50.0 #final value of t
dt= 0.01 #step size

def f(t,i):
    y = e-i*r
    return y

while t<=tf:
    tt.append(t)
    ii.append(i)
    i+=dt* f(t, i)
    t=t+dt

tt = np.array((tt))
ii = np.array((ii))
exact = (e/r)*(1.0 - np.exp(-r/1**tt))

print ("Plotting of the solution with exact results")
#plt.subplot(1,1,1)
plt.title("Plotting of the solution with exact results")
plt.plot(tt,ii, 'k--',label="dt=%.4Ff"%(dt))
plt.plot(tt,exact, 'k',label="Exact Solution")
plt.xlabel('time')
plt.ylabel('current')
plt.grid()
plt.show()
```



<u>UID:2309111</u> **<u>PLOTTING CONIC SECTIONS USING PYTHON</u>** *<u>Practical:2</u>*

Aim: To learn two-dimensional and three-dimensional graphs in Python and plot a circle, ellipse, hyperbola and parabola using Python.

Python Methods:
***<u>NumPy</u>*:**
numpy.cos(th) and numpy.sin(th): These functions compute the cosine and sine of an array of angles th. In the code, they are used to calculate the x and y coordinates for circles, ellipses, and other curves based on the trigonometric functions.

numpy.sqrt(const/x_coff) and numpy.sqrt(const/y_coff): These functions calculate the square root of the specified values. In the code, they are used to compute the values of 'a' and 'b' for ellipses, as well as other mathematical operations.

numpy.linspace(start, end, num): This function returns evenly spaced values over a specified range. It is used to generate values for 'th' (theta) to create curves with many data points.
numpy.square(th): This function calculates the square of each element in the array 'th'. In the code, it is used to create parabolic curves.

numpy.meshgrid(x, y): This function creates a grid of points from two arrays 'x' and 'y'. It is used to create 3D surfaces for the paraboloid plot.

***<u>Matplotlib.pyplot</u>*:**

matplotlib.pyplot.figure(): This function creates a new figure for plotting. It can specify the figure size, among other properties.

matplotlib.pyplot.plot(x, y): This function is used to plot a curve by specifying the x and y coordinates of the data points.

matplotlib.pyplot.scatter(x, y): This function is used to plot individual data points as scatter points. In the code, it is used to plot the Sun as a yellow point.

matplotlib.pyplot.xlabel(label), matplotlib.pyplot.ylabel(label): These functions are used to label the x and y axes of the plot, respectively.

matplotlib.pyplot.title(title): This function sets the title of the plot.

matplotlib.pyplot.xlim(min, max), matplotlib.pyplot.ylim(min, max): These functions set the limits for the x and y axes of the plot, defining the visible range of the data.

matplotlib.pyplot.legend(): This function is used to add a legend to the plot, labeling different curves or data series.

matplotlib.pyplot.gca().set_aspect('equal', adjustable='box'): This code is used to ensure that the aspect ratio of the plot is equal, preventing distortion of the shapes.

matplotlib.pyplot.grid(True): This function adds a grid to the plot for better visualization.

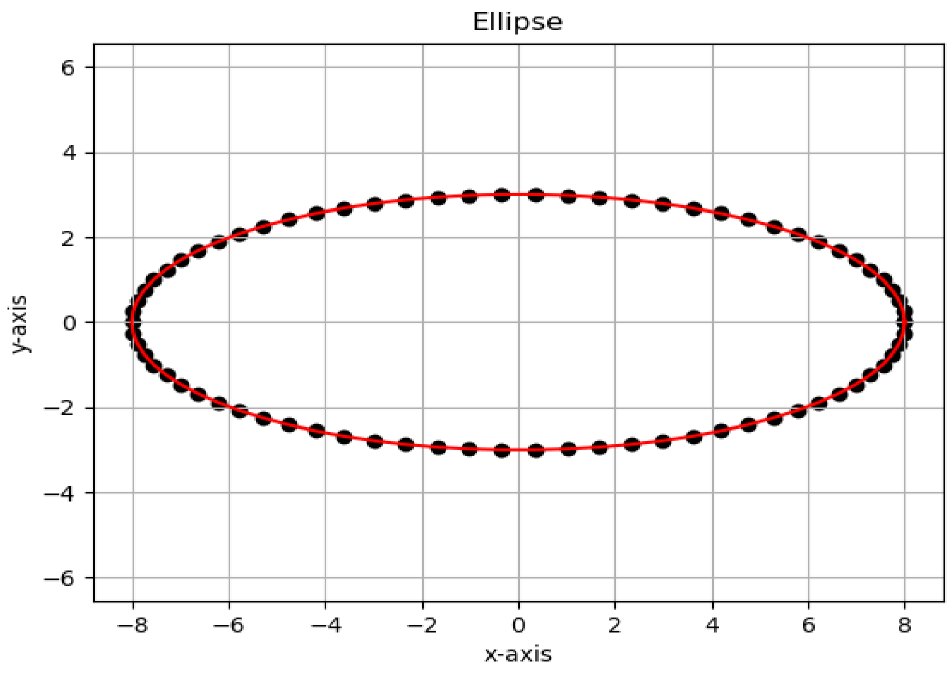
matplotlib.pyplot.show(): This function displays the plot on the screen.

Conclusion:
A circle, ellipse, hyperbola, parabola and the orbit of the Moon around the Earth have been plotted using Python.

PLOTTING CONIC SECTIONS USING PYTHON

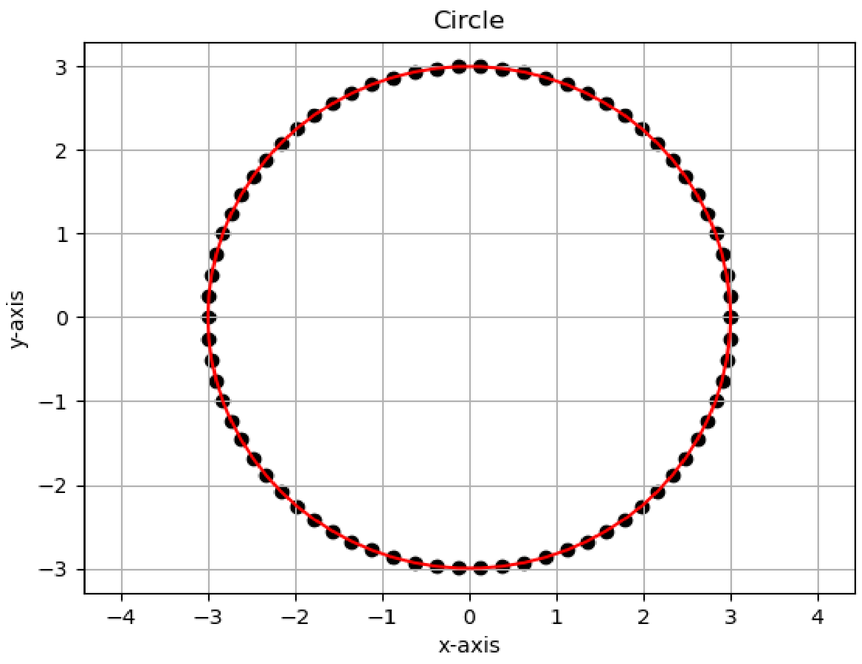
* **ELLIPSE CODE:**

```
import numpy as np
import matplotlib.pyplot as plt

#creating a parameter theta array
th = np.linspace(0,2*np.pi,75)

#initializing the constants for the given equation to be plot
a=8;b=3

#co-ordinates for the center of the ellipse
xcord = 0;ycord = 0

#creating parametric equation of x and y arrays from theta

x = xcord + a*np.cos(th)
y = ycord + b*np.sin(th)

#plotting the ellipse
plt.plot(x,y, color = 'red')
plt.scatter(x,y, color = 'black')
plt.axis('equal')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('Ellipse')
plt.grid()
plt.show()
```



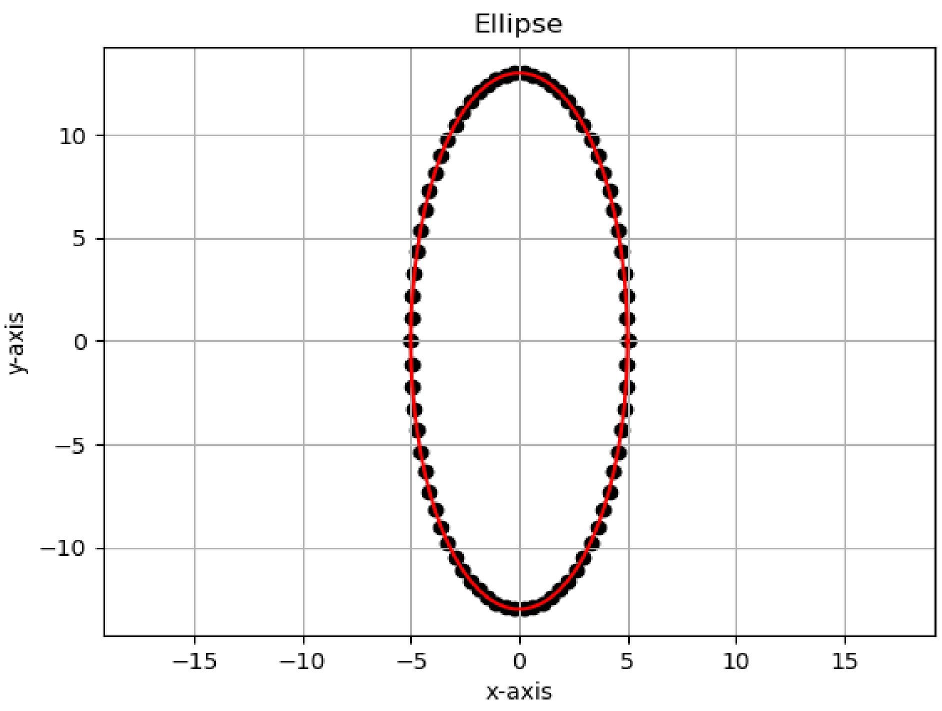
* **Circle with radius(r)=3**

```
import numpy as np
import matplotlib.pyplot as plt

#creating a parameter theta array
th = np.linspace(0,2*np.pi,75)

#initializing the constants for the given equation to be plot
a=3;b=3
#co-ordinates for the center of the Circle
xcord = 0;ycord = 0

#creating parametric equation of x and y arrays from theta
x = xcord + a*np.cos(th)
y = ycord + b*np.sin(th)

#plotting the Circle
plt.plot(x,y, color = 'red')
plt.scatter(x,y, color = 'black')
plt.axis('equal')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('Circle')
plt.grid()
plt.show()
```



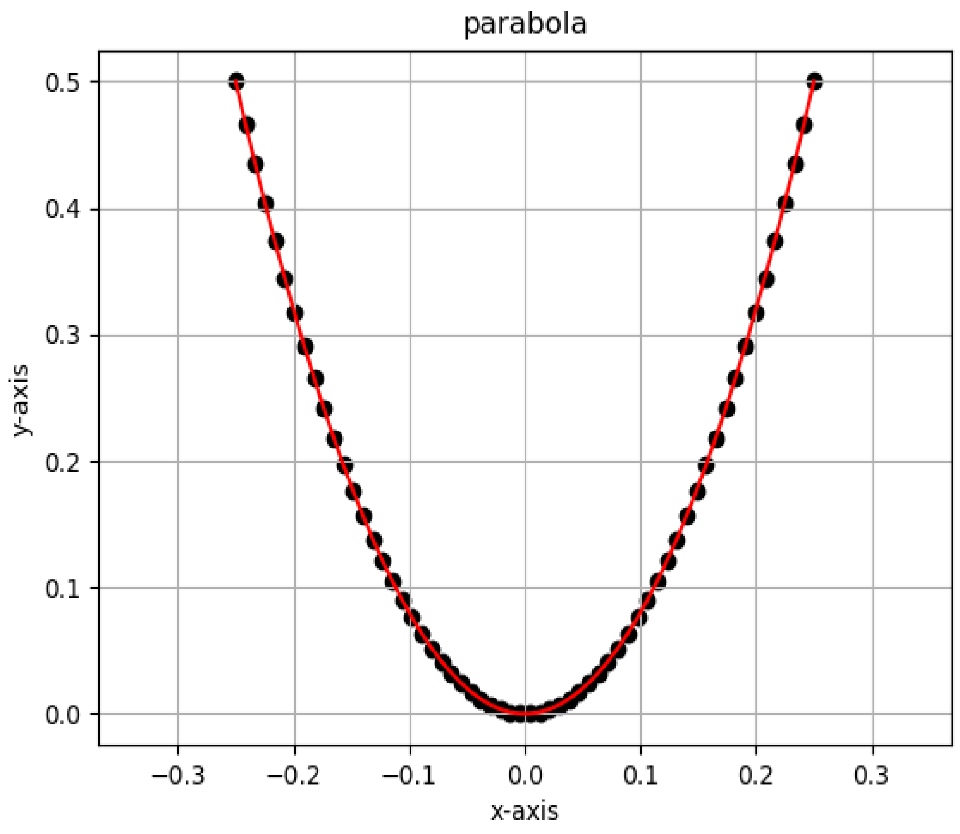
* **Plot of an ellipse $169x^2 + 25y^2 = 4225$.**

```
import numpy as np
import matplotlib.pyplot as plt

#creating a parameter theta array
th = np.linspace(0,2*np.pi,75)

#initializing the constants for the given equation to be plot
a=5;b=13

#co-ordinates for the center of the ellipse
xcord = 0;ycord = 0

#creating parametric equation of x and y arrays from theta

x = xcord + a*np.cos(th)
y = ycord + b*np.sin(th)

#plotting the ellipse
plt.plot(x,y, color = 'red')
plt.scatter(x,y, color = 'black')
plt.axis('equal')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('Ellipse')
plt.grid()
plt.show()
```



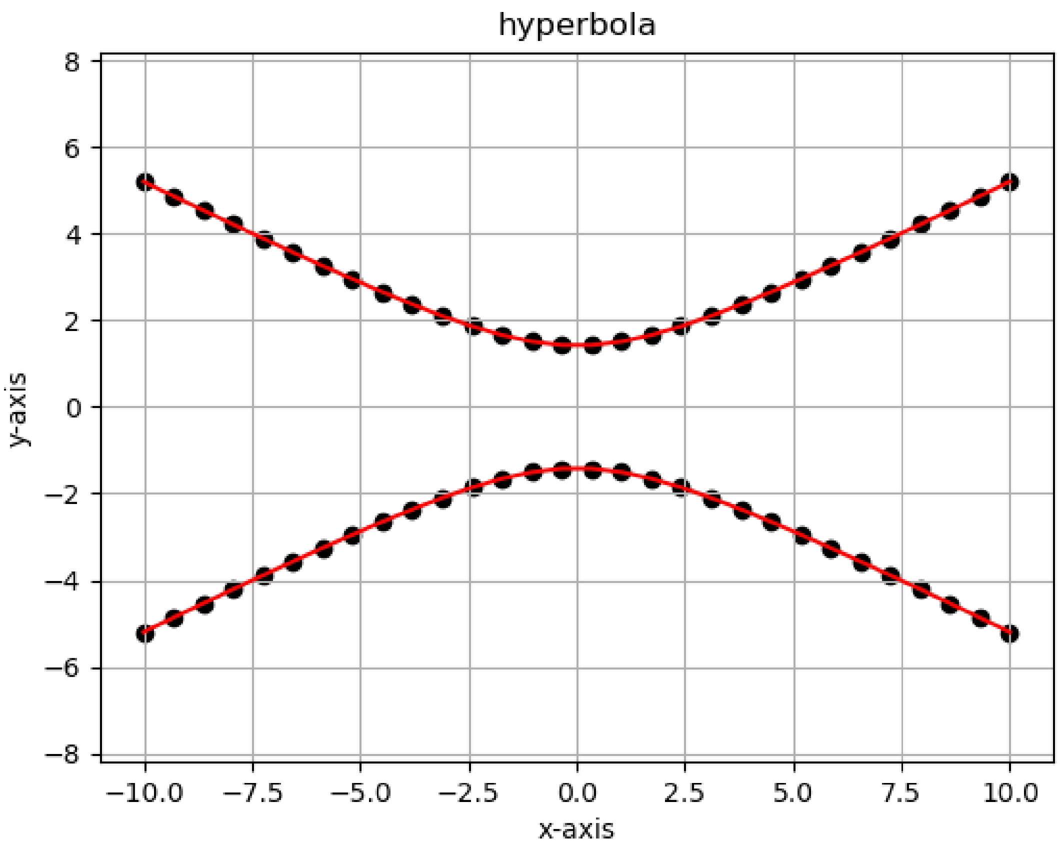
- **Plot of a parabola with focus at (0,2) and directrix at y=-2**

```
import numpy as np
import matplotlib.pyplot as plt

#creating a parameter theta array
x = np.linspace(-0.25,0.25,60)

#initializing the constants for the given equation to be plot
a=2

y=4*a*x**2      #eqn for parabola

#plotting the parabola
plt.plot(x,y, color = 'red')
plt.scatter(x,y, color = 'black')
plt.axis('equal')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('parabola')
plt.grid()
plt.show()
```



- **Hyperbola plot for $8y^2-2x^2=16$**

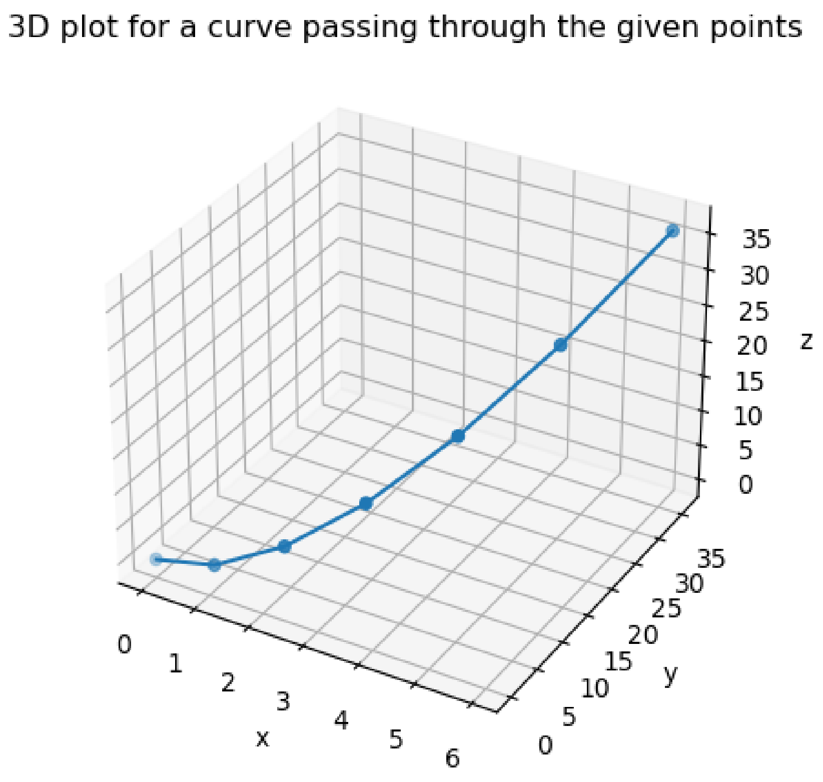
```
import numpy as np
import matplotlib.pyplot as plt

#creating a parameter theta array
x = np.linspace(-10,10,30)

a=np.sqrt(2+x**2/4)
b=-np.sqrt(2+x**2/4)

#plotting the hyperbola
plt.plot(x,a, color = 'red')
plt.scatter(x,a, color = 'black')
plt.plot(x,b, color = 'red')
plt.scatter(x,b, color = 'black')
plt.axis('equal')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('hyperbola')
plt.grid()
plt.show()
```



PLOTTING 3D GRAPHS

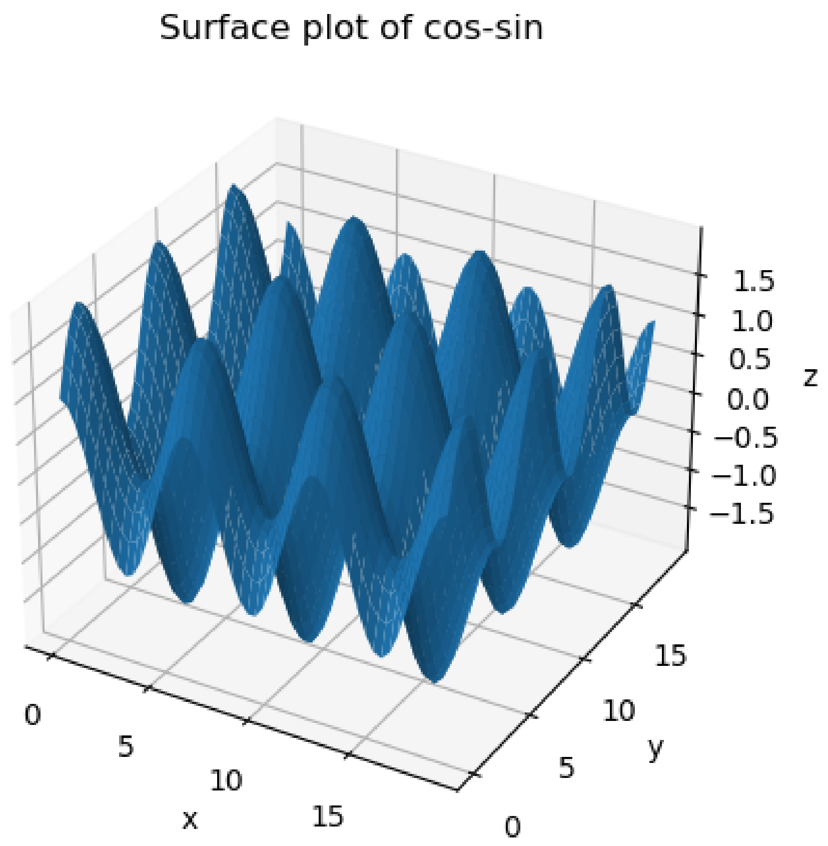
Example(1): Plot a curve for the following set of points

```
import numpy as np
import matplotlib.pyplot as plt

#creating an empty canvas
fig = plt.figure()
ax = plt.axes(projection ='3d')

x=[0,1,2,3,4,5,6]
y=[0,1,4,9,16,25,36]
z=[0,1,4,9,16,25,36]

#syntax for the curve plotting
ax.plot3D(x,y,z)
ax.scatter(x,y,z)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('3D plot for a curve passing through the given points')
plt.show()
```



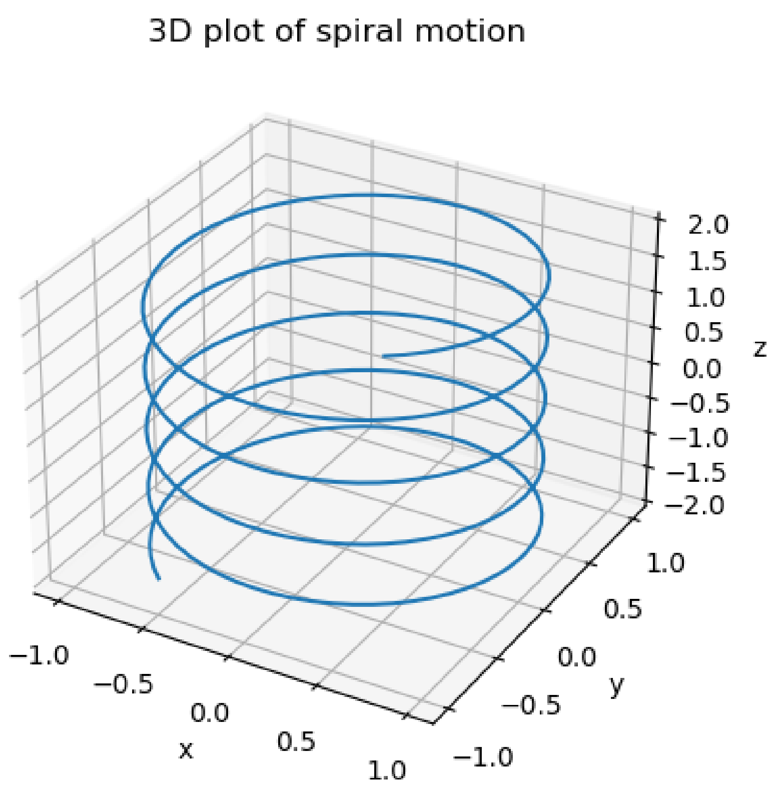
Example(2): **Plot a Surface for parametric equation Z=cos(x)+sin(y) for 0≤x,y≤2π**

```
import numpy as np
import matplotlib.pyplot as plt

#defining surface and axes

x  = np.linspace(0,6*np.pi,50)
y  = np.linspace(0,6*np.pi,50)
x,y= np.meshgrid(x,y)
z  =np.cos(x) + np.sin(y)

#syntax for thev3D plotting
ax = plt.axes(projection = '3d')

#syntax for the surface plotting
ax.plot_surface(x,y,z)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Surface plot of cos-sin')
plt.show()
```



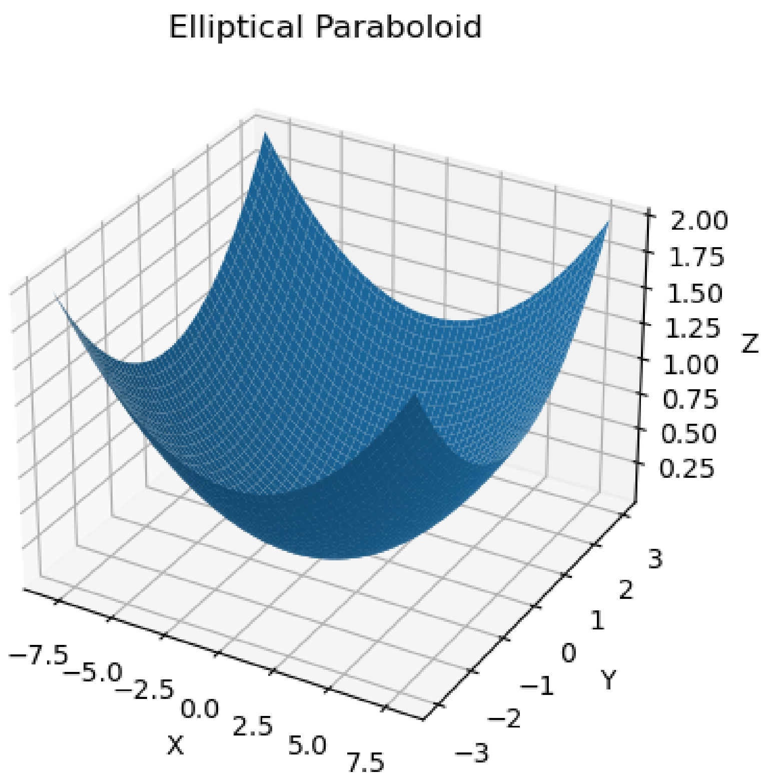
- **Graph of spiral motion with radius(r)=2 units**

```
import numpy as np
import matplotlib.pyplot as plt

#creating an empty canvas
fig = plt.figure()
ax = plt.axes(projection ='3d')

th=np.linspace(-15,15,300)

x=np.sin(th);y=np.cos(th);z=np.linspace(-2,2,300)

#syntax for the curve plotting
ax.plot3D(x,y,z)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('3D plot of spiral motion')
plt.show()
```



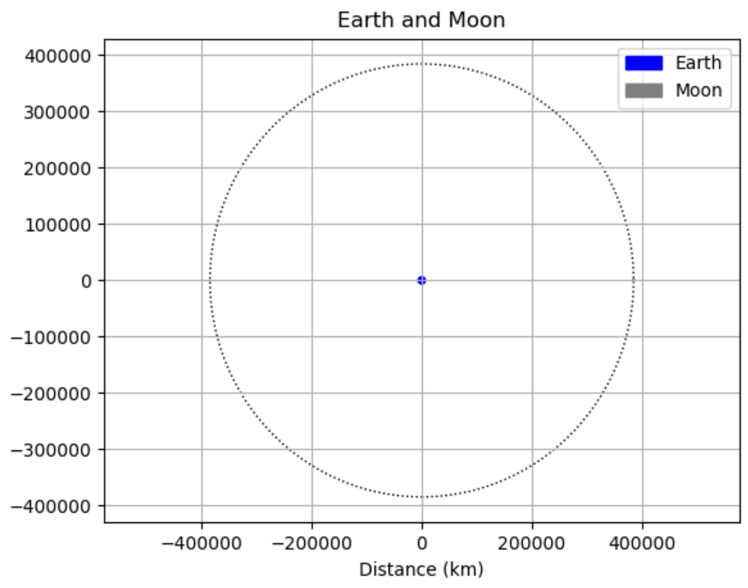
- **Plot a suraface of an elliptial paraboloid**

```
import numpy as np
import matplotlib.pyplot as plt

#grid creation
x=np.linspace(-8,8,100)
y=np.linspace(-3,3,100)
X,Y=np.meshgrid(x,y)

a=8;b=3

Z=(X/a)**2+(Y/b)**2

#plotting the surface
fig=plt.figure()
ax=fig.add_subplot(111, projection='3d')
surf=ax.plot_surface(X,Y,Z)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Elliptical Paraboloid')
plt.show()
```



<u>Earth and moon system code:</u>

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Circle

fig, ax = plt.subplots()

# Define Earth's properties
earth_radius = 6371  # Earth's radius in kilometers
earth_color = 'blue'

# Define Moon's properties
moon_radius = 1737  # Moon's radius in kilometers
moon_distance = 384400  # Moon's distance from Earth in kilometers
moon_color = 'gray'

# Plot Earth as a circle
earth_circle = Circle((0, 0, 0), earth_radius, color=earth_color, label='Earth')
ax.add_patch(earth_circle)

# Plot Moon's orbit
moon_orbit = plt.Circle((0, 0, 0), moon_distance, color='black', fill=False,
linestyle='dotted')
ax.add_artist(moon_orbit)

# Plot Moon as a circle
moon_circle = Circle((moon_distance, 0), moon_radius, color=moon_color, label='Moon')
ax.add_patch(moon_circle)

# Set aspect ratio to be equal, so the circle appears as a circle
ax.set_aspect('equal', adjustable='datalim')

# Set plot limits based on Moon's distance
ax.set_xlim(-moon_distance * 1.5, moon_distance * 1.5)
ax.set_ylim(-moon_distance * 1.5, moon_distance * 1.5)
ax.legend()

plt.xlabel('Distance (km)')
plt.ylabel('Distance (km)')
plt.title('Earth and Moon')
plt.grid()
plt.show()
```



UID:2309111 *NUMERICAL INTEGRATION USING PYTHON * PRACTICAL:3

Aim: To implement and compare two numerical integration methods, the Trapezoidal Rule and the Midpoint Rule, using Python, for approximating definite integrals.

Theory:

Trapezoidal Rule: The Trapezoidal Rule approximates the integral of a function f(x) over the interval [a,b] by dividing the area under the curve into trapezoids and summing their areas.

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(a+ih) + f(b)\right]$$

Where, h=width of each sub-intervals; n= number of sub-intervals, a and b = the limits of integration.

Midpoint Rule: The Midpoint Rule estimates the integral of f(x) over [a,b] by calculating the areas of rectangles whose heights are determined by the function's values at the midpoints of each subinterval.

$$\int_a^b f(x)dx \approx h\sum_{i=1}^{n} f\left(a + \frac{h}{2} + (i-1)h\right)$$

Where, h=width of each sub-intervals; n= number of sub-intervals, a and b = the limits of integration.

Conclusion:

We implemented and compared two numerical integration methods, the Trapezoidal Rule and the Midpoint Rule, using Python.
The Trapezoidal Rule offers versatility and moderate accuracy, while the Midpoint Rule provides computational efficiency but may be less accurate for certain functions.
Choosing between these methods depends on the specific problem's requirements, allowing us to make informed decisions for numerical integration in Python.

PYTHON CODE

Code 1(a):Midpoint Method

```
import matplotlib.pyplot as plt
from math import *
import numpy as np

def func(x):
    y= x**2+1
    return y
a=-2;b=4;n=5;sum=0;i=0
dx=(b-a)/n
x=[];y=[];Z=[]

while i<n:
    midpt=a+(i+1/2)*dx
    x.append(midpt)
    fn=func(midpt)
    y.append(fn)
    sum = sum+y[i]*dx
    Z.append(sum)
    i=i+1
print("integrated value is =", round(sum,3))
plt.figure()
plt.subplot(2,1,1)
plt.plot(x,y)
plt.title('graph of f(x)=x and $\int f(x)$')
plt.xlabel('x')
plt.ylabel('f(x)')

plt.subplot(2,1,2)
plt.plot(x,Z)
plt.xlabel('x')
plt.ylabel('$\int_{a}^{b} f(x) dx$')
plt.show()
```

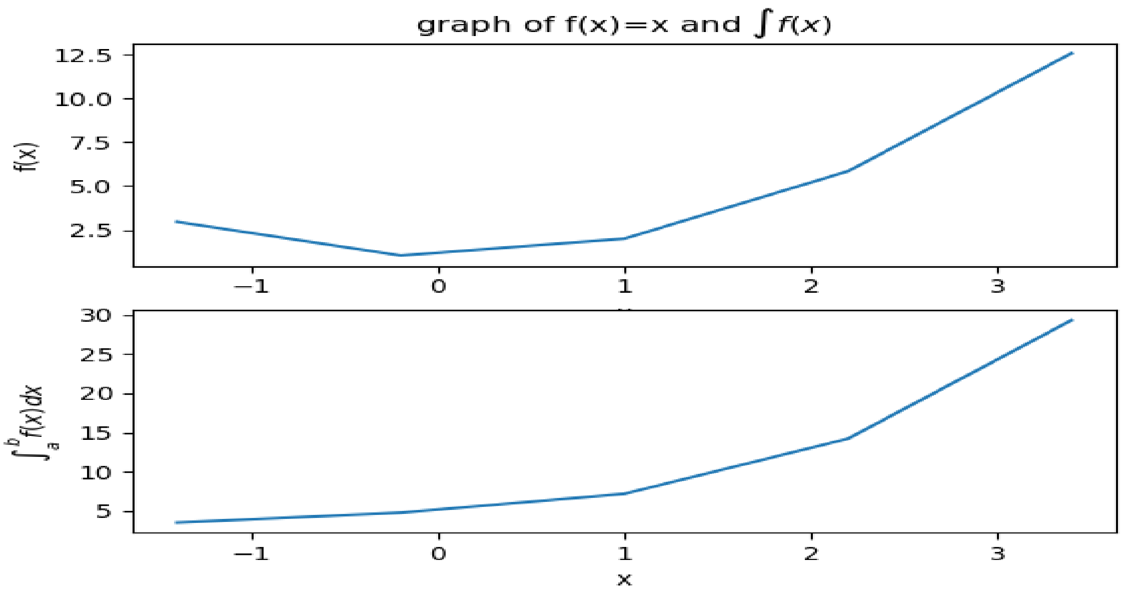
CODE 1(b): Midpoint Method

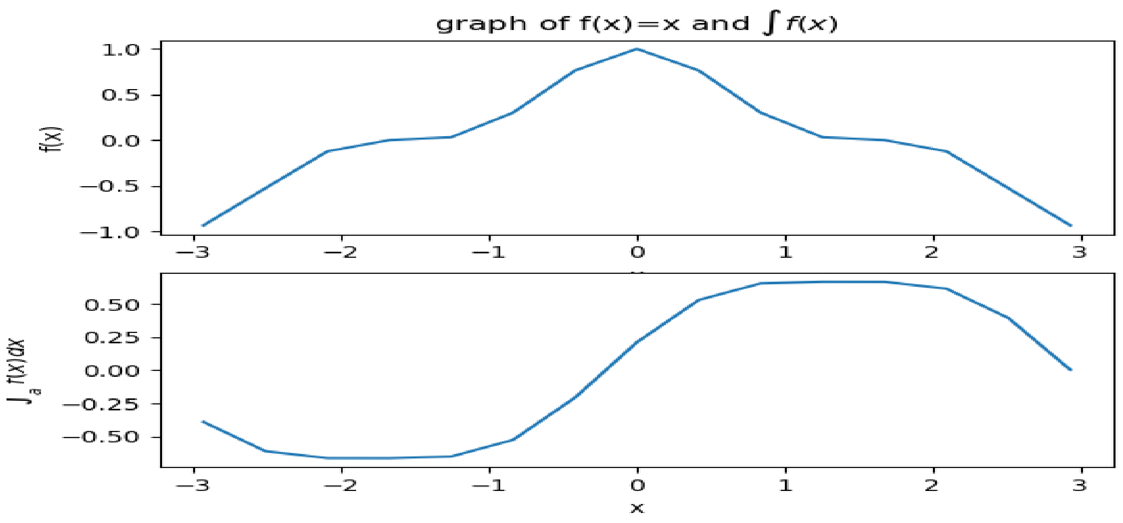
```
import matplotlib.pyplot as plt
from math import *
import numpy as np

def func(x):
    y=((np.cos(x))**3)
    return y

a=-3.14;b=3.14;n=15;sum=0;i=0
dx=(b-a)/n

x=[];y=[];Z=[]

while i<n:
    midpt=a+(i+1/2)*dx
    x.append(midpt)
    fn=func(midpt)
    y.append(fn)

    sum = sum+y[i]*dx
    Z.append(sum)
    i=i+1

print("integrated value is =", round(sum,3))

plt.figure()
plt.subplot(2,1,1)
plt.plot(x,y)
plt.title('graph of f(x)=x and $\int f(x)$')
plt.xlabel('x')
plt.ylabel('f(x)')

plt.subplot(2,1,2)
plt.plot(x,Z)
plt.xlabel('x')
plt.ylabel('$\int_{a}^{b} f(x) dx$')

plt.show()
```

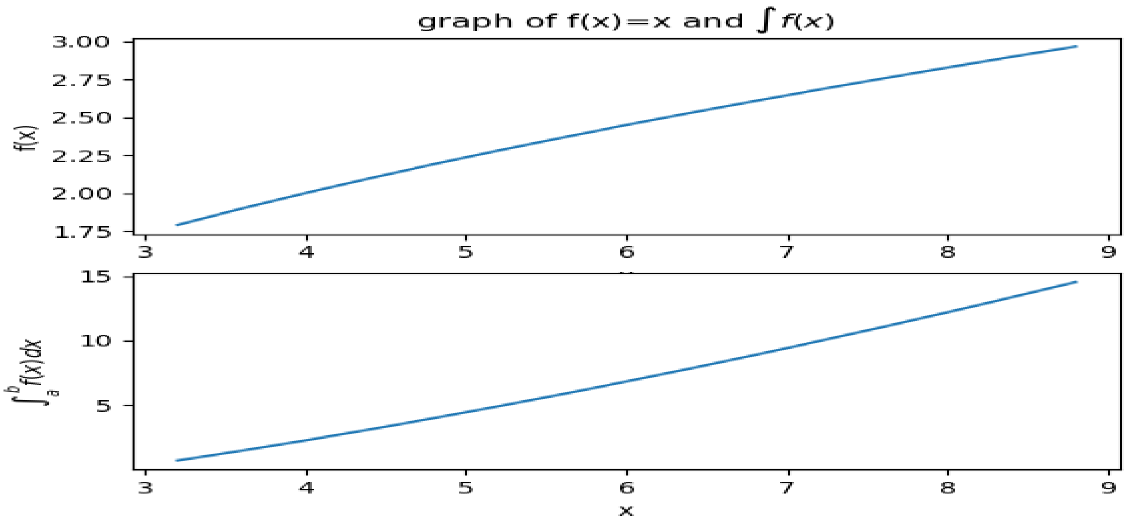
<u>CODE 1(c): Midpoint Method</u>

```
import matplotlib.pyplot as plt
from math import *
import numpy as np

def func(x):
    y=np.sqrt(x)
    return y

a=3;b=9;n=15;sum=0;i=0
dx=(b-a)/n

x=[];y=[];Z=[]

while i<n:
    midpt=a+(i+1/2)*dx
    x.append(midpt)
    fn=func(midpt)
    y.append(fn)

    sum = sum+y[i]*dx
    Z.append(sum)
    i=i+1

print("integrated value is =", round(sum,3))

plt.figure()
plt.subplot(2,1,1)
plt.plot(x,y)
plt.title('graph of f(x)=x and $\int f(x)$')
plt.xlabel('x')
plt.ylabel('f(x)')

plt.subplot(2,1,2)
plt.plot(x,Z)
plt.xlabel('x')
plt.ylabel('$\int_{a}^{b} f(x) dx$')

plt.show()
```

Results for Midpoint Method: In sequence of codes a,b,c :-



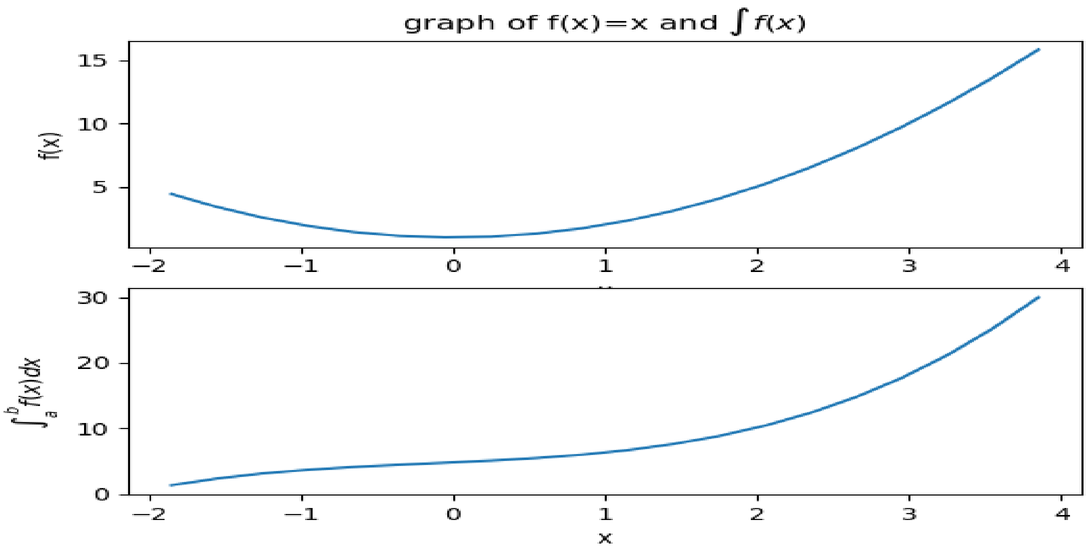




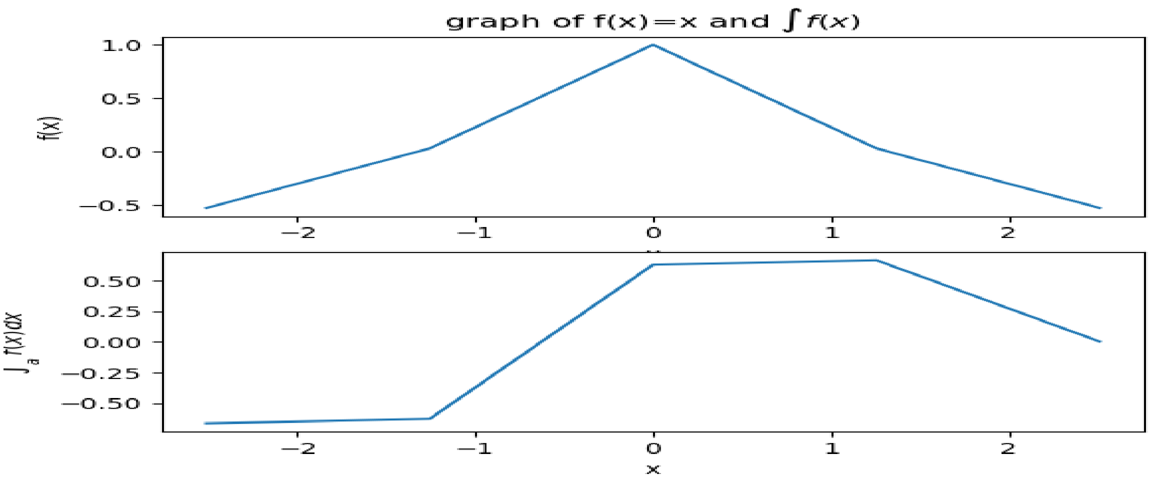
TRAPEZOIDAL METHOD PYTHON CODES:

CODE 2(a):

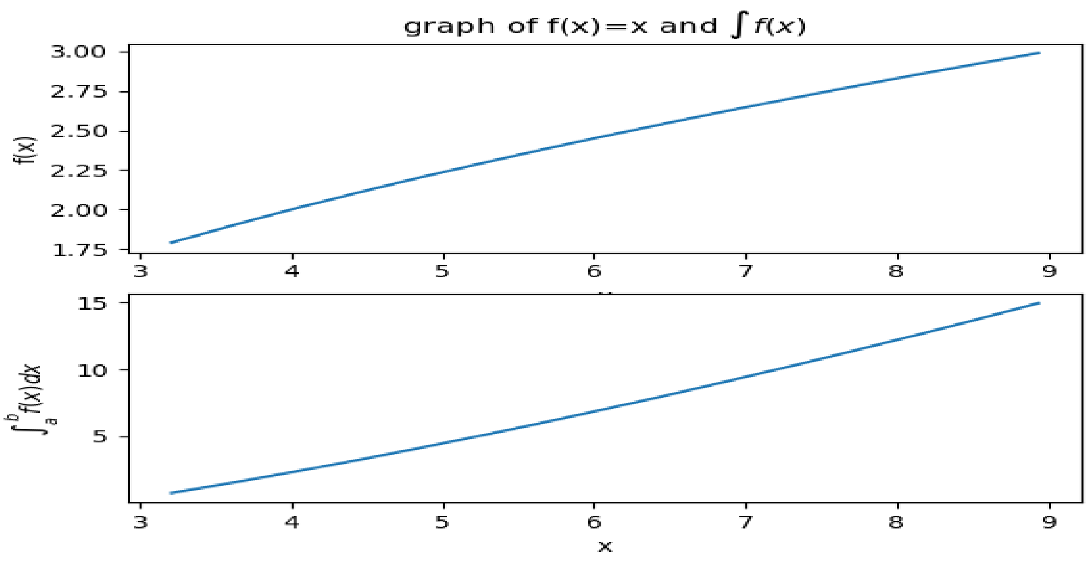
```
import matplotlib.pyplot as plt

def func(x):
    return x**2+1

a=-2;b=4;n=20;sum=0;i=0
dx=(b-a)/n

x=[];y=[];Z=[]

while i<n:
    x1= a + i * dx
    x2= x1+dx
    trapz=(x1+x2)/2
    x.append(trapz)
    fn=func(trapz)
    y.append(fn)

    sum += y[i]*dx
    Z.append(sum)
    i=i+1

print("integrated value is =", round(sum,5))

plt.figure()
plt.subplot(2,1,1)
plt.plot(x,y)
plt.title('graph of f(x)=x and $\int f(x)$')
plt.xlabel('x')
plt.ylabel('f(x)')

plt.subplot(2,1,2)
plt.plot(x,Z)
plt.xlabel('x')
plt.ylabel('$\int_{a}^{b} f(x) dx$')
plt.show()
```



```
CODE 2(b):
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.pyplot as plt

def func(x):
    return ((np.cos(x))**3)

a=-3.14;b=3.14;n=5;sum=0;i=0
dx=(b-a)/n

x=[];y=[];Z=[]

while i<n:
    x1= a + i * dx
    x2= x1+dx
    trapz=(x1+x2)/2
    x.append(trapz)
    fn=func(trapz)
    y.append(fn)

    sum += y[i]*dx
    Z.append(sum)
    i=i+1

print("integrated value is =", round(sum,5))

plt.figure()
plt.subplot(2,1,1)
plt.plot(x,y)
plt.title('graph of f(x)=x and $\int f(x)$')
plt.xlabel('x')
plt.ylabel('f(x)')

plt.subplot(2,1,2)
plt.plot(x,Z)
plt.xlabel('x')
plt.ylabel('$\int_{a}^{b} f(x) dx$')
plt.show()
```



```
CODE 2(c):
import matplotlib.pyplot as plt
import numpy as np

def func(x):
    return np.sqrt(x)

a=3;b=9.14;n=15;sum=0;i=0
dx=(b-a)/n

x=[];y=[];Z=[]

while i<n:
    x1= a + i * dx
    x2= x1+dx
    trapz=(x1+x2)/2
    x.append(trapz)
    fn=func(trapz)
    y.append(fn)

    sum += y[i]*dx
    Z.append(sum)
    i=i+1

print("integrated value is =", round(sum,5))

#plt.figure()
plt.subplot(2,1,1)
plt.plot(x,y)
plt.title('graph of f(x)=x and $\int f(x)$')
plt.xlabel('x')
plt.ylabel('f(x)')

plt.subplot(2,1,2)
plt.plot(x,Z)
plt.xlabel('x')
plt.ylabel('$\int_{a}^{b} f(x) dx$')
plt.show()
```



OBSERVATION TABLE

MIDPOINT METHOD:

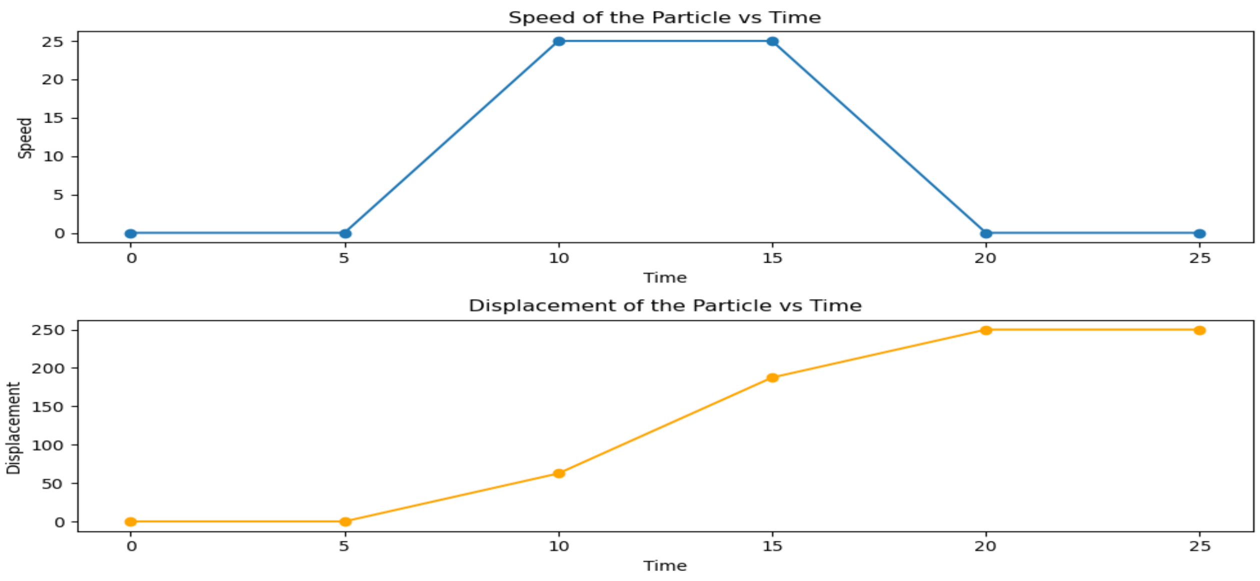
Sr.No	Number of steps					Analytical Answer
	5	10	15	20	50	
1a	29.28	29.82	29.92	29.955	29.993	30
1b	14.543	14.538	14.537	14.536	14.536	14.535
1c	0.004	0.003	0.003	0.003	0.003	0

TRAPEZOIDAL METHOD:

Sr.No	Number of steps					Analytical Answer
	5	10	15	20	50	
2a	29.28	29.82	29.92	29.955	29.993	30
2b	14.543	14.538	14.537	14.536	14.535	14.535
2c	0.004	0.003	0.003	0.003	0.003	0

HOME ACTIVITY: Acceleration and displacement.

```
import matplotlib.pyplot as plt
import numpy as np

# Define the acceleration values and time intervals
accelerations = [0, 5, 0, -5, 0]
time_intervals = np.arange(0, 26, 5)  # Time from 0 to 25 with 5 intervals each

# Initialize arrays to store speed and displacement values
speed = [0]  # Initial speed is 0
displacement = [0]  # Initial displacement is 0

# Numerical integration using Euler's method
for i in range(len(time_intervals) - 1):
    dt = time_intervals[i + 1] - time_intervals[i]
    new_speed = speed[-1] + accelerations[i] * dt
    new_displacement = displacement[-1] + speed[-1] * dt + 0.5 *
accelerations[i] * dt**2
    speed.append(new_speed)
    displacement.append(new_displacement)

# Plotting
plt.figure(figsize=(10, 6))

plt.subplot(2, 1, 1)
plt.plot(time_intervals, speed, marker='o')
plt.xlabel('Time')
plt.ylabel('Speed')
plt.title('Speed of the Particle vs Time')

plt.subplot(2, 1, 2)
plt.plot(time_intervals, displacement, marker='o', color='orange')
plt.xlabel('Time')
plt.ylabel('Displacement')
plt.title('Displacement of the Particle vs Time')

plt.tight_layout()
plt.show()
```



UID:2309111 ***NUMERICAL DIFFERENTIATION USING PYTHON*** *PRACTICAL:4*

AIM: To learn how to numerically differentiate the functions.

THEORY:

The derivative of a function of a single variable at a chosen input value, when it exists, is the slope of the tangent line to the graph of the function at that point.
The tangent line is the best linear approximation of the function near that input value. For this reason, the derivative is often described as the "instantaneous rate of change", the ratio of the instantaneous change in the dependent variable to that of the independent variable.
In order to numerically calculate the derivative of the given function we employ Taylor series expansion to derive finite-divided-difference approximation of derivatives.
The forward Taylor series expansion of function f(x) around point xi [predicting value of f(x) at xi+1].

fxi+1=fxi+f'xih+f"xi2h2+… ---------------(1) where h=xi+1-xi
Which can be solved for

f'xi=fxi+1-fxih-f"xi2h+O(h2) ---------------(2) where O(h2)represents the remaining terms of the series.

To get the expression for f"xi, we expand the function f(x) (for point xi+2) around point xi.

fxi+2=fxi+f'xi2h+f"xi2(2h)2+… ---------------(3)

Now equation-1 can be multiplied by 2 and subtracted from equation-3 to give

fxi+2-2fxi+1=-fxi+f"xih2+… ---------------(4)

Which can be solved for

f"xi=fxi+2-fxi+1+fxih2+Ω(h) ---------------(5)

This is inserted in equation-2 to get

f'xi=fxi+1-fxih-fxi+2-fxi+1+fxih2h+ζ(h2) --------(6)

Rearranging the terms in the above equation,

f'xi=-fxi+2+4fxi+1-3fxi2h+ψ(h2)----------------(7)

The first term in equation 7 can be used as first derivative of f(x) at point xi with error ψ(h2). For small h, ψ(h2) term can be ignored. Therefore we can write

f'xi=-fxi+2+4fxi+1-3fxi2h

PYTHON CODES:

EXAMPLE CODE:

```
import numpy as np
import matplotlib.pyplot as plt
def f(x):
      y=x**2
      return y

x=np.linspace(0,50, 500)
a=0;b=50;n= 500;i=0
h= (b-a)/n
derv=[]
while i<n-2:
      dervj=(-f(x[i+2])+4*f(x[i+1])-(3*f(x[i])))/(2*h)
      derv.append(dervj)
      i=i+1

print('differentiated value=',derv[-1])
plt.plot(x[0:498],derv)
plt.title('Result')
plt.xlabel('x')
plt.ylabel('derv')
plt.show()
```

CODE 1(a):

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
      y = np.sqrt(x)*np.exp(-x)
      return y

x = np.linspace(0.2,2, 500)
a = 0.2;b = 2;n = 500              #minimum range,maximum range,step size
h = (b-a)/n
i  = 0
derv=[ ]
while i<n-2:
      dervj = (-f(x[i+2])+4*f(x[i+1])-(3*f(x[i])))/(2*h)
      derv.append(dervj)
      i = i+1
print('differentiated value=',derv[-1])
plt.plot(x[0:498],derv)
plt.title('Result $\\sqrt{x}*e^{-x}$')
plt.xlabel('x')
plt.ylabel('derv')
plt.show()
```

CODE 1(b):

```
import numpy as np
import matplotlib.pyplot as plt
def f(x):
      y=(np.sin(x))**2/np.cos(x)
      return y
x=np.linspace(0,np.pi, 50)
a=0;b=np.pi;n= 50 #minimum range,maximum range,step size
h= (b-a)/n
i=0
derv=[]
while i<n-2:
      dervj=(-f(x[i+2])+4*f(x[i+1])-(3*f(x[i])))/(2*h)
      derv.append(dervj)
      i=i+1

print('differentiated value=',derv[-1])
plt.plot(x[0:48],derv)
plt.title('Result $\sin^2(x)\div(\cos(x))$')
plt.xlabel('x')
plt.ylabel('derv')
plt.show()
```

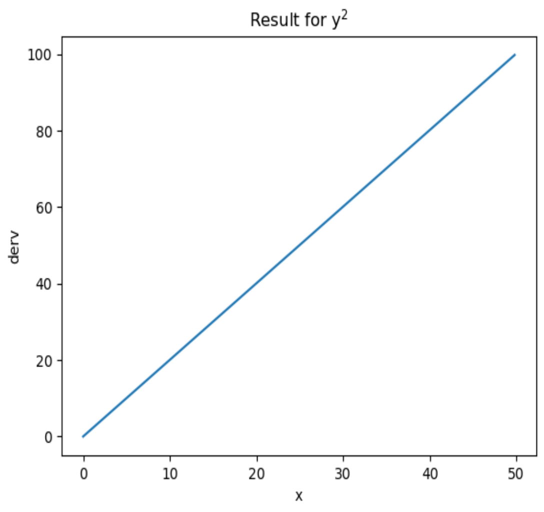
CODE 1(c):

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
      y=x**2/np.sqrt(x**3+1)
      return y

x=np.linspace(0.2,2, 5)
a = 0.2;b = 2;n = 5  #minimum range,maximum range,step size
h = (b-a)/n
i  = 0
derv=[ ]
while i<n-2:
      dervj=(-f(x[i+2])+4*f(x[i+1])-(3*f(x[i])))/(2*h)
      derv.append(dervj)
      i=i+1

print('differentiated value=',derv[-1])
plt.plot(x[0:3],derv)
plt.title('Result $x^2\div\sqrt{x^3+1}$')
plt.xlabel('x')
plt.ylabel('derv')
plt.show()
```

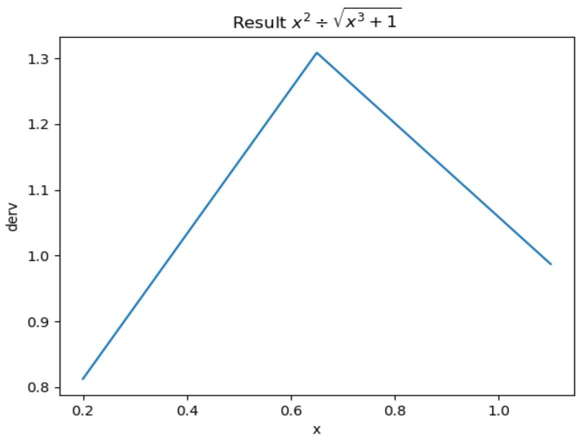
Home Activity Code:

```
import matplotlib.pyplot as plt
import numpy as np
def function(Q, r):
    V=-Q/r
    return V

Q=4;Q1 =2*Q;x_st=1;x_en=5;n = 100;i = 0
x = np.linspace(x_st, x_en, n)
x1=np.linspace(x_en, x_st, n)
h =(x_en- x_st)/n
y1=[]
while i < len(x) - 2:
    t=(4*function(Q, x[i+1])-function(Q, x[i+2])-3*function(Q, x[i])) /
(2*h)
    t1=(4*function(Q1, x1[i+1])-function(Q1, x1[i+2])-3*function (Q1,
x1[i])) / (2*h)
    y1.append(abs(t1-t))
    i+=1

min_electric_field = min(y1)
min_electric_field_x = x[y1.index(min_electric_field)]

print("The minimum value of the electric field is:", min_electric_field)
print('The Electric field is minimum at x =', round(min_electric_field_x,
4))
plt.plot(x[0:n -2 ], y1,'--', color='black')
plt.xlabel('Distance between two point charges', fontsize=10)
plt.ylabel('Magnitude of the electric field', fontsize=10)
plt.scatter(min_electric_field_x, min_electric_field, color='red')
plt.title('Analyzing the Electric Field between two points along the x-
axis', fontsize=15 )
plt.legend(fontsize=20)
plt.show()
```

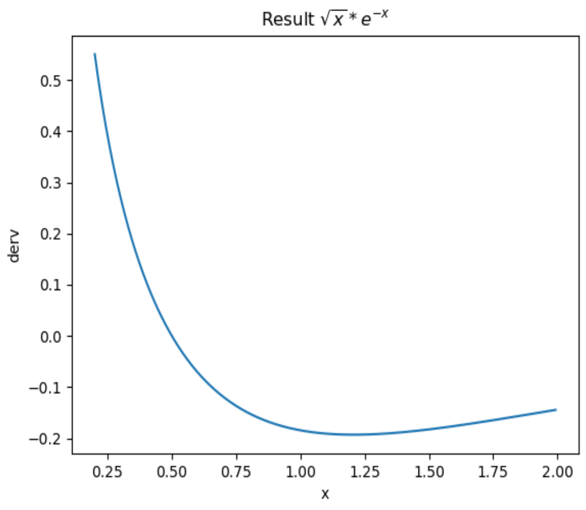
CONCLUSION:
Numerical differentiation is a valuable technique for estimating derivatives
when analytical solutions are unavailable or impractical. Python, with libraries like NumPy, provides a convenient environment for implementing and experimenting with various numerical differentiation methods. When using numerical differentiation, it's important to choose appropriate methods and step sizes to balance accuracy and stability for our specific problem.



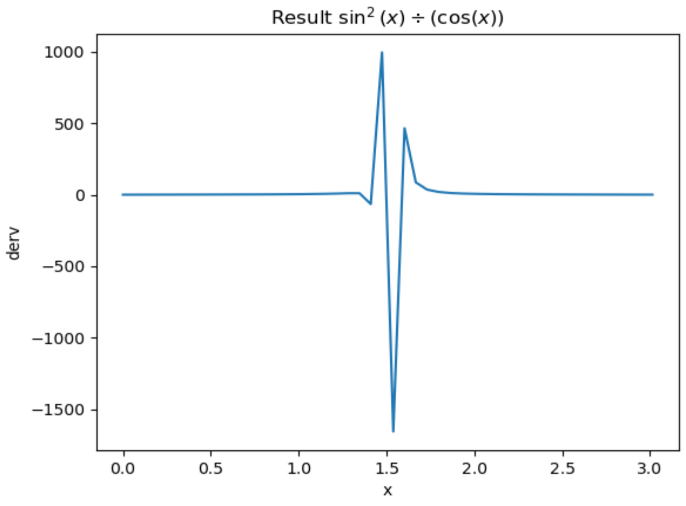
Result for y^2
derv
x



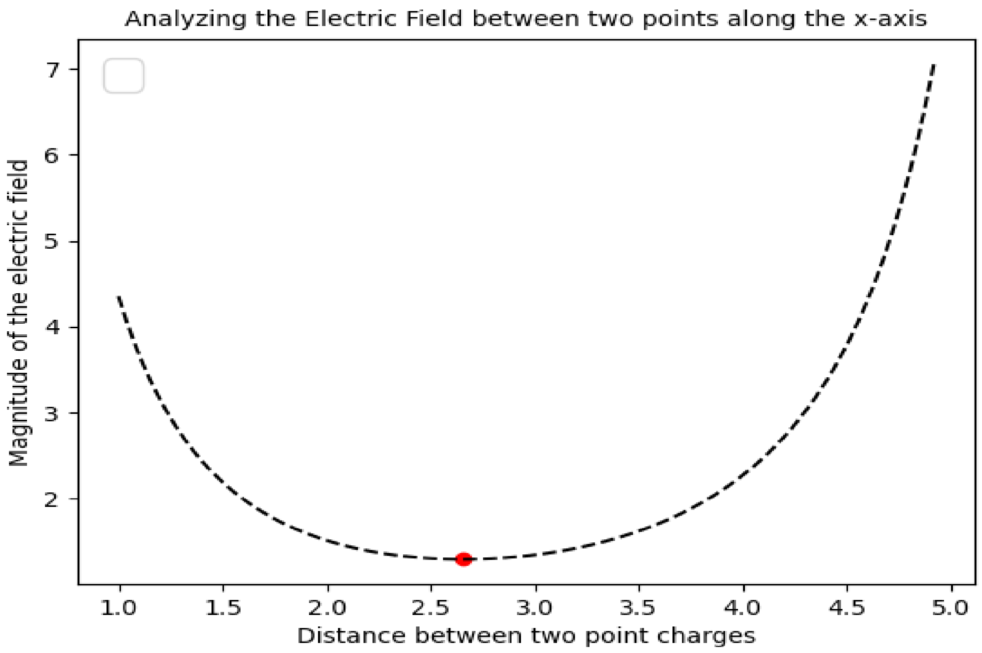
Result $x^2 \div \sqrt{x^3+1}$
derv
x



Result $\sqrt{x} * e^{-x}$
derv
x

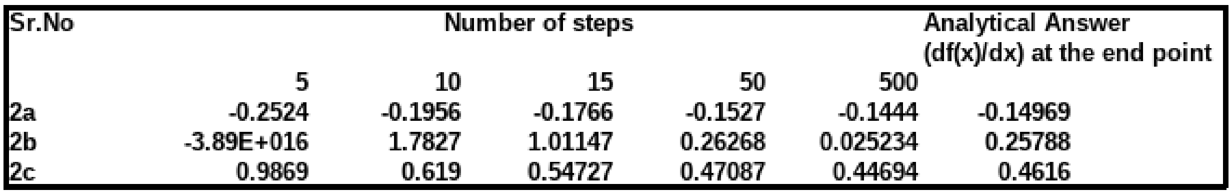


Result $\sin^2(x) \div (\cos(x))$
derv
x



Analyzing the Electric Field between two points along the x-axis
Magnitude of the electric field
Distance between two point charges

OBSERVATION TABLES:



Sr.No	Number of steps					Analytical Answer (df(x)/dx) at the end point
	5	10	15	50	500	
2a	-0.2524	-0.1956	-0.1766	-0.1527	-0.1444	-0.14969
2b	-3.89E+016	1.7827	1.01147	0.26268	0.025234	0.25788
2c	0.9869	0.619	0.54727	0.47087	0.44694	0.4616