Tribhuvan University
Institute of Engineering
PASHCHIMANCHAL CAMPUS
Pokhara, Nepal

NUMERICAL METHODS

LAB MANUAL

BCT, BEI, BCE, BEL

ASST. PROF. KHEMRAJ KOIRALA
HEAD OF DEPARTMENT

ASST. PROF. NABIN LAMICHHANE
DEPUTY HEAD OF DEPARTMENT

ER. NABARAJ SUBEDI
LECTURER

Department of Electronics and Computer Engineering
IOE, TU

त्रिभुवन विश्वविद्यालय
Tribhuvan University
इन्जिनियरिङ अध्ययन संस्थान
Institute of Engineering
पश्चिमाञ्चल क्याम्पस
PASHCHIMANCHAL CAMPUS
Accredited by University Grants Commission (UGC) Nepal. (2021 A.D.)

PO box- 46, Lamachaur, Pokhara
Tel: 977-061-440457, 440002, 440465,
Fax: 977-061-440158
info@wrc.edu.np, www.wrc.edu.np
info@ioepas.edu.np, www.ioepas.edu.np
पो.ब. नं- ४६, लामाचौर, पोखरा
९७७-०६१-४४०४५७, ४४०००२,४४०४६५,
फ्याक्स- ९७७-०६१-४४०१५८

Our Ref:

Date: 2025/01/20

Letter of Appreciation

I am very much delighted with the skills and efforts of our faculty members **Asst. Prof. Nabin Lamichhane** and **Er. Nabaraj Subedi**, in preparing the Lab Manual for Numerical Methods.

The Numerical Methods is one of the compulsory subjects taught at different semester for engineering students to provide them with a strong foundation in numerical analysis and practical applications. The lab manual prepared by the authors has simple explanations of relevant concepts, accompanied by clear and practical examples for better understanding. It also includes implementation of different numerical algorithm along with relevant working principle and pseudocode.

The authors have expertly structured the manual with detailed practical implementations in Python using tools like NumPy, Pandas, Matplotlib, Seaborn. This lab manual serves as an excellent resource for both students and instructors, making it easier to teach and learn effectively.

In my experience, students often face challenges in understanding and implementing data science concepts. This lab manual serves as an excellent resource for both students and instructors, making it easier to teach and learn effectively.

I hope the authors will continue their efforts in academic activities for the best in the future. I wish them the best of luck in their future endeavours.

Best Wishes,

Asst. Prof. Khemraj Koirala
Head of Department
Department of Electronics and Computer Engineering
Pashchimanchal Campus
Institute of Engineering, Tribhuvan University

ACKNOWLEDGEMENT

We want to extend our heartfelt gratitude to all those who contributed to completing this **LAB MANUAL FOR NUMERICAL METHODS** of **Pashchimanchal Campus**, from the **Department of Electronics and Computer Engineering.**

First and foremost, we sincerely thank the esteemed faculty members of our department for their unwavering support, guidance, and valuable insights throughout the preparation of this manual. Their continuous encouragement has greatly contributed to the overall quality of this work.

We also express our appreciation to the administrative and technical staff members, whose cooperation and assistance have been instrumental in facilitating the preparation of this manual.

A special mention goes to **Mr. Rabindra Baral**, the editor, whose meticulous work and attention to detail have significantly improved the content and presentation of the manual.

This manual would not have been possible without the collective efforts of all the individuals mentioned above. We are truly grateful for their contributions.

Thank you all for your support.

The Authors
Pashchimanchal Campus
Department of Electronics and Computer Engineering

Practical (45 hours)(ENSH 202)
Programming language to be used: Python
Results to be visualized graphically wherever possible
Practical report contents: Working principle, Pseudocode, Source code, Test Cases

1. Basics of programming in Python:
    - Basic input/output
    - Basic data types and data structures
    - Control flow
    - Functions and modules
    - Basic numerical and scientific computation
    - Graphical visualization
2. Solution of Non-linear equations:
    - Bisection method
    - Secant method
    - Newton-Raphson
    - System of non-linear equations using Newton-Raphson method
3. System of linear algebraic equations:
    - Gauss Jordan Method
    - Gauss elimination method with partial pivoting
    - Gauss-Seidal method
    - Power method
4. Interpolation
    - Newton's forward difference interpolation
    - Lagrange interpolation
    - Least square method for linear, exponential and polynomial curve fitting
    - Cubic spline interpolation
5. Numerical Integration
    - Trapezoidal rule
    - Simpson's 1/3 rule or Simpson's 3/8 rule
    - Boole's Rule or Weddle's Rule
    - Gauss-Legendre integration
6. Solution of Ordinary Differential Equations:
    - Runge-Kutta fourth order method for first order ODE
    - Runge-Kutta fourth order method for system of ODEs / 2nd order ODE
    - Solution of two-point boundary value problem using Shooting method
    - Solution of two-point boundary value problem using finite difference method
7. Solution of partial differential equations using finite difference approach:
    - Laplace equation using Gauss-Seidal iteration
    - Poisson's equation using Gauss-Seidal iteration
    - One-dimensional heat equation using Bendre-Schmidt method
    - One-dimensional heat equation using Crank-Nicholson method

LAB1:

Basics of programming in Python: Basic input/output Basic data types and data structures Control flow Functions and modules Basic numerical and scientific computation Graphical visualization

LAB 1:Basics of python programming

1.1 Basic Input Output

```
b = 2

print(b)
```

```
2
```

```
string1 = "This is python programming"

print(string1[1:3])
```

```
hi
```

```
n=input("enter the number")
print(n)
```

```
enter the number12
12
```

Tuples

```
tup=("hello",1,2,3)

tup
```

```
('hello', 1, 2, 3)
```

List

```
l1=[1,2,"hello"]

l1
```

```
[1, 2, 'hello']
```

Dictionary

```
dict1={"a":1,2:"hi"}

dict1
```

```
{'a': 1, 2: 'hi'}
```

Set

```
s1=set({1,2,1,"hello"})

s1
```

```
{1, 2, 'hello'}
```

Control flow

IF else

```
a=10
b=20
if(a>b):
  print("a is greater")
else:
  print("b is greater")
```

```
b is greater
```

```
cmd="start"
match cmd:
  case "start":
    print("start the game")
  case "stop":
    print("stop the game")
```

```
start the game
```

For loop

```
for i in range(10):
  print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

Functions and modules

```
def hello():
  print("hello")

hello()
```

```
hello
```

```
import numpy as np

a= np.array([1,2,3,4])

a
```

```
array([1, 2, 3, 4])
```

```
from cmath import sin
a=sin(10)
print(a)
```

```
(-0.5440211108893698-0j)
```

```
import math
a=math.sqrt(18)
print(a)
```

```
4.242640687119285
```

Error handling

```
a=-2
try:
  if a<0:
     raise ValueError("Cannot find the squareroot of negative number")

  b=math.sqrt(a)
  print(b)
except ValueError as e:
  print(e)
```

```
Cannot find the squareroot of negative number
```

```
try:
    c = 10 / 0  # This will raise a ZeroDivisionError
    raise ValueError("Cannot divide by zero")  # This won't execute
because of the above error
except ZeroDivisionError as e:
    print("Caught a ZeroDivisionError:", e)
except ValueError as e:
    print("Caught a ValueError:", e)
finally:
    print("The error is of another source or handled completely.")
```

```
Caught a ZeroDivisionError: division by zero
The error is of another source or handled completely.
```

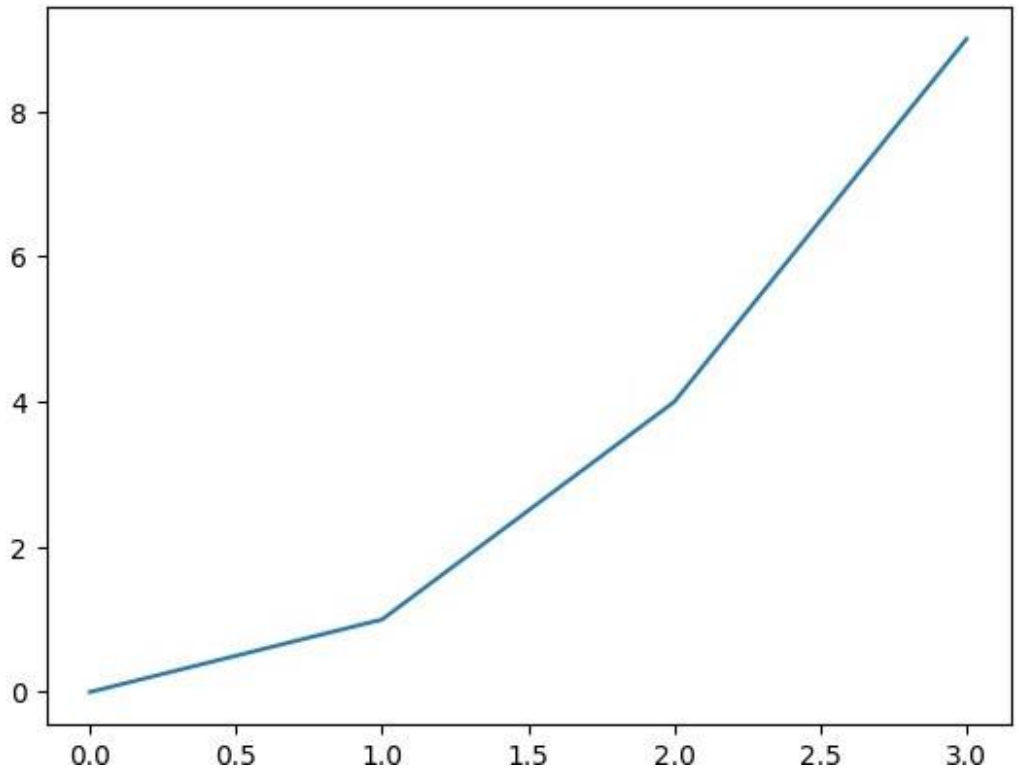
Numerical and scientific computations

```
a=np.array([[1,2,3],[5,6,7]])

a

array([[1, 2, 3],
       [5, 6, 7]])

b=np.array([[23,7,3],[8,6,9]])

b

array([[23,  7,  3],
       [ 8,  6,  9]])

c=np.array([[4,5],[6,7],[9,0]])

c

array([[4, 5],
       [6, 7],
       [9, 0]])

d=np.dot(a,c)

d

array([[ 43,  19],
       [119,  67]])

e=a@c

e

array([[ 43,  19],
       [119,  67]])

f=a+b

f

array([[24,  9,  6],
       [13, 12, 16]])

g=a*2

g
```

```
array([[ 2,  4,  6],
       [10, 12, 14]])

h=a+c.T

h

array([[ 5,  8, 12],
       [10, 13,  7]])

import numpy as np

a = np.array([1, 2, 3])
b = 5  # Scalar

result = a + b
print(result)

[6 7 8]

a = np.array([[1, 2, 3],
              [4, 5, 6]])
b = np.array([10, 20, 30])  # Shape (3,)

result = a + b
print(result)

[[11 22 33]
 [14 25 36]]

a = np.array([[1, 2, 3],
              [4, 5, 6]])
b = np.array([[10],
              [20]])  # Shape (2, 1)

result = a + b
print(result)

[[11 12 13]
 [24 25 26]]
```

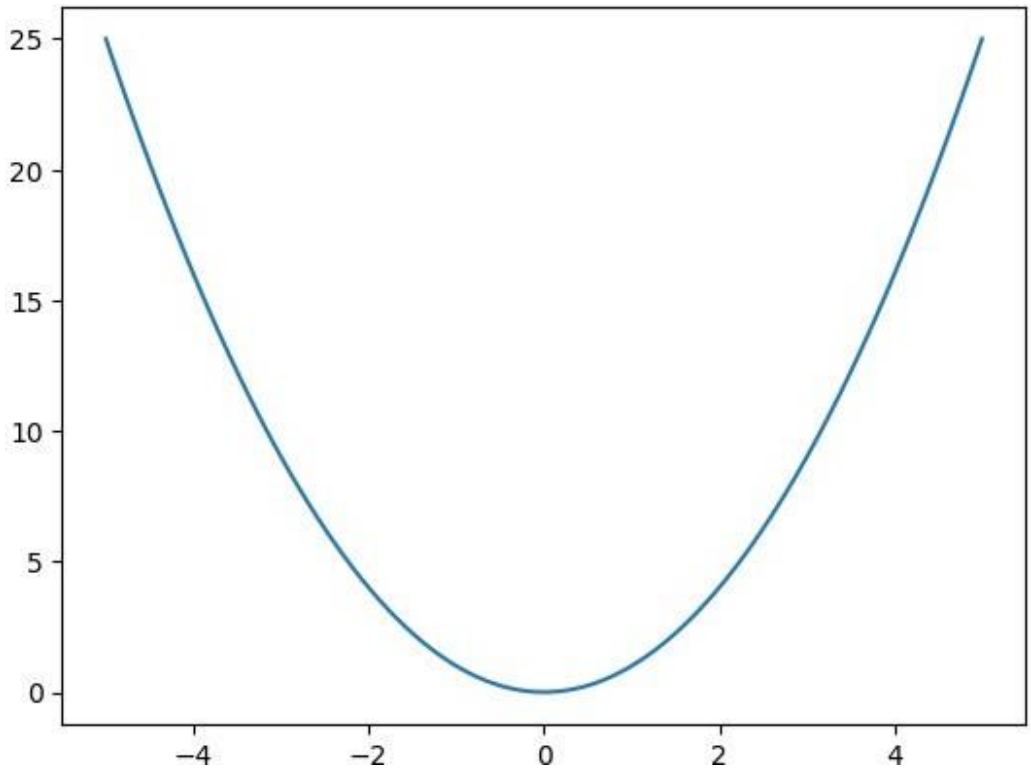
Visualization and Plotting

```
%matplotlib inline
import matplotlib.pyplot as plt

x = [0, 1, 2, 3]
y = [0, 1, 4, 9]

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



Make a plot of the function f (x) = x2 for −5 ≤ x ≤ 5

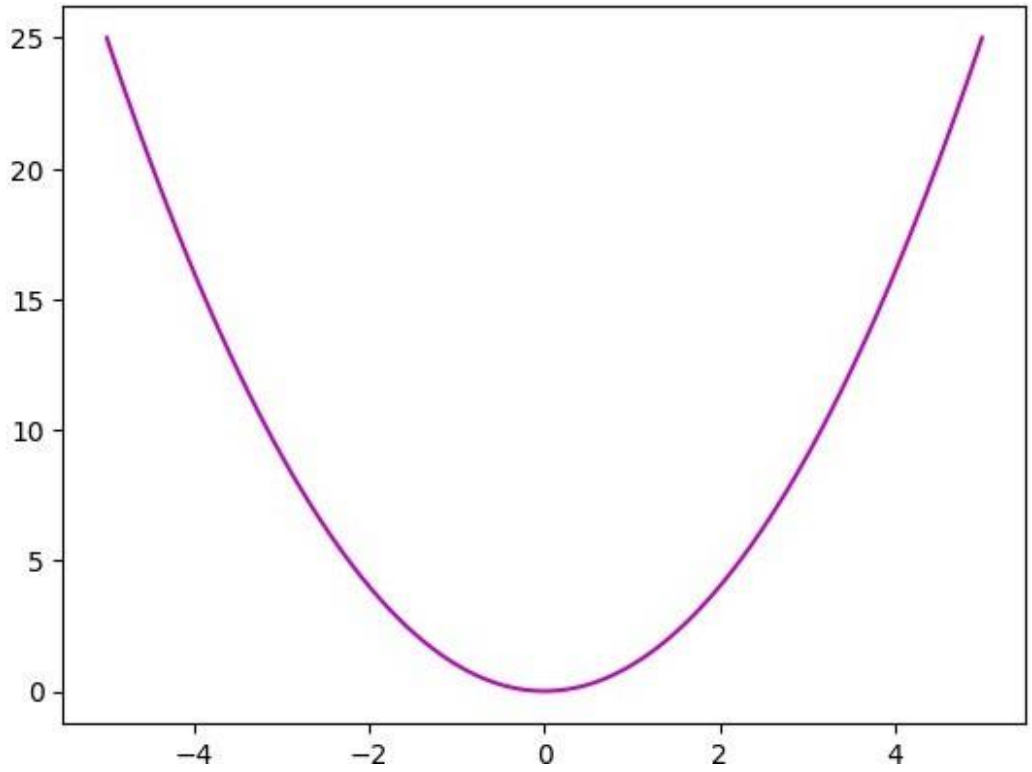
```
 %matplotlib inline
x = np.linspace(-5,5, 100)
plt.plot(x, x**2)
plt.show()
```



you can specify the color and format using the below table

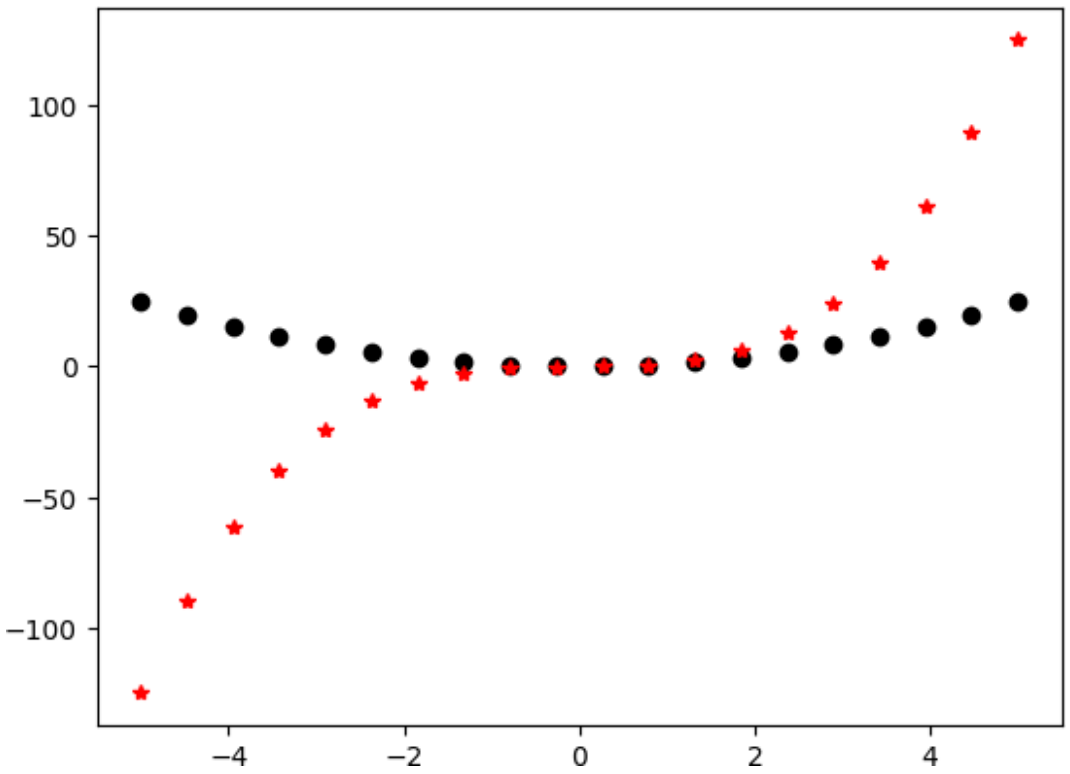
Symbol	Description	Symbol	Description
b	blue	T	T
g	green	s	square
r	red	d	diamond
c	cyan	v	triangle (down)
m	magenta	^	triangle (up)
y	yellow	<	triangle (left)
k	black	>	triangle (right)
w	white	p	pentagram
.	point	h	hexagram
o	circle	-	solid
x	x-mark	:	dotted
+	plus	-.	dashed–dotted
*	star	-	dashed

```
%matplotlib inline
x = np.linspace(-5,5, 100)
plt.plot(x, x**2,"m-")
plt.show()
```

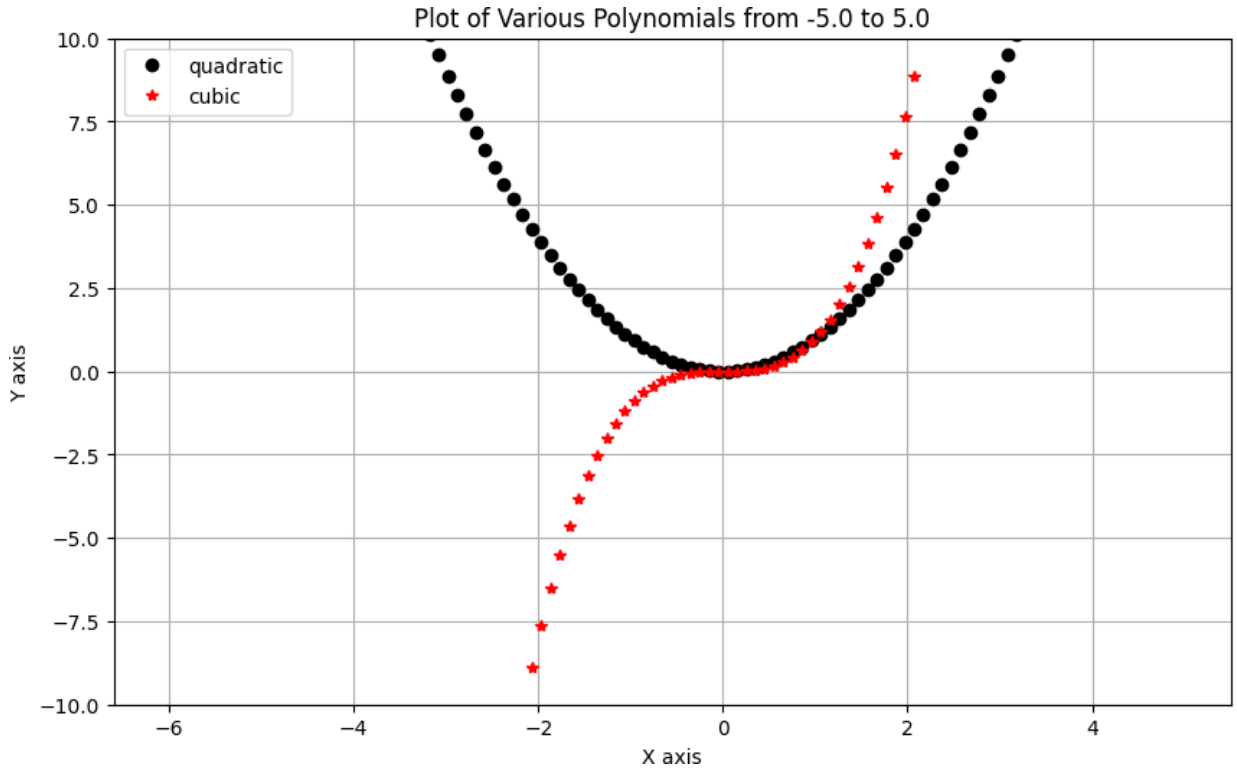


Make a plot of the function f (x) = x2 and g(x) = x3 for −5 ≤ x ≤ 5. Use different colors and markers for each function.

```
x = np.linspace(-5,5,20)
plt.plot(x, x**2, "ko")
plt.plot(x, x**3, "r*")
plt.show()
```



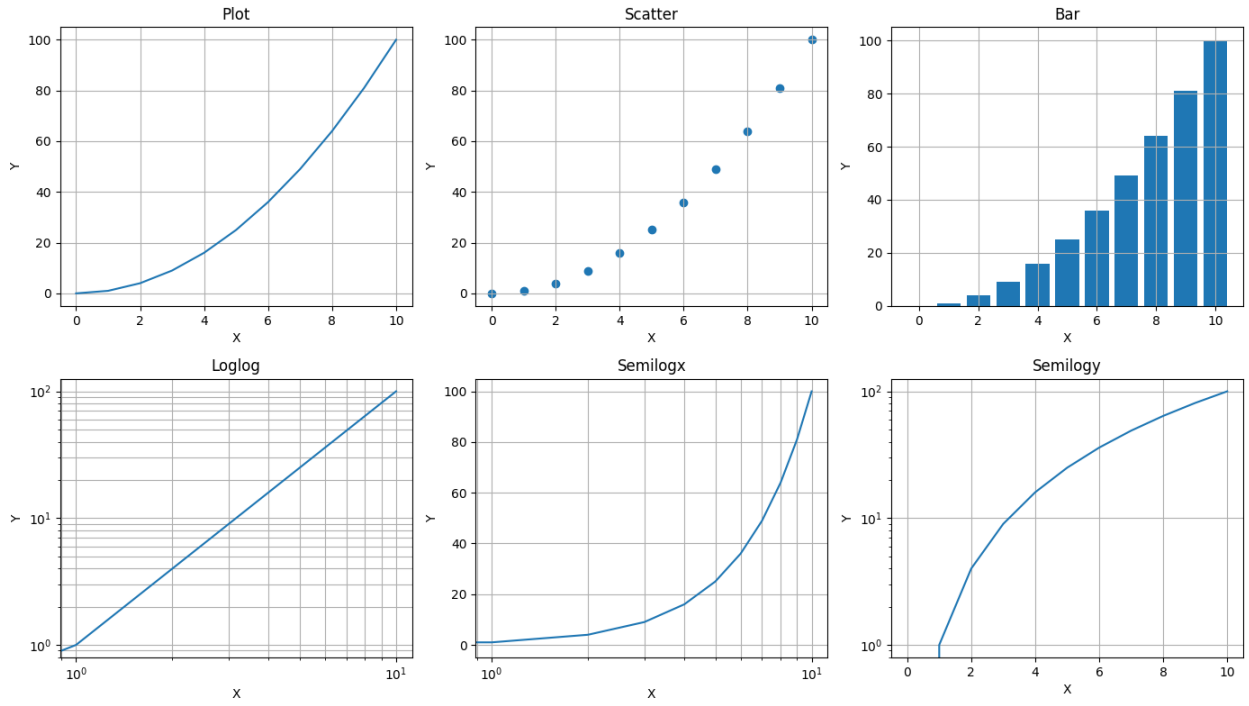

```
plt.figure(figsize = (10,6))
x = np.linspace(-5,5,100)
plt.plot(x, x**2, "ko", label = "quadratic")
plt.plot(x, x**3, "r*", label = "cubic")
plt.title(f"Plot of Various Polynomials from {x[0]} to {x[-1]}")
plt.xlabel("X axis")
plt.ylabel("Y axis")
plt.legend(loc = 2)
plt.xlim(-6.6)
plt.ylim(-10,10)
plt.grid()
plt.show()
```



Given the lists x = np.arange(11) and y = x2, create a 2 × 3 subplot where each subplot plots x versus y using plot, scatter, bar, loglog, semilogx, and semilogy. Title and label each plot appropriately. Use a grid, but a legend is not necessary here.

```
x = np.arange(11)
y = x**2
plt.figure(figsize = (14, 8))
plt.subplot(2, 3, 1)
plt.plot(x,y)
plt.title("Plot")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid()
plt.subplot(2, 3, 2)
plt.scatter(x,y)
plt.title("Scatter")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid()
plt.subplot(2, 3, 3)
plt.bar(x,y)
plt.title("Bar")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid()
plt.subplot(2, 3, 4)
```

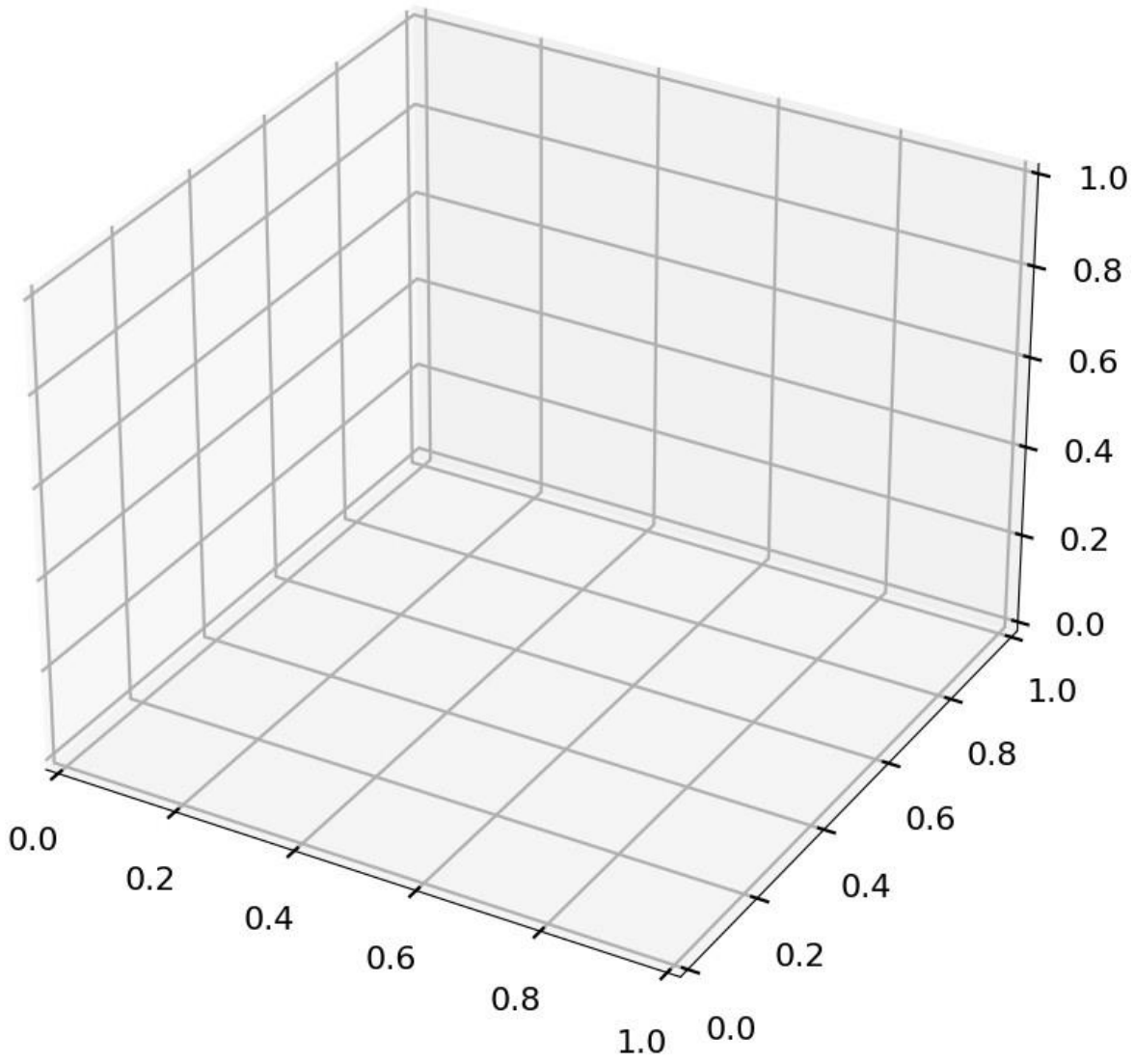
```
plt.loglog(x,y)
plt.title("Loglog")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(which="both")
plt.subplot(2, 3, 5)
plt.semilogx(x,y)
plt.title("Semilogx")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(which="both")
plt.subplot(2, 3, 6)
plt.semilogy(x,y)
plt.title("Semilogy")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid()
plt.tight_layout()
plt.show()
```



3D plotting

```
import numpy as np
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
plt.style.use("seaborn-v0_8-poster")
fig = plt.figure(figsize = (10,10))
```

```
ax = plt.axes(projection="3d")
plt.show()
```



Consider the parameterized dataset: t is a vector from 0 to 10π with a step π/50, x = sin(t), and y = cos(t). Make a 3D plot of the (x, y,t) dataset using plot3. Turn on the grid, make the axis equal, and add axis labels and a title. Activate the i

```
%matplotlib inline
fig = plt.figure(figsize = (8,8))
ax = plt.axes(projection="3d")
ax.grid()
t = np.arange(0, 10*np.pi, np.pi/50)
```

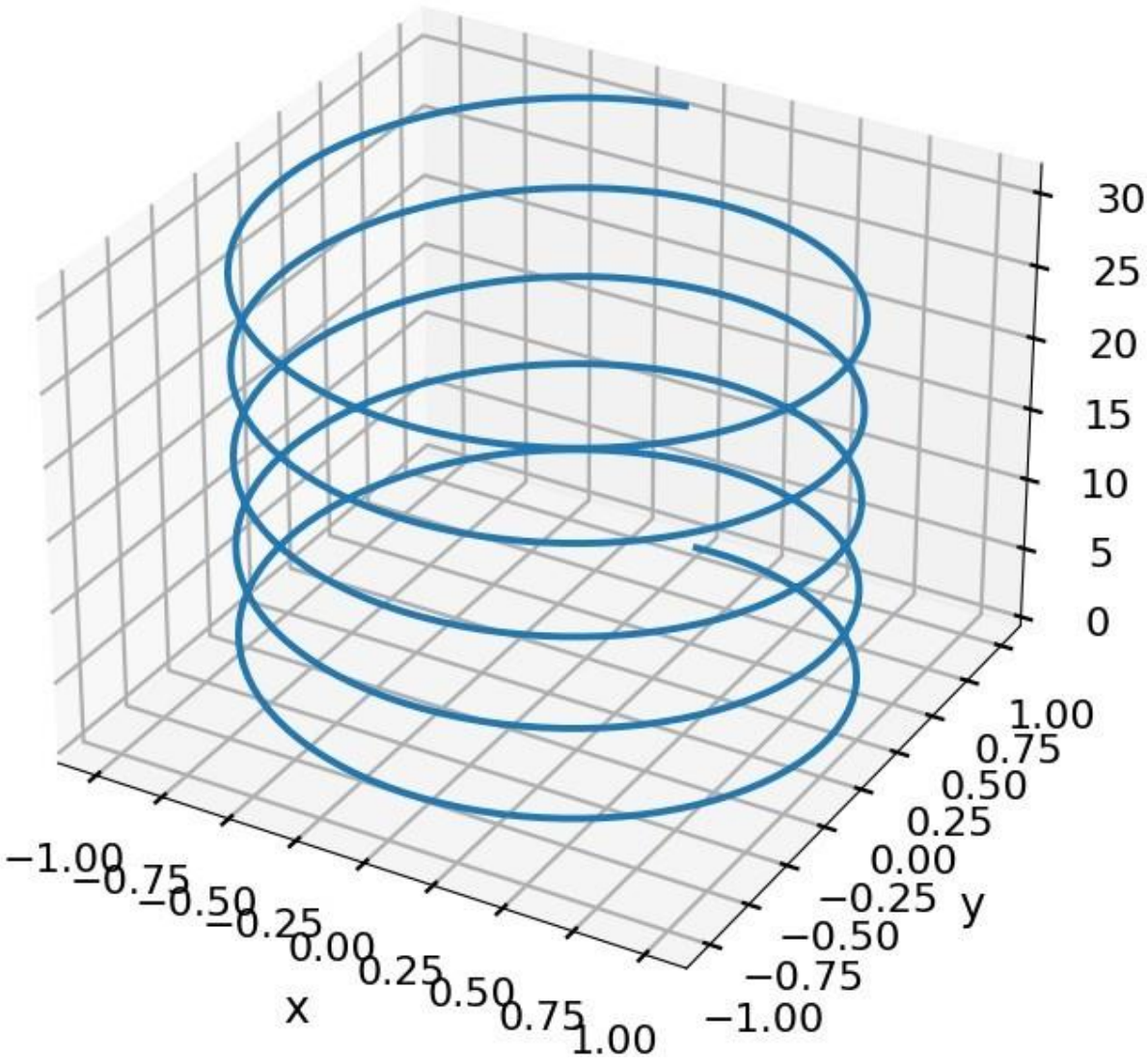
```
x = np.sin(t)
y = np.cos(t)

ax.plot3D(x, y, t)
ax.set_title("3D Parametric Plot")
# Set axes label
ax.set_xlabel("x", labelpad=20)
ax.set_ylabel("y", labelpad=20)
ax.set_zlabel("t", labelpad=20)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

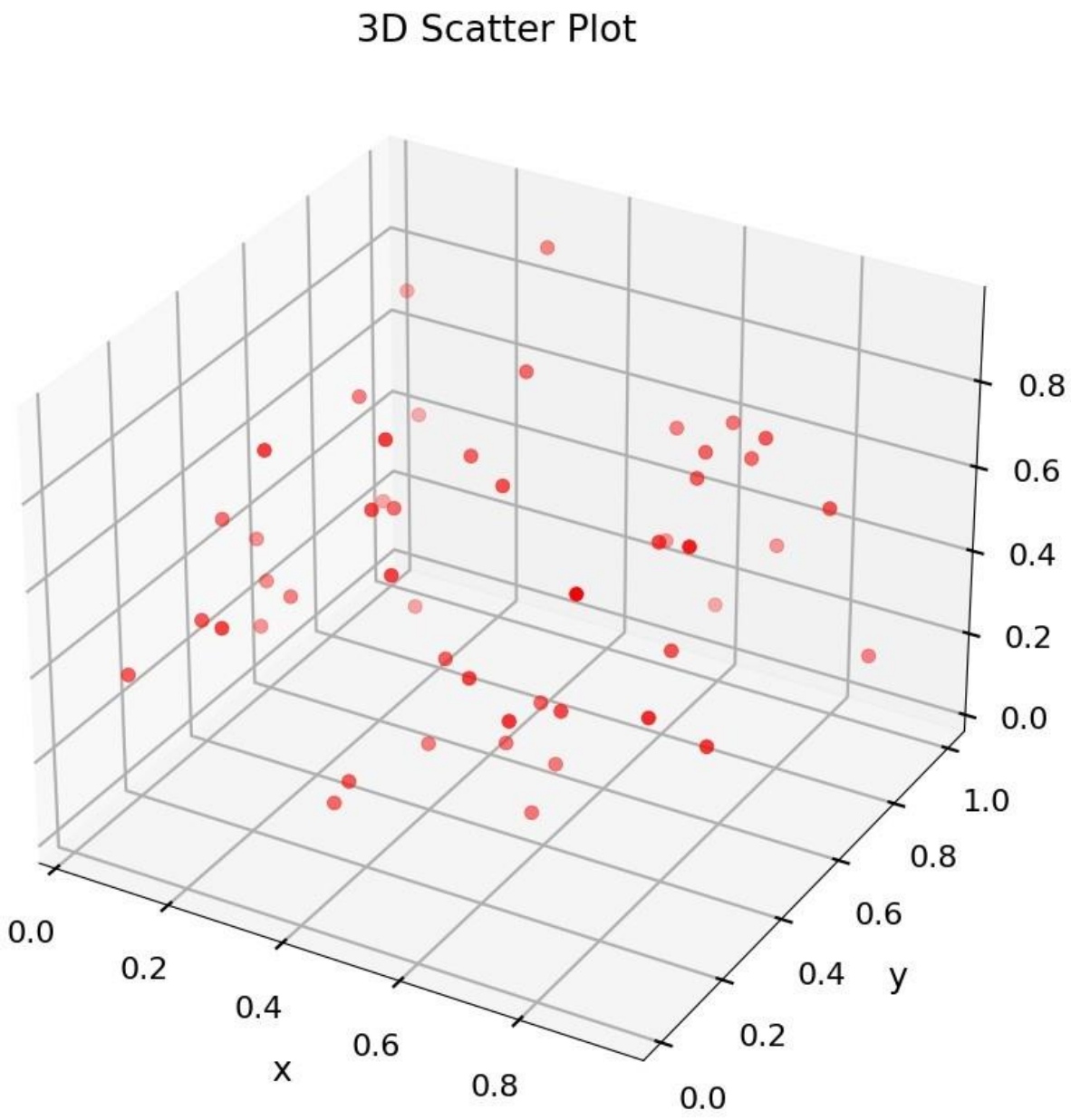
```
<IPython.core.display.HTML object>
```



Make a 3D scatter plot with randomly generated 50 data points for x, y, and z. Set the point color as red and size of the point as 50.

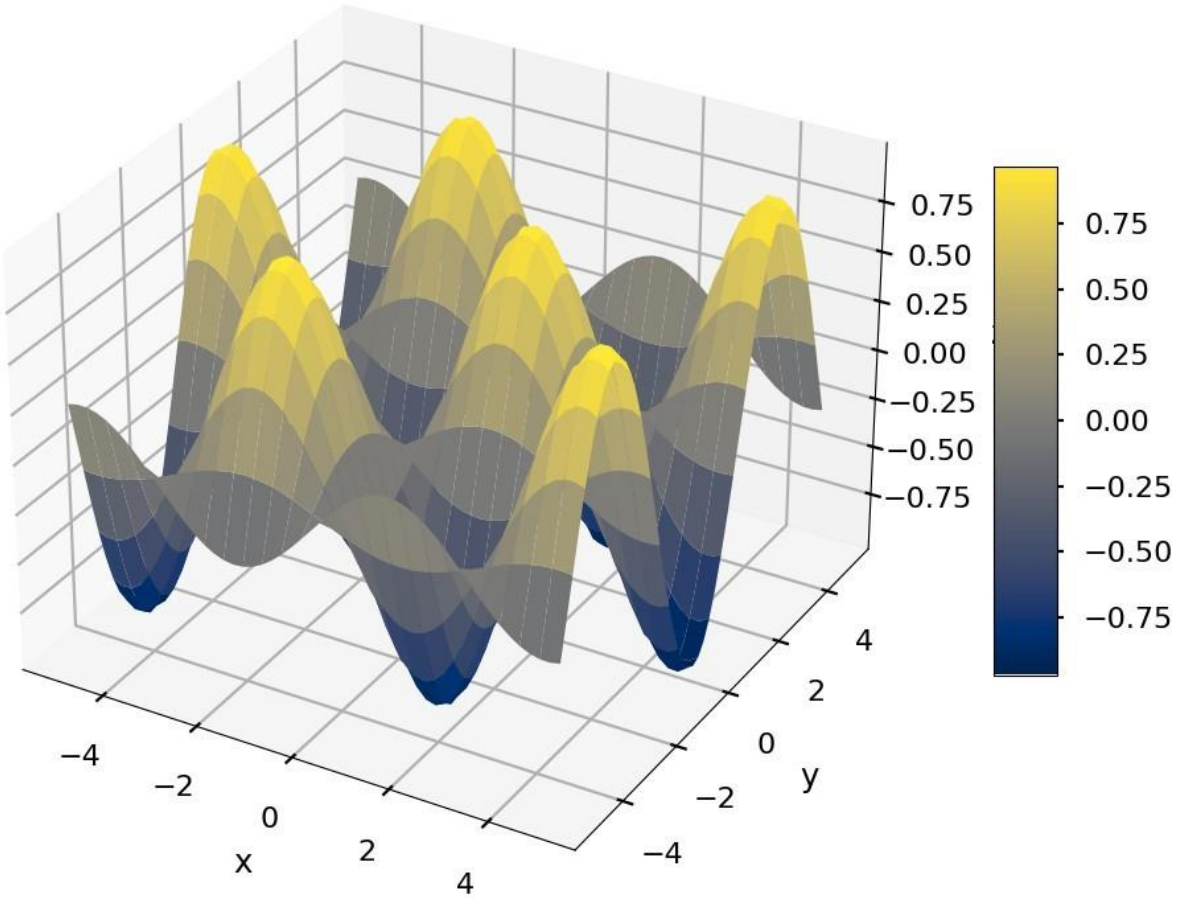
```
%matplotlib inline
x = np.random.random(50)
y = np.random.random(50)
z = np.random.random(50)
fig = plt.figure(figsize = (10,10))
ax = plt.axes(projection="3d")
ax.grid()
ax.scatter(x, y, z, c = "r", s = 50)
ax.set_title("3D Scatter Plot")
```

```
# Set axes label
ax.set_xlabel("x", labelpad=20)
ax.set_ylabel("y", labelpad=20)
ax.set_zlabel("z", labelpad=20)
plt.show()
```



Make a plot of the surface f (x, y) = sin(x) · cos(y) for −5 ≤ x ≤ 5,−5 ≤ y ≤ 5 using the plot_surface function. Take care to use a sufficiently fine discretization in x and y so that the plot looks smooth.

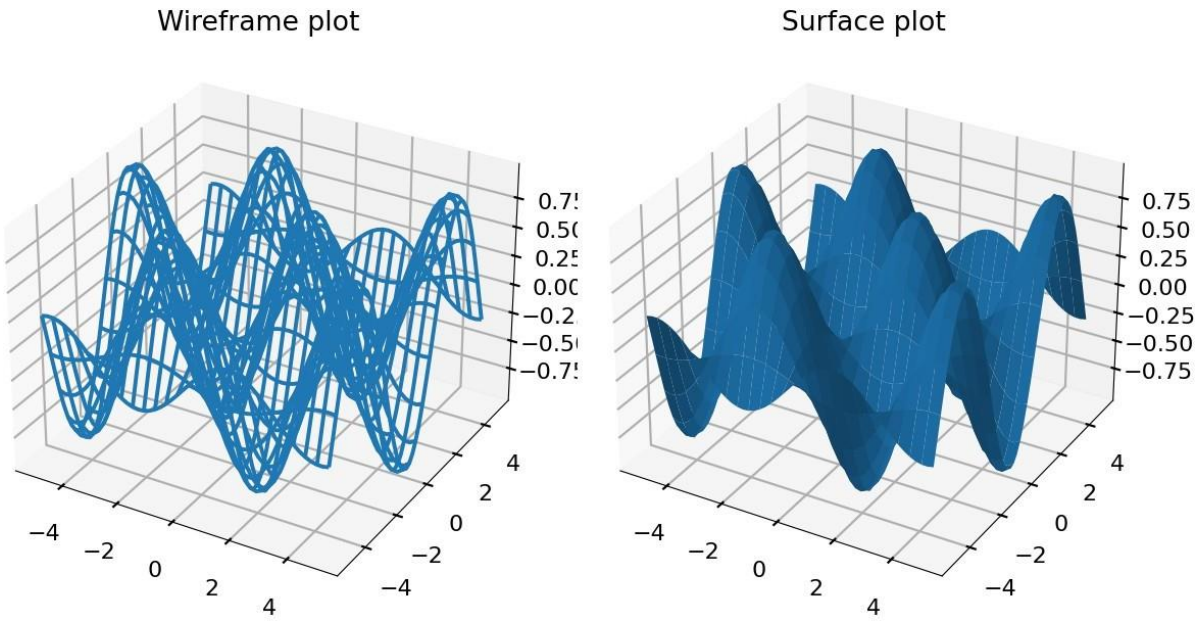
```
fig = plt.figure(figsize = (12,10))
ax = plt.axes(projection="3d")
x = np.arange(-5, 5.1, 0.2)
y = np.arange(-5, 5.1, 0.2)
X, Y = np.meshgrid(x, y)
Z = np.sin(X)*np.cos(Y)
surf = ax.plot_surface(X, Y, Z, cmap = plt.cm.cividis)
# Set axes label
ax.set_xlabel("x", labelpad=20)
ax.set_ylabel("y", labelpad=20)
ax.set_zlabel("z", labelpad=20)
fig.colorbar(surf, shrink=0.5, aspect=8)
plt.show()
```



Make a 1 × 2 subplot to plot the above X, Y, Z data in a wireframe plot and a surface plot

```
fig = plt.figure(figsize=(12,6))
ax = fig.add_subplot(1, 2, 1, projection="3d")
ax.plot_wireframe(X,Y,Z)
```

```
ax.set_title("Wireframe plot")
ax = fig.add_subplot(1, 2, 2, projection="3d")
ax.plot_surface(X,Y,Z)
ax.set_title("Surface plot")
plt.tight_layout()
plt.show()
```



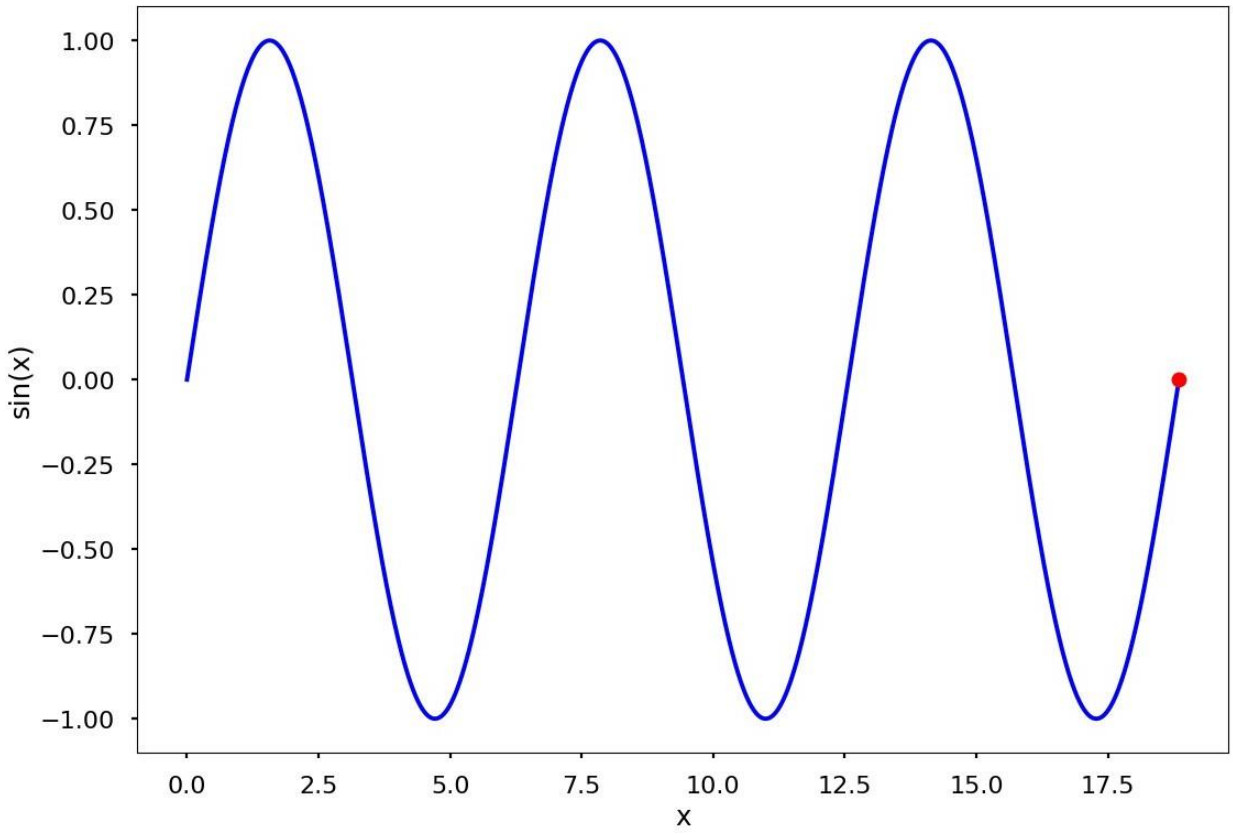
Annimations of the graphs

Create an animation of a red circle following a blue sine wave

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as manimation
n = 1000
x = np.linspace(0, 6*np.pi, n)
y = np.sin(x)
# Define the meta data for the movie
FFMpegWriter = manimation.writers["ffmpeg"]
metadata = dict(title="Movie Test", artist="Matplotlib",
comment="a red circle following a blue sine wave")
writer = FFMpegWriter(fps=15, metadata=metadata)
# Initialize the movie
fig = plt.figure()
# plot the sine wave line
sine_line, = plt.plot(x, y, "b")
red_circle, = plt.plot([], [], "ro", markersize = 10)
plt.xlabel("x")
plt.ylabel("sin(x)")
# Update the frames for the movie
```

```
with writer.saving(fig, "writer_test.mp4", 100):
  for i in range(n):
    x0 = x[i]
    y0 = y[i]
    red_circle.set_data(x0, y0)
    writer.grab_frame()
```

```
<ipython-input-91-1c65db709537>:24: MatplotlibDeprecationWarning:
Setting data with a non sequence type is deprecated since 3.7 and will
be remove two minor releases later
  red_circle.set_data(x0, y0)
```



LAB 2

1. Bisection method

Working principle The Bisection Method is a root-finding algorithm that repeatedly bisects an interval and then selects the subinterval in which a root must lie. The method requires two initial points, a and b, such that the function values at these points have opposite signs. The algorithm iteratively narrows down the search interval by calculating the midpoint and evaluating the function at this point until the root is approximated to a desired precision.

Pseudo code:

1. Given initial values a, b, and tolerance ε.
2. Check if f(a) and f(b) have opposite signs.
3. Repeat until the interval length is less than ε:

   a. Find midpoint m = (a + b) / 2.

   b. If f(m) is close to 0, then m is the root.

   c. Otherwise, if f(a) and f(m) have opposite signs, set b = m.

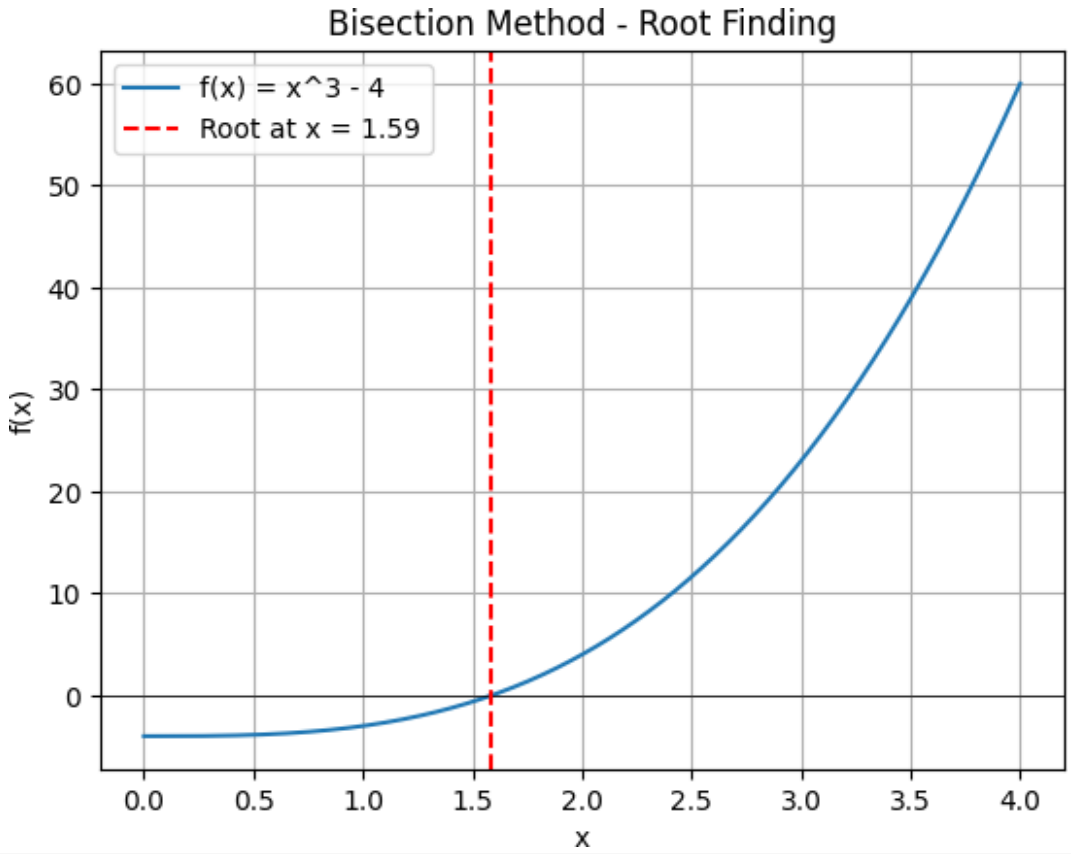
   If f(m) and f(b) have opposite signs, set a = m.
4. Return the root approximation m.

Source code:

```
import numpy as np
import matplotlib.pyplot as plt
from cmath import sin
%matplotlib inline

def my_bisection(f, a, b, tol):
    """
    Approximates a root, R, of the function f bounded by a and b to
within tolerance tol.

    Arguments:
    f -- Function whose root we are trying to find
    a -- Left boundary of the interval
    b -- Right boundary of the interval
    tol -- Tolerance value for stopping the iteration

    Returns:
    m -- Approximate root
    """
    # Check if a and b bound a root
    if np.sign(f(a)) == np.sign(f(b)):
        raise Exception("The scalars a and b do not bound a root")

    # Get midpoint
    m = (a + b) / 2

    # Stopping condition, report m as root if f(m) is within tolerance
    if np.abs(f(m)) < tol:
        return m
    elif np.sign(f(a)) == np.sign(f(m)):
        # Case where m is an improvement on a. Make recursive call
with a = m
        return my_bisection(f, m, b, tol)
    elif np.sign(f(b)) == np.sign(f(m)):
        # Case where m is an improvement on b. Make recursive call
with b = m
        return my_bisection(f, a, m, tol)


def f(x):
    return x**3 - 4

# Bisection method
root = my_bisection(f, 1, 3, 0.01)
print("Root:", root)

# Plotting the function and the root
x_vals = np.linspace(0, 4, 400)
y_vals = f(x_vals)
```

```
plt.plot(x_vals, y_vals, label="f(x) = x^3 - 4")
plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(root, color='r', linestyle='--', label=f'Root at x =
{root:.2f}')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Bisection Method - Root Finding')
plt.legend()
plt.grid(True)
plt.show()
```

```
Root: 1.587890625
```




```
def func(x):
    return x**3 - 5 * x - 9

def bisection_method(func, a, b, tolerance):
    iteration = 1
    while True:
        midpoint = (a + b) / 2
        f_mid = func(midpoint)
        print(f"Iteration-{iteration}, x = {midpoint:.6f}, f(x) = {f_mid:.6f}")

        # Update the bracket
        if func(a) * f_mid < 0:
            b = midpoint
```

```
        else:
            a = midpoint

        # Check if the root is found within tolerance
        if abs(f_mid) <= tolerance:
            print(f"Required Root is: {midpoint:.8f}")
            return midpoint

        iteration += 1

def main():
    print("BISECTION METHOD IMPLEMENTATION")
    print()

    # Default values
    function_str = "x**3 - 5 * x - 9"
    a, b, tolerance = 2, 3, 0.0001

    print(f"f(x) = {function_str}")
    print(f"a = {a}, b = {b}, tolerance = {tolerance}")
    print()

    # Check if the guesses bracket the root
    if func(a) * func(b) >= 0:
        print("Error: The guesses do not bracket a root. Try again with
different values.")
        return

    # Perform the bisection method
    bisection_method(func, a, b, tolerance)

if __name__ == "__main__":
    main()
```

Output:

BISECTION METHOD IMPLEMENTATION

f(x) = x**3 - 5 * x - 9
a = 2, b = 3, tolerance = 0.0001

Iteration-1, x = 2.500000, f(x) = -5.875000
Iteration-2, x = 2.750000, f(x) = -1.953125
Iteration-3, x = 2.875000, f(x) = 0.388672
Iteration-4, x = 2.812500, f(x) = -0.815186
Iteration-5, x = 2.843750, f(x) = -0.221588
Iteration-6, x = 2.859375, f(x) = 0.081448
Iteration-7, x = 2.851562, f(x) = -0.070592
Iteration-8, x = 2.855469, f(x) = 0.005297

Iteration-9, x = 2.853516, f(x) = -0.032680
Iteration-10, x = 2.854492, f(x) = -0.013700
Iteration-11, x = 2.854980, f(x) = -0.004204
Iteration-12, x = 2.855225, f(x) = 0.000546
Iteration-13, x = 2.855103, f(x) = -0.001829
Iteration-14, x = 2.855164, f(x) = -0.000641
Iteration-15, x = 2.855194, f(x) = -0.000048
Required Root is: 2.85519409

Test cases

Test for f(x) = x^2 - 4 in the interval [1, 3] with tolerance 0.01.

Test for f(x) = cos(x) - x in the interval [0, 1] with tolerance 0.001

2.Secant method

Working Principle: The Secant Method is an iterative technique for finding the root of a function. It uses two initial guesses and approximates the root by the intersection of secant lines drawn through successive points on the graph of the function. The method avoids the need for the derivative of the function but still converges to the root under the right conditions.

Pseudocode:

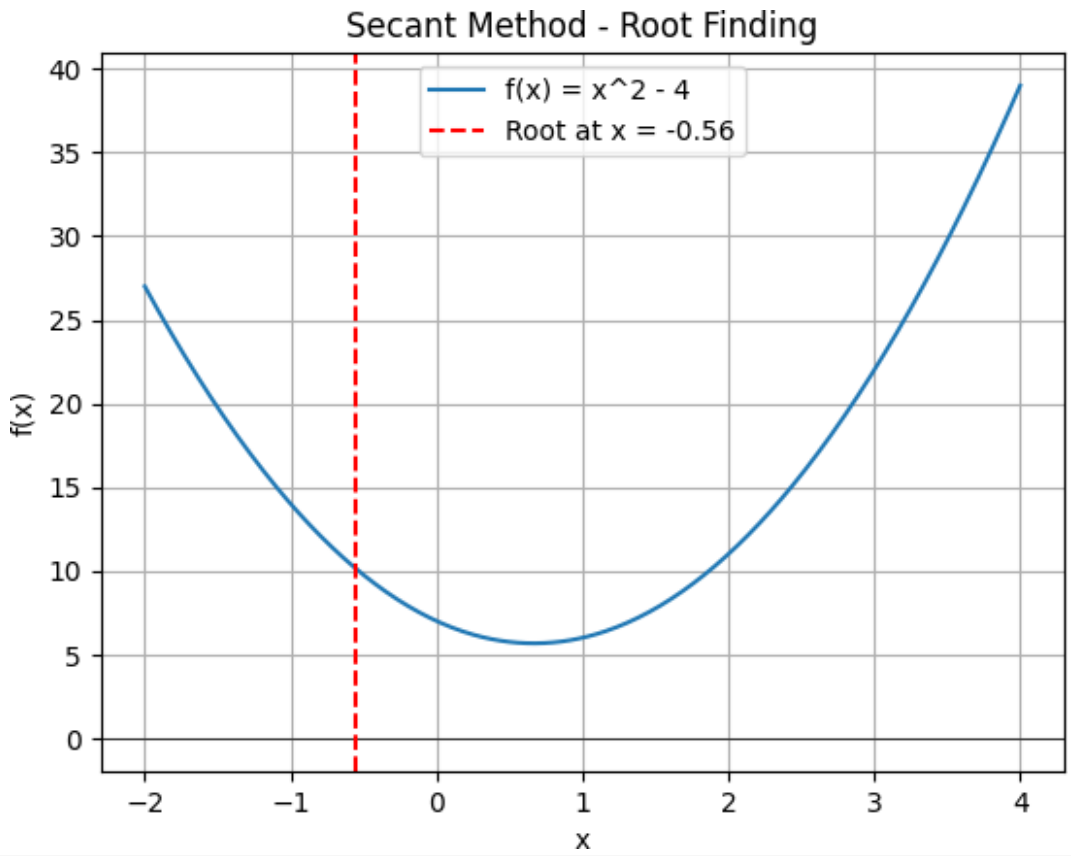
1. Given initial values x0 and x1, and tolerance ε.
2. Repeat until the absolute difference between successive approximations is less than ε:

   a. Calculate the next approximation: x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))

   b. Set x0 = x1 and x1 = x2.
3. Return the root approximation x2.

Code:

```
def secant_method(f, x0, x1, tol):
    while abs(x1 - x0) > tol:
        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        x0 = x1
        x1 = x2
    return x1

def f(x):
    return 3*x**2 - 4*x+7

# Secant method
root = secant_method(f, 1, 2, 0.01)

# Plotting the function and the root
x_vals = np.linspace(-2, 4, 400)
y_vals = f(x_vals)

plt.plot(x_vals, y_vals, label="f(x) = x^2 - 4")
plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(root, color='r', linestyle='--', label=f'Root at x =
{root:.2f}')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Secant Method - Root Finding')
plt.legend()
plt.grid(True)
plt.show()
```




```
import sympy as sp
import numpy as np

def example_function(x):
    # Example function: f(x) = x^3 - 5x - 9
    return x**3 - 5 * x - 9

def get_custom_function():
    # Prompt user to input a custom function as a string
    function_expression = input("Provide a mathematical function using 'x'
(e.g., x**3 - 5*x - 9): ")
    x = sp.symbols('x')
    symbolic_function = sp.sympify(function_expression)
    numeric_function = sp.lambdify(x, symbolic_function, modules=['numpy'])
    return numeric_function, symbolic_function

def secant_method_solver(function, guess1, guess2, error_limit,
max_iterations):
    # Secant method implementation to find the root of a function
    for iteration in range(1, max_iterations + 1):
        # Check to avoid division by zero
        if function(guess1) == function(guess2):
            print("Error: Division by zero encountered as f(guess1) and
f(guess2) are identical.")
            return None

        # Calculate the new approximation for the root
```

```
        next_guess = guess1 - (guess2 - guess1) * function(guess1) /
(function(guess2) - function(guess1))
        print(f"Step-{iteration}, x = {next_guess:.6f}, f(x) =
{function(next_guess):.6f}")
        print()

        # Check if the result is within the error tolerance
        if abs(function(next_guess)) <= error_limit:
            print(f"Root found: {next_guess:.8f}")
            return next_guess

        # Update guesses for the next iteration
        guess1, guess2 = guess2, next_guess

    # If the solution did not converge within the allowed steps
    print("The method failed to converge within the given iterations.")
    return None

def main():
    print("IMPLEMENTATION OF THE SECANT METHOD")
    print()

    # Default function and settings
    predefined_function = "x**3 - 5 * x - 9"
    numeric_function = example_function
    initial_guess1, initial_guess2, error_tolerance, max_steps = 2, 3, 0.0001,
50

    # User decision to use default or custom inputs
    use_defaults = input("Would you like to proceed with default settings?
(y/n): ").strip().lower() == 'y'
    if not use_defaults:
        numeric_function, predefined_function = get_custom_function()
        initial_guess1 = float(input("Enter the first approximation (guess1):
"))
        initial_guess2 = float(input("Enter the second approximation (guess2):
"))
        error_tolerance = float(input("Enter the acceptable error tolerance:
"))
        max_steps = int(input("Enter the maximum number of iterations allowed:
"))

    print(f"Selected function: f(x) = {predefined_function}")
    print(f"Initial guesses: guess1={initial_guess1}, guess2={initial_guess2}")
    print(f"Maximum iterations: {max_steps}, Error tolerance:
{error_tolerance}")
    print()

    # Call the solver
```

```
    secant_method_solver(numeric_function, initial_guess1, initial_guess2,
error_tolerance, max_steps)

if __name__ == "__main__":
    main()
```

Output:

IMPLEMENTATION OF THE SECANT METHOD

Would you like to proceed with default settings? (y/n): y
Selected function: f(x) = x**3 - 5 * x - 9
Initial guesses: guess1=2, guess2=3
Maximum iterations: 50, Error tolerance: 0.0001

Step-1, x = 2.785714, f(x) = -1.310860

Step-2, x = 2.850875, f(x) = -0.083923

Step-3, x = 2.855332, f(x) = 0.002635

Step-4, x = 2.855196, f(x) = -0.000005

Root found: 2.85519628

Test Cases:

Test for f(x) = x^2 - 4 starting with initial guesses x0 = 1 and x1 = 2 with tolerance 0.01.

Test for f(x) = e^x - 2x starting with initial guesses x0 = 0 and x1 = 1 with tolerance 0.001.

3.Newton Raphson method

Working Principle: The Newton-Raphson Method is an iterative root-finding algorithm that uses the derivative of the function to approximate the root. Starting with an initial guess x0, the method iterates using the formula $x_{n+1} = x_n - f(x_n) / f'(x_n)$ until the result converges to the root. This method is known for its rapid convergence, particularly when the initial guess is close to the true root.

Pseudocode:

1. Given initial guess x0, tolerance ε, and the derivative of the function f'(x).
2. Repeat until the difference between successive approximations is less than ε:

   a. Calculate the next approximation: x1 = x0 - f(x0) / f'(x0)

   b. Set x0 = x1.

3. Return the root approximation x1.

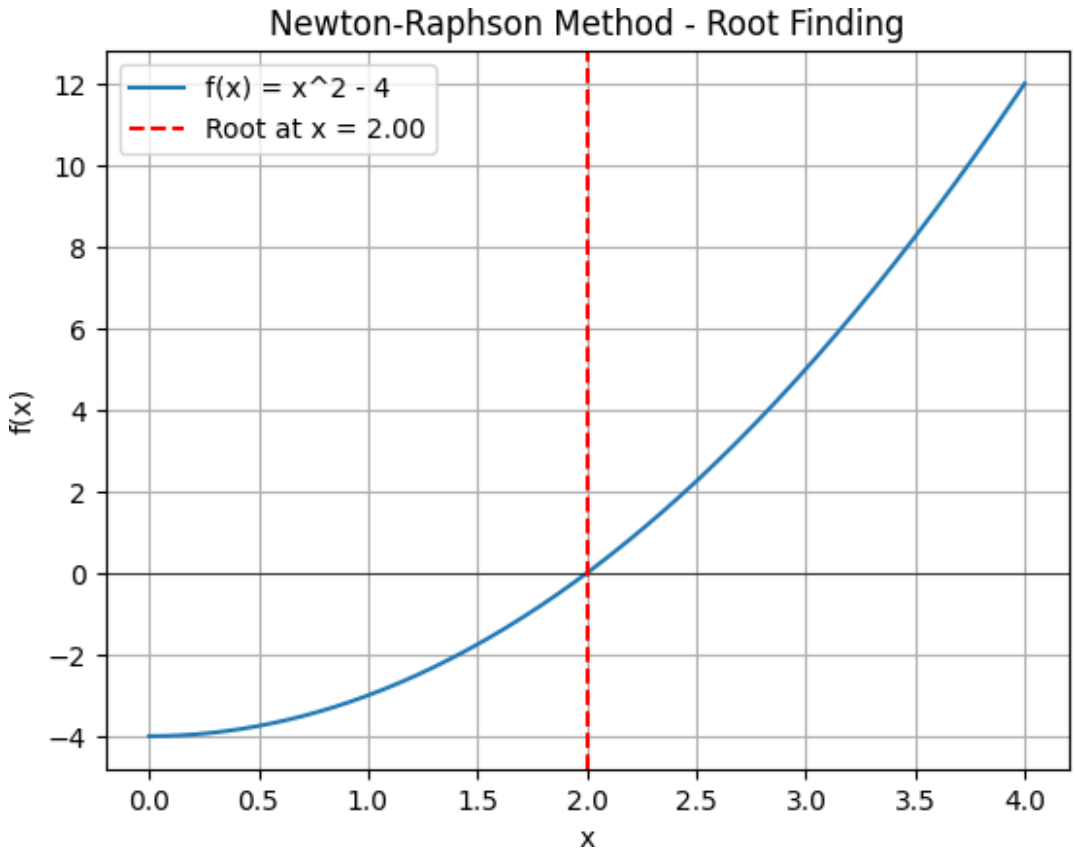
Code:

```
def newton_raphson(f, df, x0, tol=1e-6, max_iter=100):
    x = x0
    for i in range(max_iter):
        x_new = x - f(x) / df(x)
        print(f"Iteration {i+1}: x = {x_new:.6f}, f(x) = {f(x_new):.6f}")
        if abs(f(x_new)) < tol:
            return x_new
        x = x_new
    return None
f = lambda x: x**3 - 5 * x - 9
df = lambda x: 3 * x**2 - 5
root = newton_raphson(f, df, x0=2.0)
print(f"Root: {root:.6f}")
```

Iteration 1: x = 3.571429, f(x) = 18.696793
Iteration 2: x = 3.009378, f(x) = 3.207103
Iteration 3: x = 2.864712, f(x) = 0.185915
Iteration 4: x = 2.855236, f(x) = 0.000771
Iteration 5: x = 2.855197, f(x) = 0.000000

Root: 2.855197

```
def newton_raphson(f, df, x0, tol):
    while abs(f(x0)) > tol:
        x0 = x0 - f(x0) / df(x0)
    return x0

def newton_raphson(f, df, x0, tol):
    while abs(f(x0)) > tol:
        x0 = x0 - f(x0) / df(x0)
    return x0

# Example function and its derivative
def f(x):
    return x**2 - 4

def df(x):
    return 2*x

# Newton-Raphson method
root = newton_raphson(f, df, 2, 0.01)

# Plotting the function and the root
x_vals = np.linspace(0, 4, 400)
y_vals = f(x_vals)

plt.plot(x_vals, y_vals, label="f(x) = x^2 - 4")
plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(root, color='r', linestyle='--', label=f'Root at x =
{root:.2f}')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Newton-Raphson Method - Root Finding')
plt.legend()
plt.grid(True)
plt.show()
```




```
import sympy as sp

def f(x):
    return x**3 - 5 * x - 9

def g(x):
    return 3 * x**2 - 5

def func_input():
    function_str = input("Enter your function (use 'x' as the variable)
(Example: x**3 - 5 * x - 9): ")
    x = sp.symbols('x')
    sp_function = sp.sympify(function_str)
    func = sp.lambdify(x, sp_function, modules=['numpy'])
    return func, sp_function

def func_derivative(func):


    x = sp.symbols('x')
    sp_derivative = sp.diff(func, x)
    derivative = sp.lambdify(x, sp_derivative, modules=['numpy'])
    return derivative, sp_derivative

def newton_raphson_method(func, derivative, initial_guess, tolerance,
max_steps):
```

```
    current_guess = initial_guess
    for step in range(1, max_steps + 1):
        # Check for divide-by-zero error
        if derivative(current_guess) == 0.0:
            print("Divide by zero error! Derivative is zero.")
            return None

        # Update guess
        next_guess = current_guess - func(current_guess) /
derivative(current_guess)
        print(f"Iteration-{step}, x = {next_guess:.6f}, f(x) =
{f(next_guess):.6f}")
        print()

        # Check for convergence
        if abs(func(next_guess)) <= tolerance:
            print(f"Required root is: {next_guess:.8f}")
            return next_guess

        current_guess = next_guess

    # If maximum steps are exceeded
    print("Not Convergent within the maximum allowed steps.")
    return None


def main():
    print("NEWTON-RAPHSON METHOD IMPLEMENTATION")
    print()

    function_str = "x**3 - 5 * x - 9"
    derivative_str = "3 * x**2 - 5"
    func = f
    derivative = g
    initial_guess, tolerance, max_steps = 2, 0.0001, 50

    default = input("Use default values ? (y/n): ").strip().lower() == 'y'
    if not default:
        func, function_str = func_input()
        derivative, derivative_str = func_derivative(function_str)
        initial_guess = float(input("Enter initial guess (x0): "))
        tolerance = float(input("Enter tolerable error: "))
        max_steps = int(input("Enter maximum steps: "))

    print(f"f(x) = {function_str}")
    print(f"g(x) = {derivative_str}")
    print(f"Initial guess={initial_guess} Max steps={max_steps}
tolerance={tolerance}")
    print()
```

```
    # Perform the Newton-Raphson method
    newton_raphson_method(func, derivative, initial_guess, tolerance,
max_steps)


if __name__ == "__main__":
    main()
```

NEWTON-RAPHSON METHOD IMPLEMENTATION

Use default values ? (y/n): y
f(x) = x**3 - 5 * x - 9
g(x) = 3 * x**2 - 5
Initial guess=2 Max steps=50 tolerance=0.0001

Iteration-1, x = 3.571429, f(x) = 18.696793

Iteration-2, x = 3.009378, f(x) = 3.207103

Iteration-3, x = 2.864712, f(x) = 0.185915

Iteration-4, x = 2.855236, f(x) = 0.000771

Iteration-5, x = 2.855197, f(x) = 0.000000

Required root is: 2.85519654

Test Cases:

Test for f(x) = x^2 - 4 and its derivative f'(x) = 2x, starting with initial guess x0 = 2 and tolerance 0.01.

Test for f(x) = e^x - 2x and its derivative f'(x) = e^x - 2, starting with initial guess x0 = 1 and tolerance 0.001.

System of Non-Linear Equations using Newton- Raphson Method

Working Principle: The Newton-Raphson method can be extended to solve systems of non-linear equations. This is done by solving the Jacobian matrix of the system, which contains the first derivatives of each equation. The system is solved iteratively using a vector form of the Newton-Raphson method, where at each iteration, a correction vector is computed to update the solution.

Pseudo code:

1. Given initial guesses x0 for the system, tolerance ε, and the Jacobian matrix J(x).
   a. Repeat until the difference between successive approximations is less than ε:
      Compute the function vector F(x) and Jacobian matrix J(x).
   b. Solve the linear system J(x) * delta = -F(x) for delta.

c. Update the guess: x1 = x0 + delta.

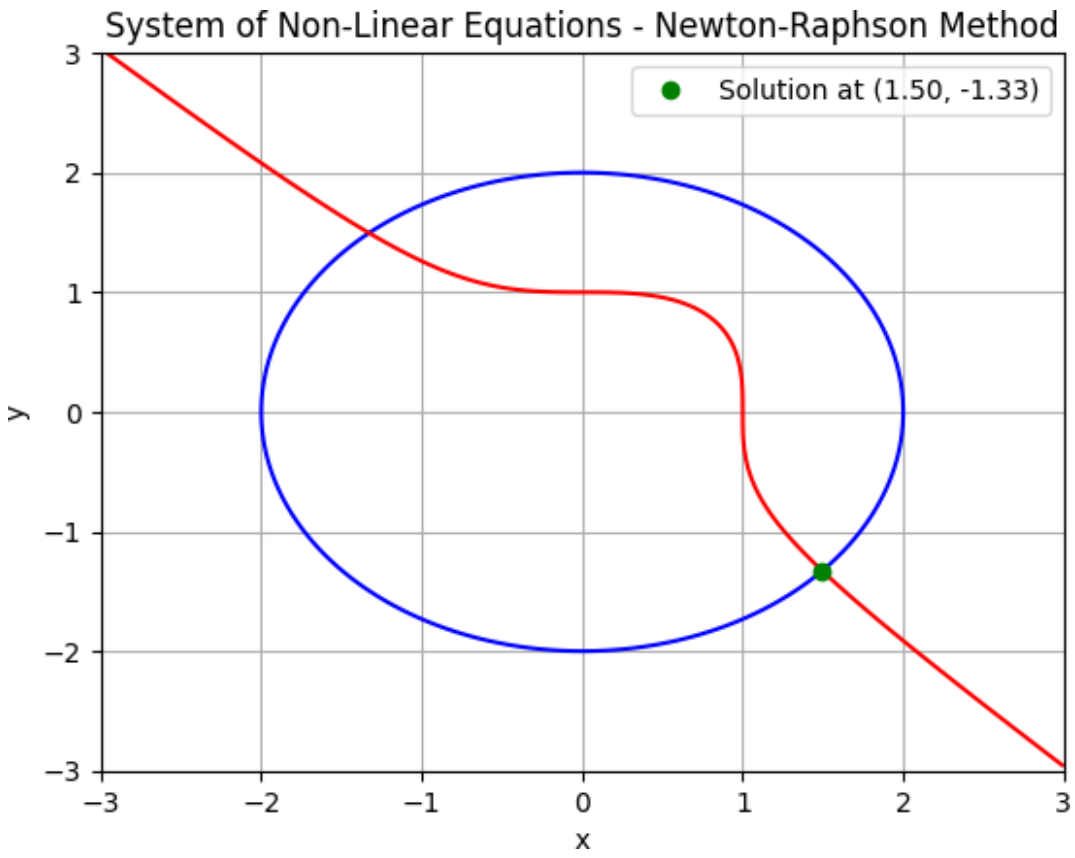
2. Return the solution vector x1.

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

# Example system of non-linear equations
def f1(x):
    return x[0]**2 + x[1]**2 - 4

def f2(x):
    return x[0]**3 + x[1]**3 - 1

# Function vector
def f(x):
    return np.array([f1(x), f2(x)])

# Jacobian matrix with regularization to avoid singularity
def jacobian(f, x, alpha=1e-6):
    n = len(x)
    J = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            h = 1e-5
            x_perturbed = np.copy(x)
            x_perturbed[j] += h
            J[i, j] = (f(x_perturbed)[i] - f(x)[i]) / h
    J += alpha * np.eye(n)  # Add a small value to diagonal to
regularize
    return J

# Newton's method for systems
def newton_system(f, x0, tol=1e-5, max_iter=100):
    iter_count = 0
    while np.linalg.norm(f(x0)) > tol and iter_count < max_iter:
        J = jacobian(f, x0)  # Compute Jacobian
        try:
            delta = np.linalg.solve(J, -f(x0))  # Solve for the update
        except np.linalg.LinAlgError:
            print("Jacobian is singular, cannot solve system.")
            return None
        x0 = x0 + delta  # Update the solution
        iter_count += 1
    if iter_count == max_iter:
        print("Maximum iterations reached without convergence.")
```

```
        return None
    return x0

# Initial guess
x0 = np.array([1.0, 1.0])

# Solve the system
root = newton_system(f, x0)

if root is not None:
    print(f"Solution found: x = {root}")

    # Plotting the root in 2D space
    x_vals = np.linspace(-3, 3, 400)
    y_vals = np.linspace(-3, 3, 400)
    X, Y = np.meshgrid(x_vals, y_vals)
    Z1 = X**2 + Y**2 - 4
    Z2 = X**3 + Y**3 - 1

    # Plotting the system
    plt.contour(X, Y, Z1, levels=[0], colors='b', label='f1(x, y) =
0')
    plt.contour(X, Y, Z2, levels=[0], colors='r', label='f2(x, y) =
0')
    plt.plot(root[0], root[1], 'go', label=f'Solution at
({root[0]:.2f}, {root[1]:.2f})')

    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('System of Non-Linear Equations - Newton-Raphson
Method')
    plt.legend()
    plt.grid(True)
    plt.show()
else:
    print("No solution found.")
```

```
Solution found: x = [ 1.4952723  -1.32821713]
```

```
<ipython-input-34-7a3a3370c2f8>:62: UserWarning: The following kwargs
were not used by contour: 'label'
  plt.contour(X, Y, Z1, levels=[0], colors='b', label='f1(x, y) = 0')
<ipython-input-34-7a3a3370c2f8>:63: UserWarning: The following kwargs
were not used by contour: 'label'
  plt.contour(X, Y, Z2, levels=[0], colors='r', label='f2(x, y) = 0')
```


System of Non-Linear Equations - Newton-Raphson Method
Solution at (1.50, -1.33)
3
2
1
0
−1
−2
−3
y
−3
−2
−1
0
1
2
3
x

LAB 3

1.Gauss Jordan Method

Working Principle:

The Gauss-Jordan method is an algorithm for solving a system of linear equations. It transforms the augmented matrix into a reduced row echelon form (RREF) using row operations. The solution is obtained directly from the final matrix.

Pseudo code Input: Augmented matrix [A|b] representing the system of linear equations Ax = b Output: Solution vector x

1. Let matrix A be of size n x n and vector b of size n.
2. Combine A and b into an augmented matrix [A|b]. AugmentedMatrix = [A | b]
3. For each row i from 1 to n:

   a. Make the leading coefficient (the pivot) of the current row 1:

   - Find the pivot element A[i, i] (diagonal element of the current row).

   - If the pivot is not 1, divide the entire row i by the pivot (A[i, i]). - This will make A[i, i] = 1.

   - RowOperation: Row[i]= Row[i] / A[i, i]

   b. For each other row j (j ≠ i), eliminate the variable corresponding to the current column (A[i, i]):

   - Update row j to make the entry at position j, i (A[j, i]) zero.

   - Subtract a multiple of row i from row j, so that A[j, i] = 0.

   - RowOperation: Row[j] = Row[j] - A[j, i] * Row[i]

4. After all rows have been processed, the augmented matrix will be in reduced row echelon form (RREF):
   - The left n x n part of the augmented matrix [A|b] will be an identity matrix I.
   - The right column will contain the solution vector x.
5. Return the solution vector x (the last column of the augmented matrix).
   x =LastColumn(augmentedMatrix)

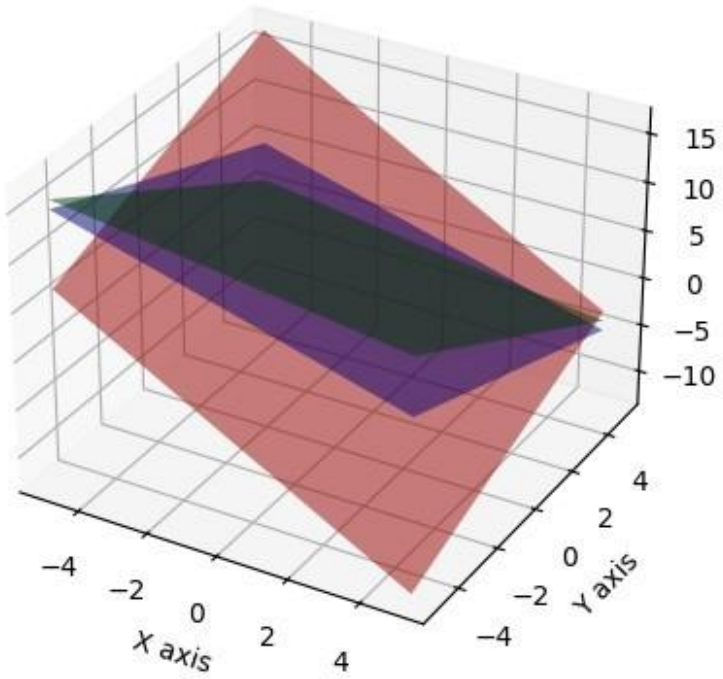
```
import numpy as np
```

```
def gauss_jordan(A, b):
    n = len(b)
    AugmentedMatrix = np.hstack([A, b.reshape(-1, 1)])  # Create
augmented matrix [A|b]

    # Perform Gauss-Jordan elimination
    for i in range(n):
        AugmentedMatrix[i] = AugmentedMatrix[i] / AugmentedMatrix[i,
i]  # Scale pivot to 1
        for j in range(n):
            if i != j:
                AugmentedMatrix[j] = AugmentedMatrix[j] -
AugmentedMatrix[j, i] * AugmentedMatrix[i]

    # Return the solution vector
    return AugmentedMatrix[:, -1]

# Example Usage
A = np.array([[2, -1, 1], [1, 3, 2], [1, 1, 1]], dtype=float)
b = np.array([2, 12, 5], dtype=float)
solution = gauss_jordan(A, b)
print("Solution:", solution)
```

```
Solution: [-1.  1.  5.]
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Create the figure and 3D axis
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Define the plane equations based on the system
X = np.linspace(-5, 5, 100)
Y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(X, Y)
Z1 = (2 - 2*X + Y) / 1
Z2 = (12 - X - 3*Y) / 2
Z3 = 5 - X - Y

# Plot the planes
ax.plot_surface(X, Y, Z1, alpha=0.5, rstride=100, cstride=100,
color='r')
ax.plot_surface(X, Y, Z2, alpha=0.5, rstride=100, cstride=100,
color='g')
ax.plot_surface(X, Y, Z3, alpha=0.5, rstride=100, cstride=100,
color='b')

ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
```

```
ax.set_zlabel('Z axis')
plt.show()
```




```
import numpy as np
import sys


def read_augmented_matrix(n):
    print("Enter Augmented Matrix Coefficients:")
    matrix = np.zeros((n, n + 1))
    for i in range(n):
        for j in range(n + 1):
            matrix[i][j] = float(input(f"a[{i}][{j}] = "))
    print(matrix)
    return matrix


def gauss_jordan_elimination(a):
    n = len(a)
    for i in range(n):
        # Check for divide-by-zero
        if a[i][i] == 0.0:
            sys.exit("Divide by zero detected in pivot element!")

        for j in range(n):
            if i != j:
                ratio = a[j][i] / a[i][i]
                for k in range(n + 1):
                    a[j][k] -= ratio * a[i][k]
```

```
    # Extracting the solution
    x = np.zeros(n)
    for i in range(n):
        x[i] = a[i][n] / a[i][i]

    return x


def main():
    print("GAUSS-JORDAN ELIMINATION METHOD")
    print()

    default = input("Use default 3x4 matrix? (y/n): ").strip().lower() == 'y'

    if default:
        # Default 3x4 augmented matrix
        n = 3
        a = np.array([
            [2, 1, -1, 8],
            [-3, -1, 2, -11],
            [-2, 1, 2, -3]
        ], dtype=float)
        print("Using Default Matrix:")
        print(a)
    else:
        n = int(input("Enter number of unknowns: "))
        a = read_augmented_matrix(n)

    # Solve using Gauss-Jordan Elimination
    solution = gauss_jordan_elimination(a)

    # Display the solution
    print("Required solution is:")
    for i, value in enumerate(solution):
        print(f"X{i} = {value:.2f}", end="\t")


if __name__ == "__main__":
    main()
```

GAUSS-JORDAN ELIMINATION METHOD

Use default 3x4 matrix? (y/n): y
Using Default Matrix:
[[2. 1. -1. 8.]
 [-3. -1. 2. -11.]
 [-2. 1. 2. -3.]]
Required solution is:
    X0 = 2.00   X1 = 3.00   X2 = -1.00

Test Case: For the system of equations: 2x - y + z = 2 x + 3y + 2z = 12 x + y + z = 5

2.Gauss elimination method with partial pivoting

Working Principle: The Gauss Elimination Method transforms a system of linear equations into an upper triangular matrix by using row operations. Partial Pivoting is used to reduce numerical errors by swapping rows to ensure that the largest absolute value of each column is placed at the pivot position. This minimizes the potential for rounding errors during elimination.

Steps for Gauss Elimination with Partial Pivoting:

Forward Elimination: Using row operations to eliminate variables in each column and create an upper triangular matrix. For each column, identify the largest absolute value in that column (pivot) and swap rows. Eliminate all entries below the pivot using row operations. Back Substitution: After obtaining the upper triangular matrix, solve for the unknowns by substituting values from the last row upwards.

Pseudocode:

Input: Augmented matrix [A|b] of size n x n+1 (for system Ax = b) Output: Solution vector x of size n

1. For i = 1 to n-1:

a. Find the pivot row with the largest absolute value in column i.

b. Swap the current row with the pivot row.

c. For j = i+1 to n: - Eliminate the i-th variable from row j: Row[j] = Row[j] - (A[j, i] / A[i, i]) * Row[i]

2. Back Substitution:

a. For i = n-1 to 0: - x[i] = (b[i] - sum(A[i, j] * x[j] for j = i+1 to n)) / A[i, i]

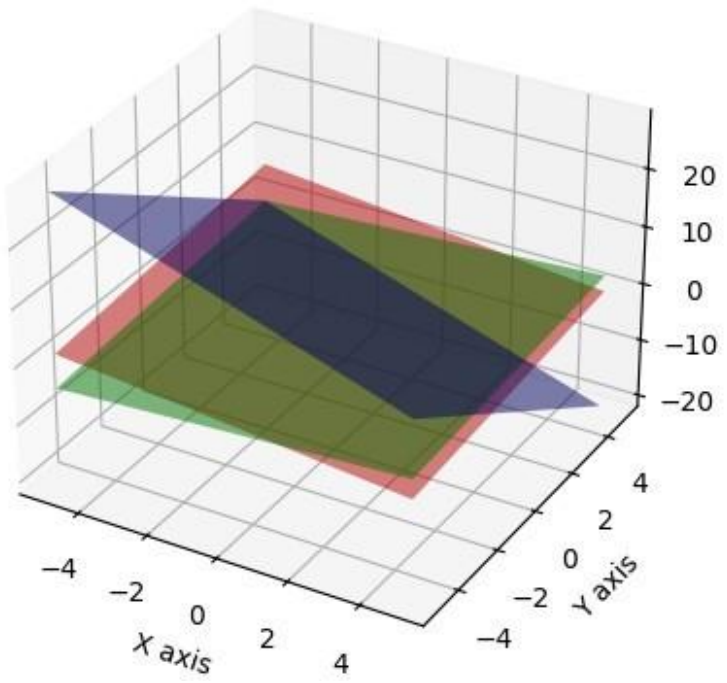
3. Return the solution vector x.

```
import numpy as np

def gauss_elimination_partial_pivoting(A, b):
    n = len(b)
    AugmentedMatrix = np.hstack([A, b.reshape(-1, 1)])  # Create
augmented matrix [A|b]

    # Forward Elimination
    for i in range(n):
        # Pivoting: Find the maximum element in the current column
        max_row = np.argmax(np.abs(AugmentedMatrix[i:n, i])) + i
        AugmentedMatrix[[i, max_row]] = AugmentedMatrix[[max_row, i]]
# Swap rows

        # Eliminate entries below the pivot
        for j in range(i+1, n):
            factor = AugmentedMatrix[j, i] / AugmentedMatrix[i, i]
            AugmentedMatrix[j, i:] -= factor * AugmentedMatrix[i, i:]

    # Back Substitution
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = (AugmentedMatrix[i, -1] - np.dot(AugmentedMatrix[i,
i+1:n], x[i+1:n])) / AugmentedMatrix[i, i]

    return x

# Example Usage
A = np.array([[3, -2,  5], [1,  1, -3], [2,  3,  1]], dtype=float)
b = np.array([3, 3, 4], dtype=float)

solution = gauss_elimination_partial_pivoting(A, b)
print("Solution:", solution)
```

```
Solution: [ 1.73469388  0.28571429 -0.32653061]
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Create the figure and 3D axis
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

```
# Define the plane equations based on the system
X = np.linspace(-5, 5, 100)
Y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(X, Y)
Z1 = (3 - 3*X + 2*Y) / 5
Z2 = (3 - X - Y) / -3
Z3 = (4 - 2*X - 3*Y) / 1

# Plot the planes
ax.plot_surface(X, Y, Z1, alpha=0.5, rstride=100, cstride=100,
color='r')
ax.plot_surface(X, Y, Z2, alpha=0.5, rstride=100, cstride=100,
color='g')
ax.plot_surface(X, Y, Z3, alpha=0.5, rstride=100, cstride=100,
color='b')

ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
plt.show()
```



Test Case: For the system of equations: 3x -

2y + 5z = 3 x + y - 3z = 3 2x + 3y + z = 4

3.Gauss-Seidal method

Working Principle: The Gauss-Seidel method is an iterative technique that modifies an initial guess for the solution vector x=(x 1 ,x 2 ,...,x n) to approach the true solution of the system of equations A $x = b$. The method updates each variable in turn using the most recent values available for the other variables.

Steps:

1. Start with an initial guess: $\boldsymbol{x^0} = (x_1^0, x_2^0, \ldots, x_n^0)$
2. For each variable x_i, calculate the new value:

$$x_i^{\text{new}} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{\text{new}} - \sum_{j=i+1}^{n} a_{ij}x_j^{\text{old}}\right)$$

where a_{ij} are the coefficients of the matrix $\boldsymbol{A}$, and b_i are the elements of the vector $\boldsymbol{b}$.

3. Repeat the process until the solution converges, i.e., the difference between the new and old solutions is sufficiently small (within a specified tolerance).

Pseudocode:

Input: Coefficient matrix A (n x n), vector b (n x 1), initial guess x0 (n x 1), tolerance tol, max iterations max_iter Output: Solution vector x (n x 1)

1. Set k = 0 (iteration counter)
2. Set x = x0 (initial guess)
3. While k < max_iter:
   a. For i = 1 to n: - Calculate the new value of x[i]: x[i] = (b[i] - sum(A[i,j] * x[j] for j = 1 to i-1) - sum(A[i,j] * x[j] for j = i+1 to n)) / A[i,i]
   b. Check if the solution has converged: - If the maximum difference between the new and old x is less than tol, break
   c. Increment k by 1
4. Return the solution vector x

```
import numpy as np

def gauss_seidel(A, b, x0, tol=1e-10, max_iter=1000):
    n = len(b)
    x = np.copy(x0)  # Initial guess
    for k in range(max_iter):
        x_old = np.copy(x)

        # Iterate over each variable
        for i in range(n):
            sum1 = np.dot(A[i, :i], x[:i])    # Sum for previous
variables
            sum2 = np.dot(A[i, i+1:], x_old[i+1:])  # Sum for future
```

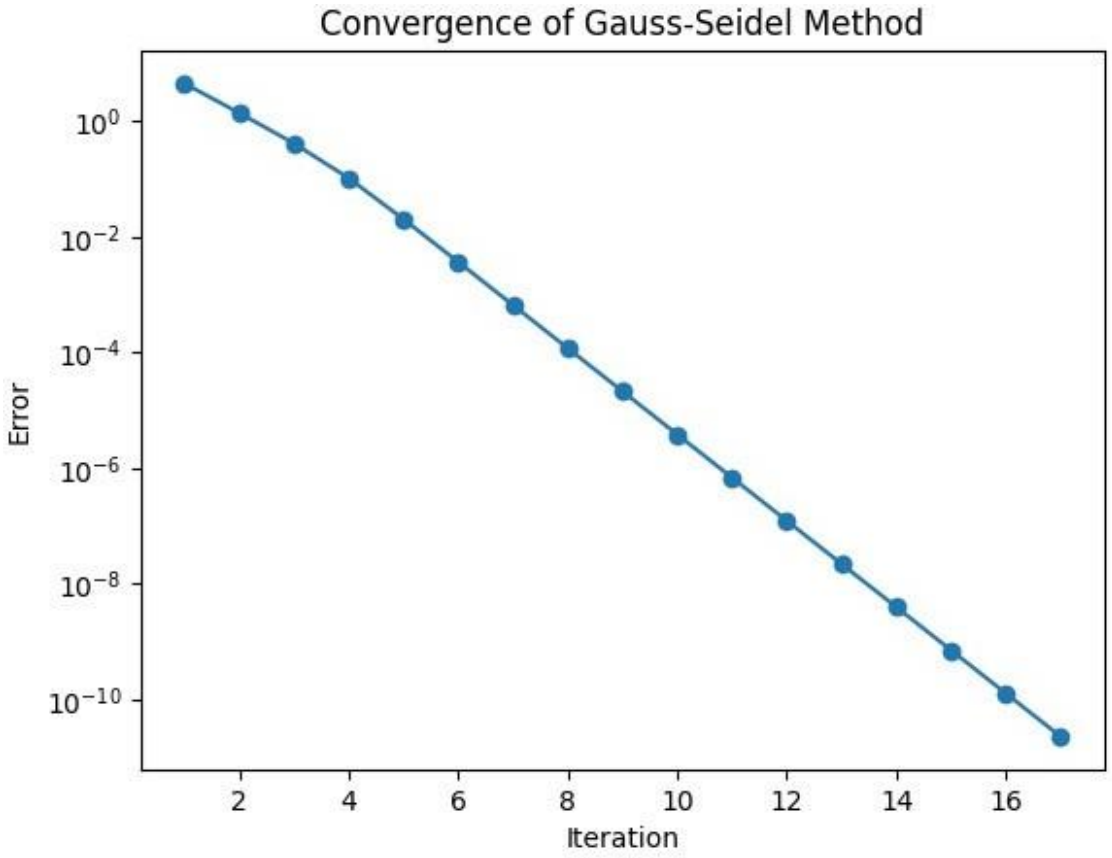
```
variables
            x[i] = (b[i] - sum1 - sum2) / A[i, i]  # Update variable

        # Check for convergence (if the change is small)
        if np.linalg.norm(x - x_old, ord=np.inf) < tol:
            print(f"Converged after {k+1} iterations")
            return x
    print("Max iterations reached")
    return x

# Example Usage
A = np.array([[4, -1, 0, 0],
              [-1, 4, -1, 0],
              [0, -1, 4, -1],
              [0, 0, -1, 3]], dtype=float)

b = np.array([15, 10, 10, 10], dtype=float)

x0 = np.zeros_like(b)

solution = gauss_seidel(A, b, x0)
print("Solution:", solution)
```

```
Converged after 17 iterations
Solution: [5. 5. 5. 5.]
```

```
import matplotlib.pyplot as plt

def gauss_seidel_with_error_plot(A, b, x0, tol=1e-10, max_iter=1000):
    n = len(b)
    x = np.copy(x0)
    error = []

    for k in range(max_iter):
        x_old = np.copy(x)

        # Iterate over each variable
        for i in range(n):
            sum1 = np.dot(A[i, :i], x[:i])    # Sum for previous
variables
            sum2 = np.dot(A[i, i+1:], x_old[i+1:])  # Sum for future
variables
            x[i] = (b[i] - sum1 - sum2) / A[i, i]  # Update variable

        # Compute error (max difference between old and new x)
        err = np.linalg.norm(x - x_old, ord=np.inf)
        error.append(err)

        # Check for convergence
        if err < tol:
            print(f"Converged after {k+1} iterations")
```

```
            break

    # Plot the error for each iteration
    plt.plot(range(1, len(error)+1), error, marker='o')
    plt.yscale('log')  # Log scale to show convergence better
    plt.xlabel('Iteration')
    plt.ylabel('Error')
    plt.title('Convergence of Gauss-Seidel Method')
    plt.show()

    return x

# Example Usage with Error Plot
solution = gauss_seidel_with_error_plot(A, b, x0)
print("Solution:", solution)
```

```
Converged after 17 iterations
```




```
Solution: [5. 5. 5. 5.]
```

Test Case 1: Consider the system of equations:

4x - y = 15 -x + 4y - z = 10 -y + 4z - w = 10 -z + 3w = 10

4.Power Method

Working Principle:

The Power Method is an iterative technique used to find the largest eigenvalue and its corresponding eigenvector of a matrix. The method works as follows:

1. Start with an initial guess for the eigenvector x0.
2. Multiply the matrix A by the current eigenvector approximation x_k to get a new vector y_k

$$y_k = A \cdot x_k$$

3. Normalize the resulting vector y_k to avoid numerical overflow:

$$x_{k+1} = \frac{y_k}{|y_k|}$$

4. Calculate the Rayleigh quotient to estimate the eigenvalue:

$$\lambda_k = \frac{x_k^T \cdot A \cdot x_k}{x_k^T \cdot x_k}$$

5. Repeat the process until the difference between consecutive eigenvalue estimates is smaller than a given tolerance.

Pseudocode: Input: Matrix A, Initial vector x0, Tolerance tol, Maximum iterations max_iter
Output: Eigenvalue λ, Eigenvector x

1. Initialize x0 with a random guess
2. Normalize x0
3. Set λ_old to 0
4. For k = 1 to max_iter:

   a. Multiply A by x_k to get y_k

   b. Normalize y_k to get x_(k+1)

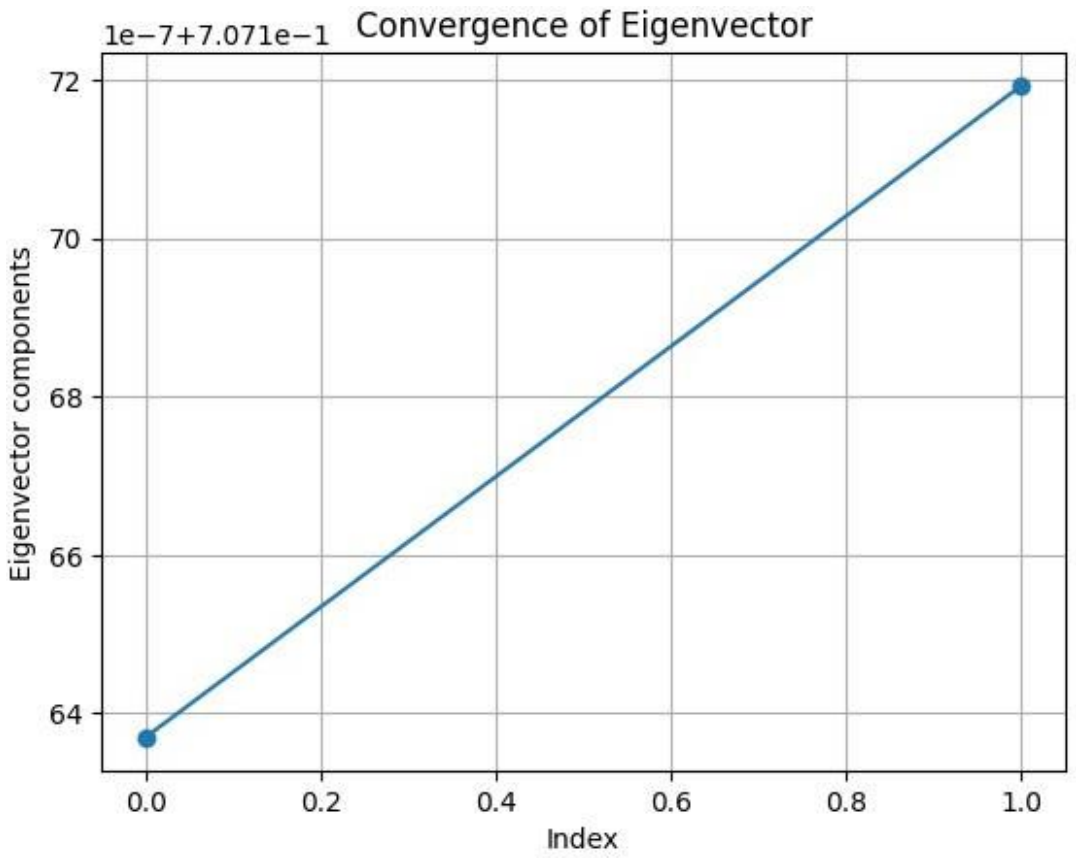
   c. Compute the Rayleigh quotient to estimate λ_k: λ_k = (x_k^T * A * x_k) / (x_k^T * x_k)

   d. If abs(λ_k - λ_old) < tol, break

   e. Set λ_old = λ_k
5. Return λ_k, x_k

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

def power_method(A, x0, tol=1e-6, max_iter=1000):
    # Normalize the initial guess
    x = x0 / np.linalg.norm(x0)
    lambda_old = 0

    # Iterate until convergence
    for i in range(max_iter):
        # Multiply A with the current eigenvector approximation
        y = np.dot(A, x)

        # Normalize the resulting vector
        x = y / np.linalg.norm(y)

        # Compute the Rayleigh quotient for eigenvalue
        lambda_new = np.dot(x.T, np.dot(A, x))

        # Check for convergence
        if abs(lambda_new - lambda_old) < tol:
            break

        lambda_old = lambda_new

    return lambda_new, x

# Example Test Case
A = np.array([[4, 1], [2, 3]])
x0 = np.random.rand(2)  # Random initial guess

# Call power method
eigenvalue, eigenvector = power_method(A, x0)

print("Largest Eigenvalue:", eigenvalue)
print("Corresponding Eigenvector:", eigenvector)

# Visualize the result
plt.plot(np.arange(len(eigenvector)), eigenvector, marker='o')
plt.title('Convergence of Eigenvector')
plt.xlabel('Index')
plt.ylabel('Eigenvector components')
plt.grid(True)
plt.show()
```

```
Largest Eigenvalue: 4.999999417469951
Corresponding Eigenvector: [0.70710637 0.70710719]
```




```python
import numpy as np

def read_matrix(n):
    matrix = np.zeros((n, n))
    print("Enter Matrix Coefficients:")
    for i in range(n):
        for j in range(n):
            matrix[i][j] = float(input(f"a[{i}][{j}] = "))
    return matrix


def read_vector(n):
    vector = np.zeros(n)
    print("Enter Initial Guess Vector:")
    for i in range(n):
        vector[i] = float(input(f"x[{i}] = "))
    return vector


def power_method(a, x, tolerable_error, max_iterations):
    lambda_old = 1.0
    step = 1
    while True:
        # Multiply matrix and vector
        x = np.matmul(a, x)
```

```
        # Compute new eigenvalue and normalize the vector
        lambda_new = max(abs(x))
        x = x / lambda_new

        # Display results
        print()
        print(f"STEP {step}")
        print("-" * 10)
        print(f"Eigenvalue = {lambda_new:.4f}")
        print("Eigenvector:")
        print("[", end=" ")
        for val in x:
            print(f"{val:.4f}", end="\t")
        print("]")

        # Check convergence
        error = abs(lambda_new - lambda_old)
        print(f"Error = {error:.6f}")
        if error < tolerable_error:
            print()
            print("Converged!")
            break

        if step >= max_iterations:
            print()
            print("Not convergent in given maximum iterations!")
            break

        lambda_old = lambda_new
        step += 1


def main():
    print("POWER METHOD IMPLEMENTATION")
    print()

    default = input("Use default matrix and vector? (y/n): ").strip().lower()
== "y"

    if default:
        # Default matrix and vector
        a = np.array([[2, 1, 1],
                      [1, 3, 2],
                      [1, 0, 0]], dtype=float)
        x = np.array([1, 1, 1], dtype=float)
        tolerable_error = 0.0001
        max_iterations = 100
        print("Using Default Values:")
```

```
        print("Matrix:")
        print(a)
        print("Initial Guess Vector:")
        print(x)
        print()
        print(f"Tolerable Error: {tolerable_error}")
        print(f"Maximum Iterations: {max_iterations}")
    else:
        # User input
        n = int(input("Enter order of matrix: "))
        a = read_matrix(n)
        x = read_vector(n)
        tolerable_error = float(input("Enter tolerable error: "))
        max_iterations = int(input("Enter maximum number of steps: "))

    power_method(a, x, tolerable_error, max_iterations)


if __name__ == "__main__":
    main()
```

POWER METHOD IMPLEMENTATION

Use default matrix and vector? (y/n): y
Using Default Values:
Matrix:
[[2. 1. 1.]
 [1. 3. 2.]
 [1. 0. 0.]]
Initial Guess Vector:
[1. 1. 1.]

Tolerable Error: 0.0001
Maximum Iterations: 100

STEP 1

Eigenvalue = 6.0000
Eigenvector:
[0.6667 1.0000 0.1667]
Error = 5.000000

STEP 2

Eigenvalue = 4.0000
Eigenvector:
[0.6250 1.0000 0.1667]
Error = 2.000000

STEP 3

Eigenvalue = 3.9583
Eigenvector:
[0.6105 1.0000 0.1579]
Error = 0.041667

STEP 4

Eigenvalue = 3.9263
Eigenvector:
[0.6059 1.0000 0.1555]
Error = 0.032018

STEP 5

Eigenvalue = 3.9169
Eigenvector:
[0.6044 1.0000 0.1547]
Error = 0.009426

STEP 6

Eigenvalue = 3.9138
Eigenvector:
[0.6039 1.0000 0.1544]
Error = 0.003132

STEP 7

Eigenvalue = 3.9127
Eigenvector:
[0.6037 1.0000 0.1543]
Error = 0.001026

STEP 8

Eigenvalue = 3.9124
Eigenvector:
[0.6037 1.0000 0.1543]
Error = 0.000337

STEP 9

Eigenvalue = 3.9123
Eigenvector:
[0.6036 1.0000 0.1543]
Error = 0.000111

STEP 10

Eigenvalue = 3.9122
Eigenvector:
[0.6036 1.0000 0.1543]
Error = 0.000036

Test Cases:

Input: $A=\begin{matrix}4 & 1\\ 2 & 3\end{matrix}$,

Initial guess =[0.5,0.5]

Expected Output: Eigenvalue≈5,

Eigenvector ≈ [0.707, 0.707]

LAB 4

1.Newton's Forward Difference Interpolation

Working Principle:
Newton's Forward Difference Interpolation is a method for estimating the values of a function for intermediate values of the independent variable when a table of values is available. The method uses the forward differences, which are based on the values at the given data points, to create an interpolation polynomial. The formula for the Newton Forward Difference interpolation is:

$$P(x) = y_0 + (x - x_0) \cdot \Delta y_0 + \{(x - x_0)(x - x_1)\}/\{2!\} \cdot \Delta^2 y_0 + \cdots$$

Where:

- $P(x)$ is the interpolated value at point xxx.
- $y_0$is the known value at x0x_0x0.
- Δy_0 is the first forward difference.
- $\Delta^2 y_0$ is the second forward difference, and so on.

The forward differences are calculated using the following recursive relations:

First forward difference: $\Delta y_i = y_{\{i+1\}} - y_i$

Second forward difference: $\Delta^2 y_i = \Delta y_{\{i+1\}} - \Delta y_i$

Pseudocode: Input: Array of data points (x, y), Value of x to interpolate (x_value), Number of data points (n)
Output: Interpolated value at x_value

1. Calculate the forward differences using the given data points
2. Initialize the interpolation polynomial P(x) with the first y value
3. For i = 1 to n-1:

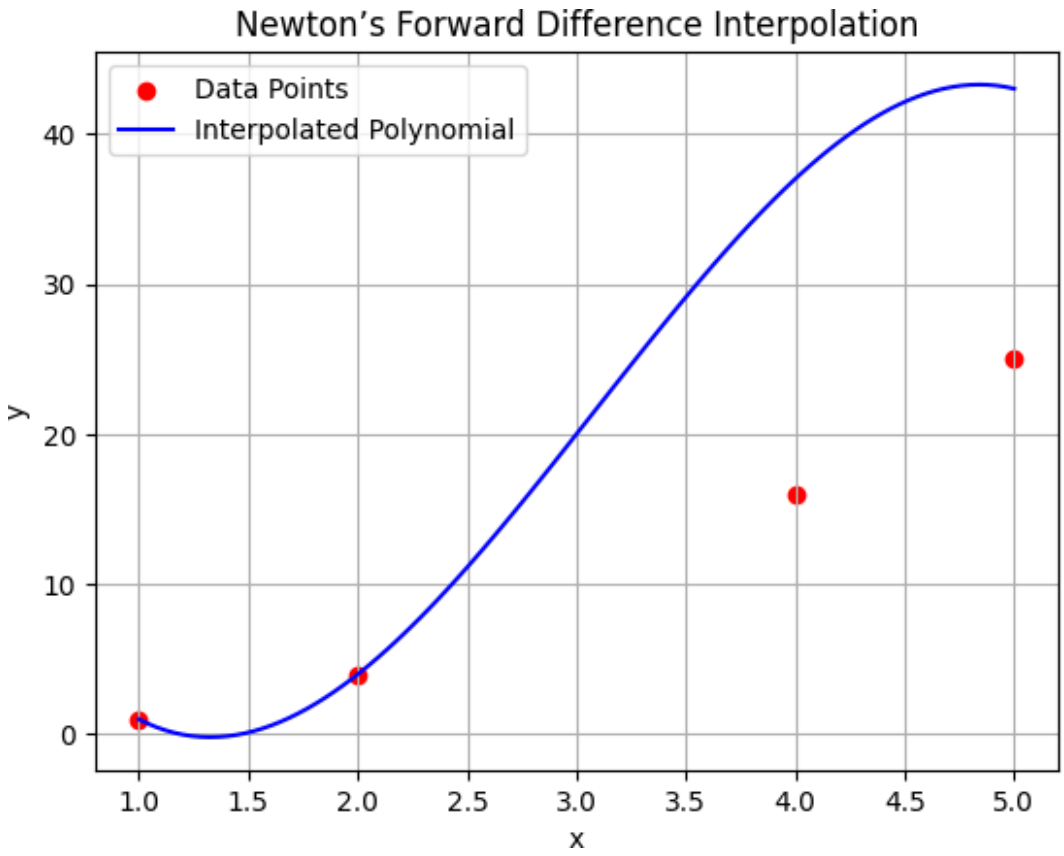
a. Calculate the term for the i-th forward difference
b. Update the polynomial P(x)

4. Return the value of the polynomial P(x) at x_value

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

# Function to calculate forward differences
def forward_difference(y_values):
    n = len(y_values)
    diff_table = np.zeros((n, n))
    diff_table[:, 0] = y_values  # First column is the given y-values

    for j in range(1, n):
        for i in range(n-j):
            diff_table[i][j] = diff_table[i+1][j-1] - diff_table[i][j-
1]

    return diff_table

# Function for Newton's Forward Difference Interpolation
def newton_forward_interpolation(x_values, y_values, x_value):
    n = len(x_values)
    diff_table = forward_difference(y_values)
    result = y_values[0]  # First term of the interpolation polynomial

    h = x_values[1] - x_values[0]  # Assuming equally spaced x-values

    # Iteratively calculate the terms of the interpolation polynomial
    for i in range(1, n):
        term = diff_table[0][i]
        for j in range(i):
            term *= (x_value - x_values[j])

        term /= np.math.factorial(i)
        result += term

    return result

# Example Test Case
x_values = np.array([1, 2, 4, 5])  # Given x-values
y_values = np.array([1, 4, 16, 25])  # Given y-values

# Value to interpolate
x_value = 3

# Call the interpolation function
interpolated_value = newton_forward_interpolation(x_values, y_values,
x_value)
```

```
print(f"The interpolated value at x = {x_value} is:
{interpolated_value}")

# Plotting the given data points and interpolation
x_interp = np.linspace(min(x_values), max(x_values), 100)
y_interp = [newton_forward_interpolation(x_values, y_values, xi) for
xi in x_interp]

plt.scatter(x_values, y_values, color='red', label='Data Points')
plt.plot(x_interp, y_interp, color='blue', label='Interpolated
Polynomial')
plt.title('Newton's Forward Difference Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

```
<ipython-input-1-3ff8318bbf79>:30: DeprecationWarning: `np.math` is a
deprecated alias for the standard library `math` module (Deprecated
Numpy 1.25). Replace usages of `np.math` with `math`
  term /= np.math.factorial(i)

The interpolated value at x = 3 is: 20.0
```



Test cases

Test Case 1:

- Input: x=[1,2,4,5], y=[1,4,16,25], x_value=3
- Expected Output: Interpolated value ≈ 9

Test Case 2:

- Input: x=[0,1,2,3], y=[1,2,4,9], x_value=1.5
- Expected Output: Interpolated value ≈ 3.25

Test Case 3:

- Input: x=[1,3,5,7], y=[2,4,8,16], x_value=4
- Expected Output: Interpolated value ≈ 6.25

2.Lagrange interpolation

Working Principle:

Lagrange Interpolation is a method for estimating the values of a function given a set of data points. It involves constructing a polynomial that passes through all the given points. The general form of the Lagrange polynomial is:

$$\boldsymbol{P(x) = L_{0(x)} \cdot y_0 + L_{1(x)} \cdot y_1 + \ldots + L_{n(x)} \cdot y_n}$$

Where:

- P(x) is the interpolated polynomial at the point xxx,
- $\boldsymbol{y_i}$ are the given values (output for each data point),
- $\boldsymbol{L_{i(x)}}$ are the Lagrange basis polynomials, defined as:

$$\boldsymbol{L_{i(x)} = \prod_{\{j=0, j\neq i\}}^{\{n\}} \{x - x_j\}/\{x_i - x_j\}}$$

The Lagrange basis polynomial $L_{i(x)}$ is constructed to be equal to 1 at $x = x_i$ and 0 at all other data points x_j, for $j \neq i$. The final polynomial is formed by summing all such products for each data point

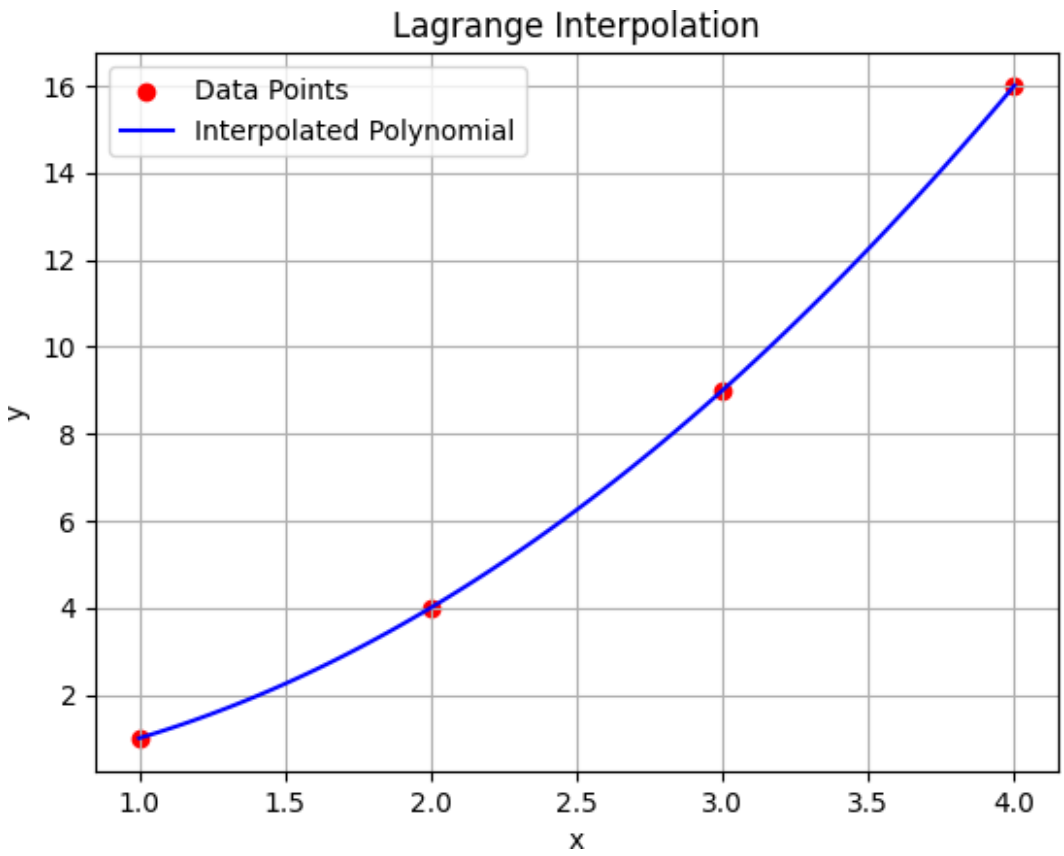
Pseudo code Input: Array of data points (x, y), Value of x to interpolate (x_value), Number of data points (n) Output: Interpolated value at x_value

1. Initialize P(x) = 0 (this will hold the final interpolated value)
2. For i = 0 to n-1:
   a. Initialize L(x) = 1 (Lagrange basis polynomial)
   b. For j = 0 to n-1: If j ≠ i: L(x) *= (x_value - x[j]) / (x[i] - x[j])
   c. Update P(x) by adding (L(x) * y[i]) to it.
3. Return the value of P(x) at x_value

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

# Function to compute Lagrange interpolation
def lagrange_interpolation(x_values, y_values, x_value):
    n = len(x_values)
    result = 0
```

```
    for i in range(n):
        # Calculate Lagrange basis polynomial L_i(x)
        term = y_values[i]
        for j in range(n):
            if j != i:
                term *= (x_value - x_values[j]) / (x_values[i] -
x_values[j])
        result += term

    return result

# Example Test Case
x_values = np.array([1, 2, 3, 4])  # Given x-values
y_values = np.array([1, 4, 9, 16])  # Given y-values

# Value to interpolate
x_value = 2.5

# Call the interpolation function
interpolated_value = lagrange_interpolation(x_values, y_values,
x_value)

print(f"The interpolated value at x = {x_value} is:
{interpolated_value}")

# Plotting the given data points and interpolation
x_interp = np.linspace(min(x_values), max(x_values), 100)
y_interp = [lagrange_interpolation(x_values, y_values, xi) for xi in
x_interp]

plt.scatter(x_values, y_values, color='red', label='Data Points')
plt.plot(x_interp, y_interp, color='blue', label='Interpolated
Polynomial')
plt.title('Lagrange Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

```
The interpolated value at x = 2.5 is: 6.25
```




```python
import numpy as np


def read_data_points(n):
    x = np.zeros(n)
    y = np.zeros(n)
    print("Enter data for x and y: ")
    for i in range(n):
        x[i] = float(input(f"x[{i}] = "))
        y[i] = float(input(f"y[{i}] = "))
    return x, y


def lagrange_interpolation(x, y, xp):
    n = len(x)
    yp = 0
    for i in range(n):
        p = 1
        for j in range(n):
            if i != j:
                p *= (xp - x[j]) / (x[i] - x[j])
        yp += p * y[i]
    return yp


def main():
```

```
    print("LAGRANGE INTERPOLATION")
    print()

    default = input("Use default data points? (y/n): ").strip().lower() == 'y'

    if default:
        x = np.array([0, 1, 2, 3], dtype=float)
        y = np.array([1, 2, 0, 5], dtype=float)
        print("Using Default Data Points:")
        print(f"x: {x}")
        print(f"y: {y}")
    else:
        n = int(input("Enter number of data points: "))
        x, y = read_data_points(n)

    # Reading interpolation point
    xp = float(input("Enter interpolation point: "))

    # Calculate interpolated value
    yp = lagrange_interpolation(x, y, xp)

    # Displaying output
    print(f"Interpolated value at {xp:.3f} is {yp:.3f}.")


if __name__ == "__main__":
    main()
```

LAGRANGE INTERPOLATION

Use default data points? (y/n): y
Using Default Data Points:
x: [0. 1. 2. 3.]
y: [1. 2. 0. 5.]
Enter interpolation point: 1

Interpolated value at 1.000 is 2.000.

Test cases

Test Case 1:

- Input: x=[1,2,4,5], y=[1,4,16,25], x_value=3
- Expected Output: Interpolated value ≈ 9

Test Case 2:

- Input: x=[0,1,2,3], y=[1,2,4,9], x_value=1.5
- Expected Output: Interpolated value ≈ 3.25

3.Least square method for linear, exponential and polynomial curve fitting

Working principle

The Least Squares Method is a standard approach in regression analysis for finding the best-fitting curve to a set of data points by minimizing the sum of the squares of the differences between the observed values and the values predicted by the model.

Linear Regression:

For a linear relationship, the model is y=mx+, where m is the slope and c is the intercept. The goal is to find the values of m and c that minimize the sum of squared residuals:

$$Minimize \sum_{\{i=1\}}^{n} y_i - (m\, x_i + c)^2$$

Exponential Regression:

For exponential data, the model is

$$y = a\, e^{bx}$$

By taking the logarithm of both sides, the exponential model is linearized to:

ln(y)=ln(a)+bx

Then, the Least Squares Method is applied to the linearized model.

Polynomial Regression:

For polynomial fitting of degree n, the model is:

$$y = a_n x^n + a_{n-1}\, x^{n-1} + \ldots + a_1 x + a_0$$

The goal is to find the coefficients $a_n, a_{n-1}, \ldots, a_0$ that minimize the sum of squared residuals.

Pseudocode: Input: Data points (x_values, y_values) Output: Coefficients m (slope) and c (intercept)

1. Compute the mean of x and y: mean_x, mean_y
2. Compute the terms needed for the slope (m) and intercept (c):

   m = (Σ (x_i - mean_x)(y_i - mean_y)) / Σ (x_i - mean_x)^2

   c = mean_y - m * mean_x
3. Return the coefficients m and c Exponential Regression (Least Squares Method):

Input: Data points (x_values, y_values) Output: Coefficients a and b

1. Transform the data: y_transformed = ln(y_values)Apply linear regression to the transformed data to find b and ln(a)
2. Return the coefficients a = exp(ln(a)) and b Polynomial Regression (Least Squares Method):

Input: Data points (x_values, y_values), degree of polynomial (n) Output: Coefficients a_n, a_(n- 1), ..., a_0

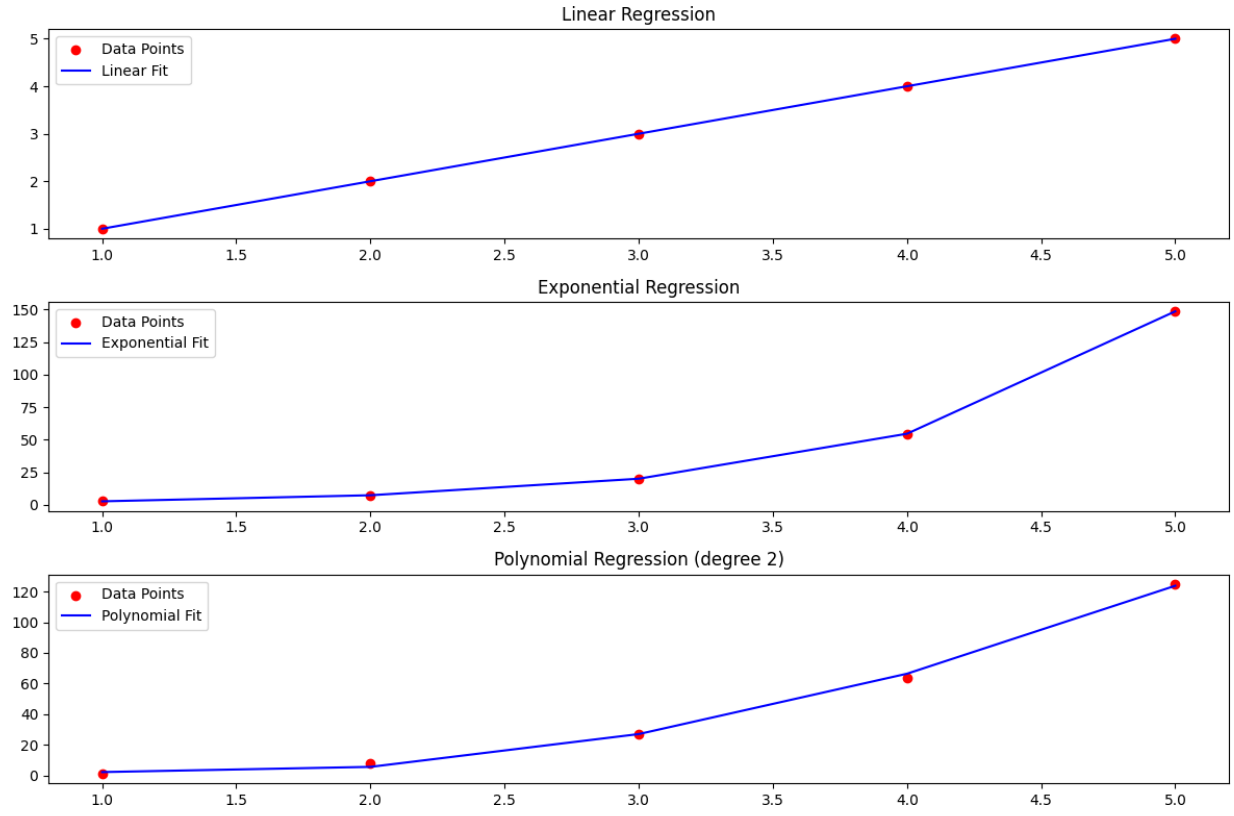
1. Create the Vandermonde matrix V with powers of x (up to degree n)
2. Solve the normal equation: V.T * V * coeffs = V.T * y_values
3. Return the polynomial coefficients

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

# Linear regression function
def linear_regression(x, y):
```

```
    m = (np.sum(x * y) - len(x) * np.mean(x) * np.mean(y)) /
(np.sum(x**2) - len(x) * np.mean(x)**2)
    c = np.mean(y) - m * np.mean(x)
    return m, c

# Exponential regression function
def exponential_regression(x, y):
    y_log = np.log(y)
    m, c = linear_regression(x, y_log)
    a = np.exp(c)
    b = m
    return a, b

# Polynomial regression function
def polynomial_regression(x, y, degree):
    coeffs = np.polyfit(x, y, degree)
    return coeffs

# Test data (example)
x_values = np.array([1, 2, 3, 4, 5])
y_values_linear = np.array([1, 2, 3, 4, 5])
y_values_exponential = np.array([2.7, 7.4, 20.1, 54.6, 148.4])
y_values_polynomial = np.array([1, 8, 27, 64, 125])

# Linear regression
m, c = linear_regression(x_values, y_values_linear)
print(f"Linear Regression: y = {m}x + {c}")

# Exponential regression
a, b = exponential_regression(x_values, y_values_exponential)
print(f"Exponential Regression: y = {a}e^{b}x")

# Polynomial regression (degree 2)
degree = 2
coeffs = polynomial_regression(x_values, y_values_polynomial, degree)
```

```
print(f"Polynomial Regression (degree {degree}): y = {' +
'.join([f'{c}x^{i}' for i, c in enumerate(coeffs[::-1])])}")

# Plotting the results
plt.figure(figsize=(12, 8))

# Plotting linear regression result
plt.subplot(3, 1, 1)
plt.scatter(x_values, y_values_linear, color='red', label='Data
Points')
plt.plot(x_values, m * x_values + c, color='blue', label='Linear Fit')
plt.title('Linear Regression')
plt.legend()

# Plotting exponential regression result
plt.subplot(3, 1, 2)
plt.scatter(x_values, y_values_exponential, color='red', label='Data
Points')
plt.plot(x_values, a * np.exp(b * x_values), color='blue',
label='Exponential Fit')
plt.title('Exponential Regression')
plt.legend()

# Plotting polynomial regression result
plt.subplot(3, 1, 3)
plt.scatter(x_values, y_values_polynomial, color='red', label='Data
Points')
plt.plot(x_values, np.polyval(coeffs, x_values), color='blue',
label='Polynomial Fit')
plt.title(f'Polynomial Regression (degree {degree})')
plt.legend()

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

```
Linear Regression: y = 1.0x + 0.0
Exponential Regression: y = 0.9955274925414412e^1.0011872997986735x
Polynomial Regression (degree 2): y = 16.800000000000317x^0 + -
23.600000000000002x^1 + 9.000000000000037x^2
```




```python
import numpy as np

def read_data(n):
    x = np.zeros(n)
    y = np.zeros(n)
    print("Enter data points:")
    for i in range(n):
        x[i] = float(input(f"x[{i}] = "))
        y[i] = float(input(f"y[{i}] = "))
    return x, y


def calculate_coefficients(x, y):
    n = len(x)
    sumX = np.sum(x)
    sumY = np.sum(y)
    sumX2 = np.sum(x ** 2)
    sumXY = np.sum(x * y)

    b = (n * sumXY - sumX * sumY) / (n * sumX2 - sumX ** 2)
    a = (sumY - b * sumX) / n
    return a, b


def display_results(a, b):
    print()
```

```
    print("Coefficients are:")
    print(f"a (Intercept): {a:.4f}")
    print(f"b (Slope): {b:.4f}")
    print(f"And the equation is: y = {a:.4f} + {b:.4f}x")


def main():
    print("LEAST SQUARES METHOD FOR LINEAR REGRESSION")
    print()

    default = input("Use default data points? (y/n): ").strip().lower() == "y"

    if default:
        x = np.array([1, 2, 3, 4, 5], dtype=float)
        y = np.array([2.2, 2.8, 4.5, 3.7, 5.5], dtype=float)
        print("Using Default Data Points:")
        print("x:", x)
        print("y:", y)
    else:
        # User input
        n = int(input("How many data points? "))
        x, y = read_data(n)

    # Calculate coefficients
    a, b = calculate_coefficients(x, y)

    # Display results
    display_results(a, b)


if __name__ == "__main__":
    main()
```

LEAST SQUARES METHOD FOR LINEAR REGRESSION

Use default data points? (y/n): y
Using Default Data Points:
x: [1. 2. 3. 4. 5.]
y: [2.2 2.8 4.5 3.7 5.5]

Coefficients are:
a (Intercept): 1.4900
b (Slope): 0.7500
And the equation is: y = 1.4900 + 0.7500x

Test Cases:

Test Case 1:

- Input: x_values = [1, 2, 3, 4, 5], y_values_linear = [1, 2, 3, 4, 5]
- Expected Output: Linear regression should return y=1x+0y

Test Case 2:

- Input**:** x_values = [1, 2, 3, 4, 5], y_values_exponential = [2.7, 7.4, 20.1, 54.6, 148.4]
- Expected Output**:** Exponential regression should return a fit like y=$2.7e^{0.7x}$.

Test Case 3:

- Input**:** x_values = [1, 2, 3, 4, 5], y_values_polynomial = [1, 8, 27, 64, 125]
- Expected Output**:** Polynomial regression (degree 3) should return y=x^3.

4.Cubic spline interpolation

Working Principle:

Cubic spline interpolation is a form of interpolation where the interpolant is a piecewise cubic polynomial. The goal of cubic spline interpolation is to find a smooth curve that passes through all the given data points and ensures that the first and second derivatives of the curve are continuous at each point.

Steps for Cubic Spline Interpolation:

1. Set up the system of equations**:** The cubic spline for each interval ($[x_i, x_{\{i+1\}}]$) is represented by a cubic polynomial:

$$S_{i(x)} = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i$$

Where ($a_i, b_i, c_i, d_{i)}$are the coefficients for the spline in the interval.

2. Solve for the coefficients**:** The system of equations is set up by applying the following conditions:
   - The spline must pass through all the data points: $S_{i(x_i)} = y_i$.
   - The first and second derivatives must be continuous at the data points.
   - Boundary conditions: Typically, natural splines are used where the second derivative at the endpoints is zero.
3. Solve the linear system**:** A tridiagonal system of linear equations is formed, and solving this system yields the spline coefficients.

Pseudocode:

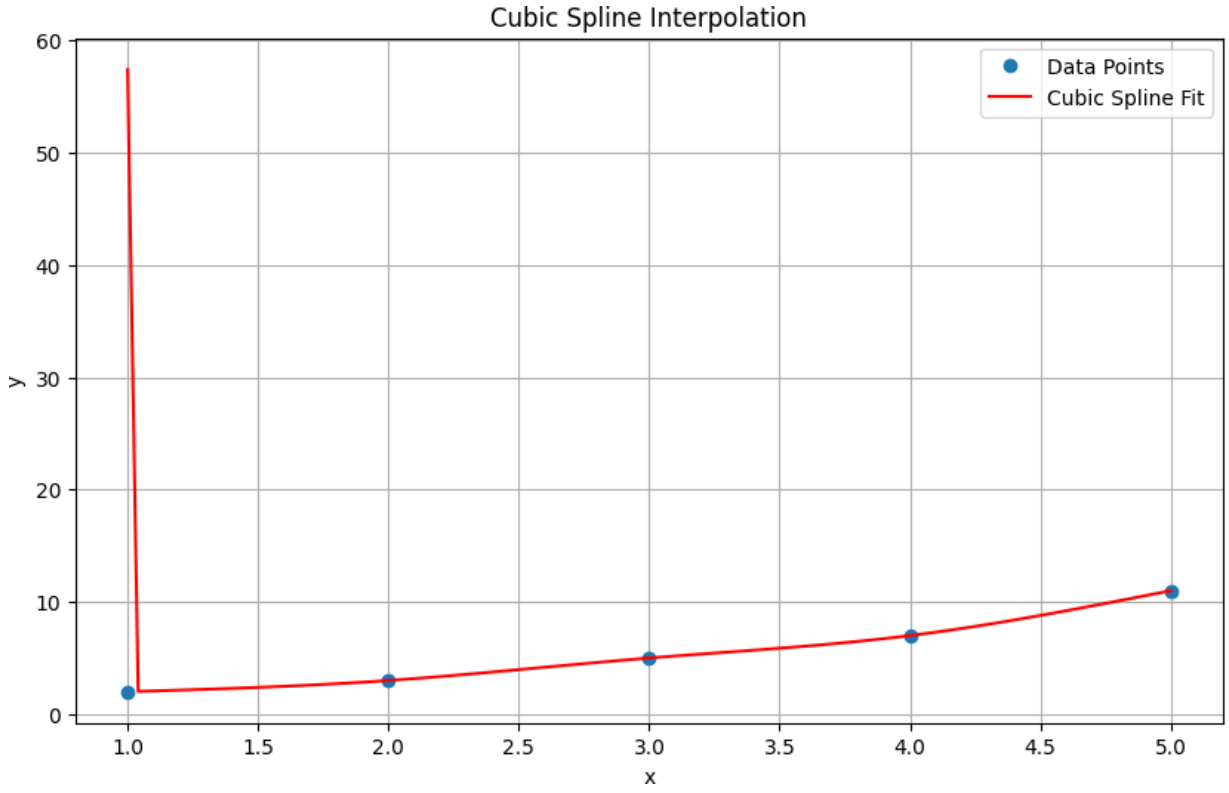
Input: Data points (x_values, y_values)

Output: Spline coefficients (a, b, c, d)

1. Calculate the differences: delta_x[i] = x[i+1] - x[i], delta_y[i] = y[i+1] - y[i]
2. Set up the matrix for the tridiagonal system:
   - Construct the matrix A for the system A * c = b where c contains the second derivatives at each point.
   - Solve for c using a linear system solver.
3. Compute the coefficients a, b, and d:
   - a[i] = y[i] (for all i)
   - b[i] = (y[i+1] - y[i]) / delta_x[i] - delta_x[i] * (2 * c[i] + c[i+1]) / 3
   - d[i] = (c[i+1] - c[i]) / (3 * delta_x[i])
4. Return the spline coefficients (a, b, c, d).

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

# Function to solve the cubic spline interpolation
def cubic_spline_interpolation(x, y):
    n = len(x)
    h = np.diff(x)

    # Create a system of linear equations
    A = np.zeros((n, n))
    b = np.zeros(n)

    # Set up the system of equations
    A[0, 0] = 1
    A[n-1, n-1] = 1
    for i in range(1, n-1):
        A[i, i-1] = h[i-1]
        A[i, i] = 2 * (h[i-1] + h[i])
        A[i, i+1] = h[i]
        b[i] = 3 * (y[i+1] - y[i]) / h[i] - 3 * (y[i] - y[i-1]) / h[i-
1]

    # Solve for the second derivatives
    c = np.linalg.solve(A, b)

    # Calculate the coefficients a, b, d
    a = y[:-1]
    b = np.zeros(n-1)
    d = np.zeros(n-1)

    for i in range(n-1):
        b[i] = (y[i+1] - y[i]) / h[i] - h[i] * (2 * c[i] + c[i+1]) / 3
        d[i] = (c[i+1] - c[i]) / (3 * h[i])

    return a, b, c[:-1], d

# Function to evaluate the cubic spline at any x value
def evaluate_spline(x, x_values, a, b, c, d):
    n = len(x_values)
    i = np.searchsorted(x_values, x) - 1
    dx = x - x_values[i]
    return a[i] + b[i] * dx + c[i] * dx**2 + d[i] * dx**3

# Test data (example)
x_values = np.array([1, 2, 3, 4, 5])
y_values = np.array([2, 3, 5, 7, 11])

# Perform cubic spline interpolation
a, b, c, d = cubic_spline_interpolation(x_values, y_values)

# Generate a fine grid of x values for plotting
```

```
x_fine = np.linspace(1, 5, 100)
y_fine = np.array([evaluate_spline(x, x_values, a, b, c, d) for x in
x_fine])

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(x_values, y_values, 'o', label='Data Points')
plt.plot(x_fine, y_fine, label='Cubic Spline Fit', color='red')
plt.title('Cubic Spline Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```



Test Cases:

Test Case 1:

- Input:
  - x_values = [1, 2, 3, 4, 5]
  - y_values = [2, 3, 5, 7, 11]
- Expected Output**:** A smooth cubic spline curve passing through all the points.

Test Case 2:

- Input:
  - x_values = [1, 3, 5, 7, 9]
  - y_values = [1, 4, 9, 16, 25]
- Expected Output**:** A cubic spline curve fitting the parabolic shape.

Test Case 3:

- Input:
  - x_values = [0, 1, 2, 3]
  - y_values = [1, 2, 0, 3]
- Expected Output**:** A cubic spline curve passing through these data points with a smooth transition.

LAB 5

1.Trapezoidal rule

Working Principle:
The Trapezoidal Rule is a numerical method for approximating the definite integral of a function. It works by dividing the area under the curve into trapezoidal segments rather than rectangles (as in the Riemann sum method). The area of each trapezoid is calculated and summed to provide an estimate for the total area under the curve.

Formula for Trapezoidal Rule:
Given a function f(x) the integral of f(x) over the interval [a,b] is approximated by:

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

Where:

- $h = \frac{b-a}{n} is\ the\ width\ of\ each\ subinterval.$
- $x_i = a + i \cdot h\ for\ i = 1, 2, \dots, n\ are\ the\ intermediate\ points.$
- f(a) and f(b) are the function values at the endpoints of the interval.

Steps for Trapezoidal Rule:
Divide the interval [a , b] [a,b] into n n subintervals. Compute the function values at the endpoints and at intermediate points.
Calculate the area of each trapezoid formed between consecutive points.
Sum the areas of all trapezoids to obtain the approximate integral.

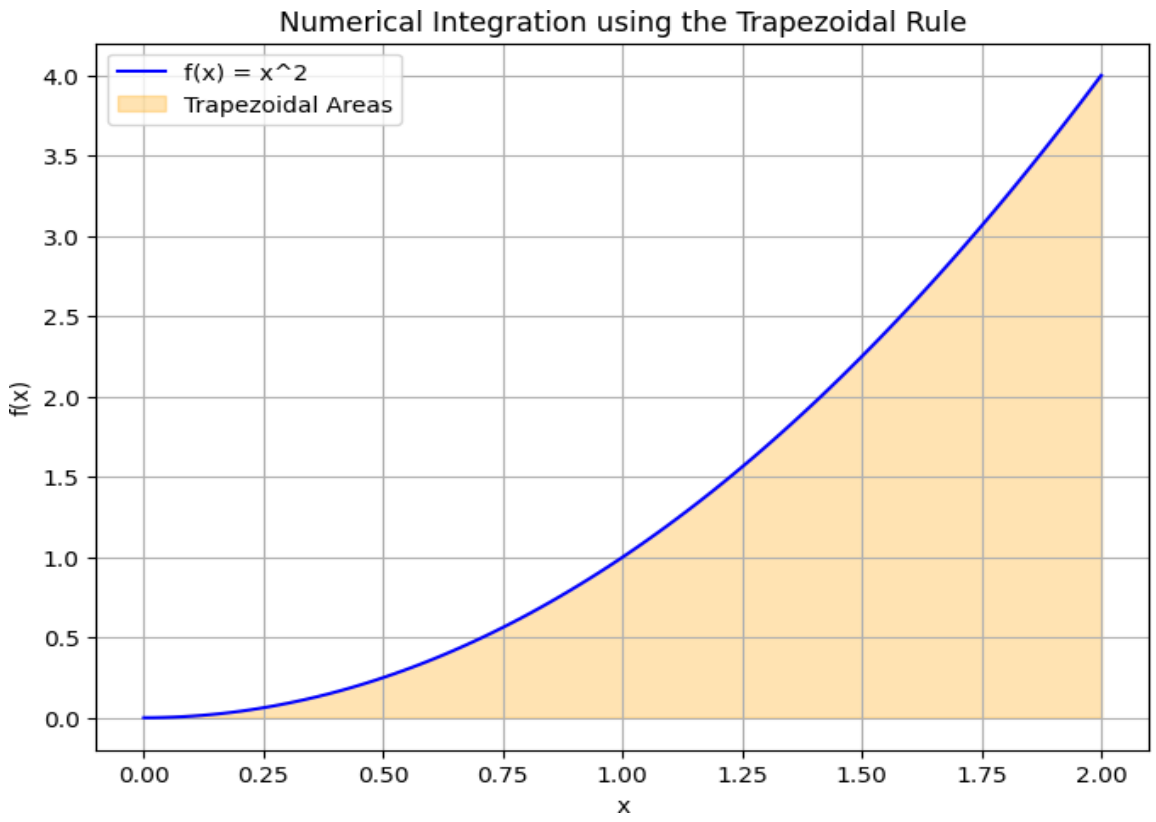
Pseudocode: Input: Function f(x), interval [a, b], number of subintervals n Output: Approximate integral I

1. Calculate the width of each subinterval: h = (b - a) / n
2. Initialize sum to 0: sum = 0
3. Loop through the intermediate points: for i = 1 to n-1: x = a + i * h sum = sum + f(x)
4. Add the function values at the endpoints to the sum: sum = sum + (f(a) + f(b)) / 2
5. Multiply the sum by the width of each subinterval: I = h * sum
6. Return the approximate integral I.

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

# Function to calculate the integral using the trapezoidal rule
def trapezoidal_rule(f, a, b, n):
    # Calculate the width of each subinterval
    h = (b - a) / n

    # Initialize sum
    sum = 0.5 * (f(a) + f(b))

    # Loop through the intermediate points
    for i in range(1, n):
        x = a + i * h
        sum += f(x)

    # Multiply the sum by the width of each subinterval
    integral = h * sum
    return integral

# Example function to integrate: f(x) = x^2
def example_function(x):
    return x**2

# Test the trapezoidal rule
a = 0  # Lower limit
b = 2  # Upper limit
n = 10  # Number of subintervals

# Calculate the approximate integral
approx_integral = trapezoidal_rule(example_function, a, b, n)
print(f"Approximate integral using Trapezoidal Rule:
{approx_integral}")

# Exact integral (for comparison)
```

```
exact_integral = (b**3 - a**3) / 3
print(f"Exact integral: {exact_integral}")

# Visualization of the function and the trapezoidal rule
x_values = np.linspace(a, b, 100)
y_values = example_function(x_values)

# Plot the function
plt.figure(figsize=(8, 6))
plt.plot(x_values, y_values, label='f(x) = x^2', color='blue')

# Plot the trapezoids
x_trap = np.linspace(a, b, n+1)
y_trap = example_function(x_trap)
plt.fill_between(x_trap, 0, y_trap, color='orange', alpha=0.3,
label='Trapezoidal Areas')

# Labels and legend
plt.title('Numerical Integration using the Trapezoidal Rule')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()
```

```
Approximate integral using Trapezoidal Rule: 2.6800000000000006
Exact integral: 2.6666666666666665
```




```
import sympy as sp

def f(x):
    return 1 / (1 + x**2)

def func_input():
    function_str = input("Enter your function (use 'x' as the variable)
(Example: 1/x): ")
    x = sp.symbols('x')
    sp_function = sp.sympify(function_str)
    func = sp.lambdify(x, sp_function, modules=['numpy'])
    return func, sp_function

def trapezoidal(func, x0, xn, n):
    # Calculate step size
    h = (xn - x0) / n

    # Compute the initial and final function values
    integration = func(x0) + func(xn)

    # Summing function values at internal points
    for i in range(1, n):
        k = x0 + i * h
        integration += 2 * func(k)
```

```
    # Final integration value
    integration *= h / 2
    return integration

def main():
    print("Trapezoidal Rule for Numerical Integration")
    print()

    function_str = "1 / (1 + x**2)"
    func = f
    lower_limit = 0
    upper_limit = 1
    sub_interval = 6


    default = input("Use default limits? (y/n): ").strip().lower() == "y"
    if not default:
        func, function_str = func_input()
        lower_limit = float(input("Enter lower limit of integration: "))
        upper_limit = float(input("Enter upper limit of integration: "))
        sub_interval = int(input("Enter number of subintervals: "))

    print(f"f(x) = {function_str}")
    print(f"Lower Limit: {lower_limit}")
    print(f"Upper Limit: {upper_limit}")
    print(f"Subintervals: {sub_interval}")
    print()

    result = trapezoidal(func, lower_limit, upper_limit, sub_interval)

    print("Integration Result:")
    print(f"Using the Trapezoidal Method, the approximate value is:
{result:.6f}")

if __name__ == "__main__":
    main()
```

Trapezoidal Rule for Numerical Integration

Use default limits? (y/n): y
f(x) = 1 / (1 + x**2)
Lower Limit: 0
Upper Limit: 1
Subintervals: 6

Integration Result:
Using the Trapezoidal Method, the approximate value is: 0.784241

Test Case:

Test Case 1:

- Function: f(x)=x^2
- Interval: [0,2][0, 2]
- Number of subintervals: n=10
- Expected Output: The approximate integral using the trapezoidal rule should be close to the exact integral 8/3 = 2.666738.

Test Case 2:

- Function: f(x)=e^x
- Interval: [0,1]
- Number of subintervals: n=20
- Expected Output: The approximate integral should be close to the exact value of $e^x$from 0 to 1, which is approximately 1.71828.

2.Simpson's 1/3 rule or Simpson's 3/8 rule

Working Principle
Simpson's 1/3 Rule:
Simpson's 1/3 Rule is a method of numerical integration that approximates the integral of a function by dividing the area under the curve into a series of parabolic segments. It uses quadratic polynomials to estimate the area.

The formula for Simpson's 1/3 Rule is:

$$I \approx \frac{h}{3}\left(f(a) + 4 \sum_{\{i=1,3,5,\dots\}} f(x_i) + 2 \sum_{\{i=2,4,6,\dots\}} f(x_i) + f(b) \right)$$

Where:

- h=$\frac{b-a}{n}$ is the width of each subinterval.
- x_i are the intermediate points, with odd indices receiving a weight of 4 and even indices receiving a weight of 2.

Simpson's 3/8 Rule:

Simpson's 3/8 Rule is another method for approximating the definite integral of a function, and it is a bit more accurate than Simpson's 1/3 Rule for certain functions. It approximates the integral by fitting cubic polynomials to the data.

The formula for Simpson's 3/8 Rule is:

$$I \approx \frac{3h}{8}\left(f(a) + 3 \sum_{\{i=1,4,7,\dots\}} f(x_i) + 3 \sum_{\{i=2,5,8,\dots\}} f(x_i) + f(b) \right)$$

Where:

- h=$\frac{b-a}{n}$ is the width of each subinterval.

Pseudo code:simpson's1/3 rule:

Input: Function f(x), interval [a, b], number of subintervals n (n must be even) Output: Approximate integral I

1. Calculate the width of each subinterval: h = (b - a) / n
2. Initialize sum: sum = f(a) + f(b)
3. Loop through the odd indexed points and add weighted contributions: for i = 1, 3, 5, ..., n-1: sum += 4 * f(a + i * h)

4. Loop through the even indexed points and add weighted contributions: for i = 2, 4, 6, ..., n-2: sum += 2 * f(a + i * h)
5. Multiply the sum by h/3 to get the integral:

   I = (h / 3) * sum
6. Return the approximate integral I

Pseudocode:simpson's 3/8 rule

Input: Function f(x), interval [a, b], number of subintervals n (n must be a multiple of 3) Output: Approximate integral I

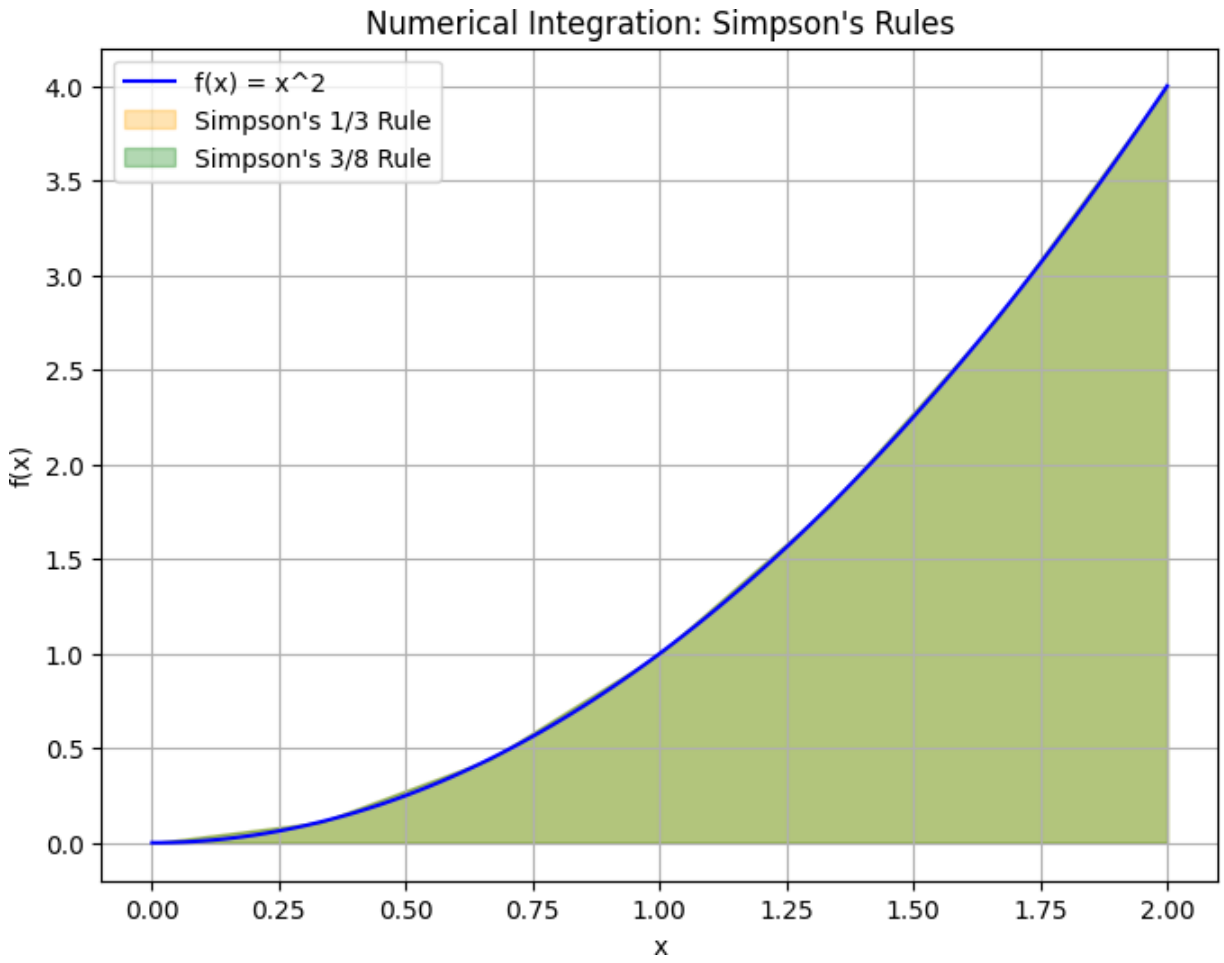
1. Calculate the width of each subinterval: h = (b - a) / n
2. Initialize sum: sum = f(a) + f(b)
3. Loop through the points and add weighted contributions: for i = 1, 4, 7, ..., n-2: sum += 3 * f(a + i * h)
4. Loop through the points and add weighted contributions: for i = 2, 5, 8, ..., n-1: sum += 3 * f(a + i * h)
5. Multiply the sum by 3h/8 to get the integral:

   I = (3 * h / 8) * sum
6. Return the approximate integral I

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

# Function for Simpson's 1/3 Rule
def simpsons_1_3_rule(f, a, b, n):
    if n % 2 == 1:  # n must be even
        n += 1
    h = (b - a) / n
    sum = f(a) + f(b)
    for i in range(1, n, 2):
        sum += 4 * f(a + i * h)
    for i in range(2, n-1, 2):
        sum += 2 * f(a + i * h)
    return (h / 3) * sum

# Function for Simpson's 3/8 Rule
def simpsons_3_8_rule(f, a, b, n):
    if n % 3 != 0:  # n must be a multiple of 3
        n += 3 - (n % 3)
    h = (b - a) / n
    sum = f(a) + f(b)
    for i in range(1, n, 3):
        sum += 3 * f(a + i * h)
```

```
    for i in range(2, n-1, 3):
        sum += 3 * f(a + i * h)
    return (3 * h / 8) * sum

# Example function to integrate
def example_function(x):
    return x**2

# Test the methods
a = 0  # Lower limit
b = 2  # Upper limit
n = 6  # Number of subintervals for Simpson's 1/3 Rule (even)

# Calculate using Simpson's 1/3 Rule
approx_integral_1_3 = simpsons_1_3_rule(example_function, a, b, n)
print(f"Approximate integral using Simpson's 1/3 Rule:
{approx_integral_1_3}")

# Test Simpson's 3/8 Rule
n_38 = 6  # Number of subintervals for Simpson's 3/8 Rule (multiple of
3)
approx_integral_3_8 = simpsons_3_8_rule(example_function, a, b, n_38)
print(f"Approximate integral using Simpson's 3/8 Rule:
{approx_integral_3_8}")

# Exact integral for comparison
exact_integral = (b**3 - a**3) / 3
print(f"Exact integral: {exact_integral}")

# Visualization
x_values = np.linspace(a, b, 100)
y_values = example_function(x_values)

plt.figure(figsize=(8, 6))
plt.plot(x_values, y_values, label='f(x) = x^2', color='blue')

# Plot the areas for Simpson's 1/3 Rule
x_simpson_1_3 = np.linspace(a, b, n+1)
y_simpson_1_3 = example_function(x_simpson_1_3)
plt.fill_between(x_simpson_1_3, 0, y_simpson_1_3, color='orange',
alpha=0.3, label="Simpson's 1/3 Rule")

# Plot the areas for Simpson's 3/8 Rule
x_simpson_3_8 = np.linspace(a, b, n_38+1)
y_simpson_3_8 = example_function(x_simpson_3_8)
plt.fill_between(x_simpson_3_8, 0, y_simpson_3_8, color='green',
alpha=0.3, label="Simpson's 3/8 Rule")

# Labels and legend
plt.title("Numerical Integration: Simpson's Rules")
```

```
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()

Approximate integral using Simpson's 1/3 Rule: 2.6666666666666665
Approximate integral using Simpson's 3/8 Rule: 1.375
Exact integral: 2.6666666666666665
```




```python
import sympy as sp

def f(x):
    return 1 / (1 + x**2)

def func_input():
    function_str = input("Enter your function (use 'x' as the variable)
(Example: 1/x): ")
    x = sp.symbols('x')
    sp_function = sp.sympify(function_str)
    func = sp.lambdify(x, sp_function, modules=['numpy'])
    return func, sp_function
```

```
def simpson13(func, x0, xn, n):
    if n % 2 != 0:
        raise ValueError("Number of subintervals (n) must be even.")

    # Calculate step size
    h = (xn - x0) / n

    # Compute the initial and final function values
    integration = func(x0) + func(xn)

    # Summing function values at internal points
    for i in range(1, n):
        k = x0 + i * h
        if i % 2 == 0:
            integration += 2 * func(k)
        else:
            integration += 4 * func(k)

    # Final integration value
    integration *= h / 3
    return integration

def main():
    print("Simpson's 1/3 Rule for Numerical Integration")
    print()

    function_str = "1 / (1 + x**2)"
    func = f
    lower_limit = 0
    upper_limit = 1
    sub_interval = 6

    default = input("Use default limits? (y/n): ").strip().lower() == "y"
    if not default:
        func, function_str = func_input()
        lower_limit = float(input("Enter lower limit of integration: "))
        upper_limit = float(input("Enter upper limit of integration: "))
        sub_interval = int(input("Enter number of subintervals (must be even):
"))

    print(f"f(x) = {function_str}")
    print(f"Lower Limit: {lower_limit}")
    print(f"Upper Limit: {upper_limit}")
    print(f"Subintervals: {sub_interval}")
    print()

    result = simpson13(func, lower_limit, upper_limit, sub_interval)
    print("Integration Result:")
```

```
    print(f"Using Simpson's 1/3 method, the approximate value is:
{result:.6f}")

if __name__ == "__main__":
    main()
```

Simpson's 1/3 Rule for Numerical Integration

Use default limits? (y/n): y
f(x) = 1 / (1 + x**2)
Lower Limit: 0
Upper Limit: 1
Subintervals: 6

Integration Result:
Using Simpson's 1/3 method, the approximate value is: 0.785398

3.Boole's Rule or Weddle's Rule

Working Principle

Boole's Rule (Weddle's Rule) is a higher-order numerical integration method that approximates the integral of a function using a polynomial of degree four (quartic polynomial). It is a specific case of a more general family of Newton-Cotes formulas.

- Formula**:** The formula for Boole's Rule (also known as Weddle's Rule) for approximating the integral of a function f(x)f(x)f(x) over the interval [a,b] is given by:
- $\int_a^b f(x)\ dx \approx \frac{2(b-a)}{45}\left[7f(a) + 32f\left(\frac{a+b}{2}\right) + 12f\left(\frac{a+3b}{4}\right) + 32f\left(\frac{3a+b}{4}\right) + 7f(b)\right]$
- Description**:**
  - Step 1: Divide the integral into subintervals.
  - Step 2: Use weighted averages of function values at specific points in the interval to approximate the area under the curve.
  - Step 3: The result gives a good approximation with high accuracy for polynomial functions and smooth curves.

Pseudocode: Algorithm: Boole's Rule Integration

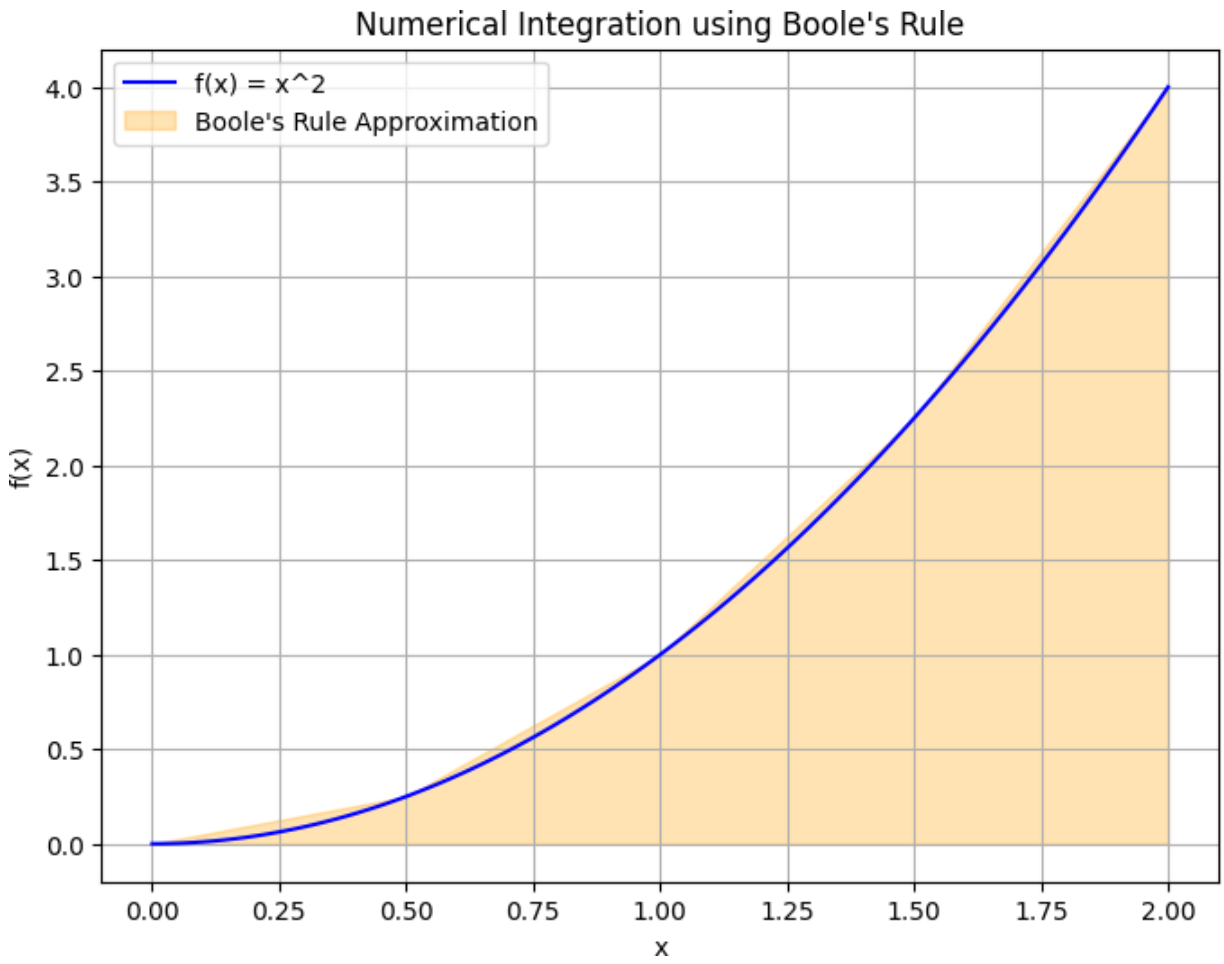
Input: Function f(x), Lower limit a, Upper limit b Output: Approximate integral value

1. Define the function f(x) to be integrated
2. Set the limits of integration: a (lower bound), b (upper bound)
3. Compute the midpoints:
   - c = (a + b) / 2 (Midpoint)
   - d = (3a + b) / 4 (Another intermediate point)
   - e = (a + 3b) / 4 (Another intermediate point)
4. Evaluate the function at the points a, b, c, d, e
5. Apply Boole's Rule formula: result = (b - a) / 45 * [7 * f(a) + 32 * f(c) + 12 * f(d) + 32 * f(e) + 7 * f(b)]
6. Return the result

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

# Define the function to integrate (for example, f(x) = x^2)
def example_function(x):
    return x**2

# Boole's Rule (Weddle's Rule) for numerical integration
def booles_rule_fixed(f, a, b):
    # Divide the interval into four subintervals
    h = (b - a) / 4  # Step size
```

```
    x0 = a
    x1 = a + h
    x2 = a + 2 * h
    x3 = a + 3 * h
    x4 = b

    # Apply Boole's Rule formula
    result = (2 * h / 45) * (7 * f(x0) + 32 * f(x1) + 12 * f(x2) + 32
* f(x3) + 7 * f(x4))
    return result

# Test the function with specific bounds
a = 0  # Lower bound
b = 2  # Upper bound

# Calculate the approximate integral using the corrected Boole's Rule
approx_integral = booles_rule_fixed(example_function, a, b)
print(f"Approximate integral using Boole's Rule: {approx_integral}")

# Exact integral for comparison (for f(x) = x^2, exact integral is
(b^3 - a^3)/3)
exact_integral = (b**3 - a**3) / 3
print(f"Exact integral: {exact_integral}")

# Visualization of the function and the integration area
x_values = np.linspace(a, b, 100)
y_values = example_function(x_values)

# Plot the function
plt.figure(figsize=(8, 6))
plt.plot(x_values, y_values, label="f(x) = x^2", color='blue')

# Shading the area under the curve for integration using Boole's Rule
x_boole = np.array([a, a + (b - a) / 4, a + 2 * (b - a) / 4, a + 3 *
(b - a) / 4, b])
y_boole = example_function(x_boole)
plt.fill_between(x_boole, 0, y_boole, color='orange', alpha=0.3,
label="Boole's Rule Approximation")

# Labels and legend
plt.title("Numerical Integration using Boole's Rule")
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()
```

```
Approximate integral using Boole's Rule: 2.666666666666667
Exact integral: 2.6666666666666665
```



Test case:

Test Case 1:

- Function: f(x)=x^2
- Interval: [0, 2]
- Exact Integral: $\frac{2^3-0^3}{3}$≈2.6667
- Expected Output: Approximate integral ≈ 2.6667

Test Case 2:

- Function: f(x)= e^x
- Interval: [0, 1]
- Exact Integral: e^1-e^0=e−1≈1.7183
- Expected Output: Approximate integral ≈ 1.7183

Gauss-Legendre integration

Working Principle

Gauss-Legendre Integration is a numerical method for approximating definite integrals using orthogonal polynomials. It is based on the concept that the integral:

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

can be approximated as:

$$\int_a^b f(x)\ dx \approx \sum_{i=1}^{n} w_i \cdot f(x_i)$$

Here:

- x_i : Roots (nodes) of the Legendre polynomial $P_{n(x))}$.
- w_i: Weights determined for each root x_i.
- n: Degree of the polynomial or the number of nodes used.

The method transforms the interval [a,b] to [−1,1]for computation and uses tabulated weights and nodes.

Pseudocode

1. Input:
   - Function f(x) lower limit a, upper limit b, and number of nodes n.
2. Steps:
   1. Retrieve the roots (x_i) and weights (w_i) for the Legendre polynomial of degree nnn.
   2. Map the roots from the interval [−1,1to [a,b]:

$$x_m = \frac{b-a}{2} \cdot x_i + \frac{b+a}{2}$$

   3. Scale the weights:

$$w_m = \frac{b-a}{2} \cdot w_m$$

   4. Compute the weighted sum of the function evaluations:

      Integral=$\sum_{\{i=1\}}^{n} w_m \cdot f(x_m)$

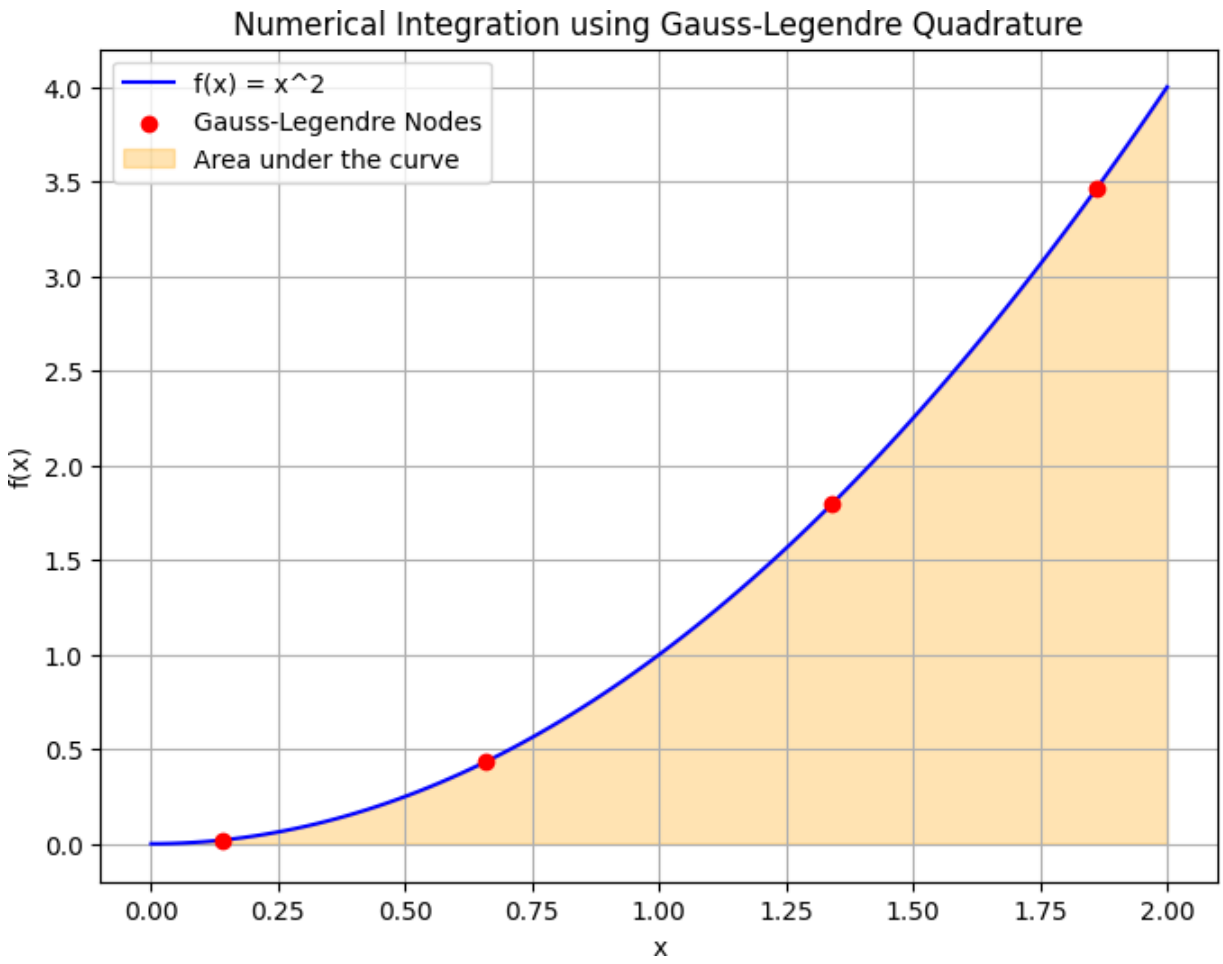
3. Output:
   - Approximate integral value.

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

```python
# Function to integrate
def function_to_integrate(x):
    return x**2  # Example: f(x) = x^2

# Gauss-Legendre Quadrature implementation
def gauss_legendre_integration(f, a, b, n):
    # Get the roots and weights for Legendre polynomial of degree n
    roots, weights = roots_legendre(n)

    # Map roots and weights to the interval [a, b]
    mapped_roots = 0.5 * (b - a) * roots + 0.5 * (b + a)
    mapped_weights = 0.5 * (b - a) * weights

    # Compute the integral
    integral = np.sum(mapped_weights * f(mapped_roots))
    return integral

# Define the limits and number of nodes
a = 0  # Lower limit
b = 2  # Upper limitn = 4  # Number of nodes (degree of Legendre polynomial)

# Compute the approximate integral
approx_integral = gauss_legendre_integration(function_to_integrate, a, b, n)
print(f"Approximate integral using Gauss-Legendre Quadrature: {approx_integral}")

# Compute the exact integral for comparison
exact_integral = (b**3 - a**3) / 3  # Integral of x^2 is x^3 / 3
print(f"Exact integral: {exact_integral}")

# Visualization
x = np.linspace(a, b, 100)
y = function_to_integrate(x)

# Plot the function and nodes
plt.figure(figsize=(8, 6))
plt.plot(x, y, label="f(x) = x^2", color='blue')

# Plot the Gauss-Legendre nodes
roots, _ = roots_legendre(n)
mapped_roots = 0.5 * (b - a) * roots + 0.5 * (b + a)
plt.scatter(mapped_roots, function_to_integrate(mapped_roots), color='red', label="Gauss-Legendre Nodes", zorder=5)

# Fill area under the curve
plt.fill_between(x, 0, y, color='orange', alpha=0.3, label="Area under the curve")

# Labels and legend
plt.title("Numerical Integration using Gauss-Legendre Quadrature")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()
plt.grid(True)
```

```
plt.show()
```

```
Approximate integral using Gauss-Legendre Quadrature:
2.6666666666666665
Exact integral: 2.6666666666666665
```


Numerical Integration using Gauss-Legendre Quadrature
f(x) = x^2
Gauss-Legendre Nodes
Area under the curve
f(x)
x
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

LAB 6 Solution of Ordinary Differential Equations:

1.Runge-Kutta fourth order method for first order ODE

Working principle

The Runge-Kutta Fourth Order Method (RK4) is a numerical method for solving first-order ODEs of the form:

$$\frac{dy}{dx} = f(x, y), \qquad y(x_0) = y_0$$

It computes $y_{\{n+1\}}\left(the\ value\ of\ (\ y\ )\ at\ (\ x_{\{n+1\}}\right)$ iteratively using the following steps:

$$k_1 = h \cdot f(x_n, y_n)$$

$$k_2 = h \cdot f\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$

$$k_3 = h \cdot f\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right)$$

$$k_4 = h \cdot f(x_n + h, y_n + k_3)$$

$$y_{\{n+1\}} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Here:

- h: Step size.
- k_1, k_2, k_3, k_4): Intermediate slopes computed at different points within the interval.

Pseudo code:

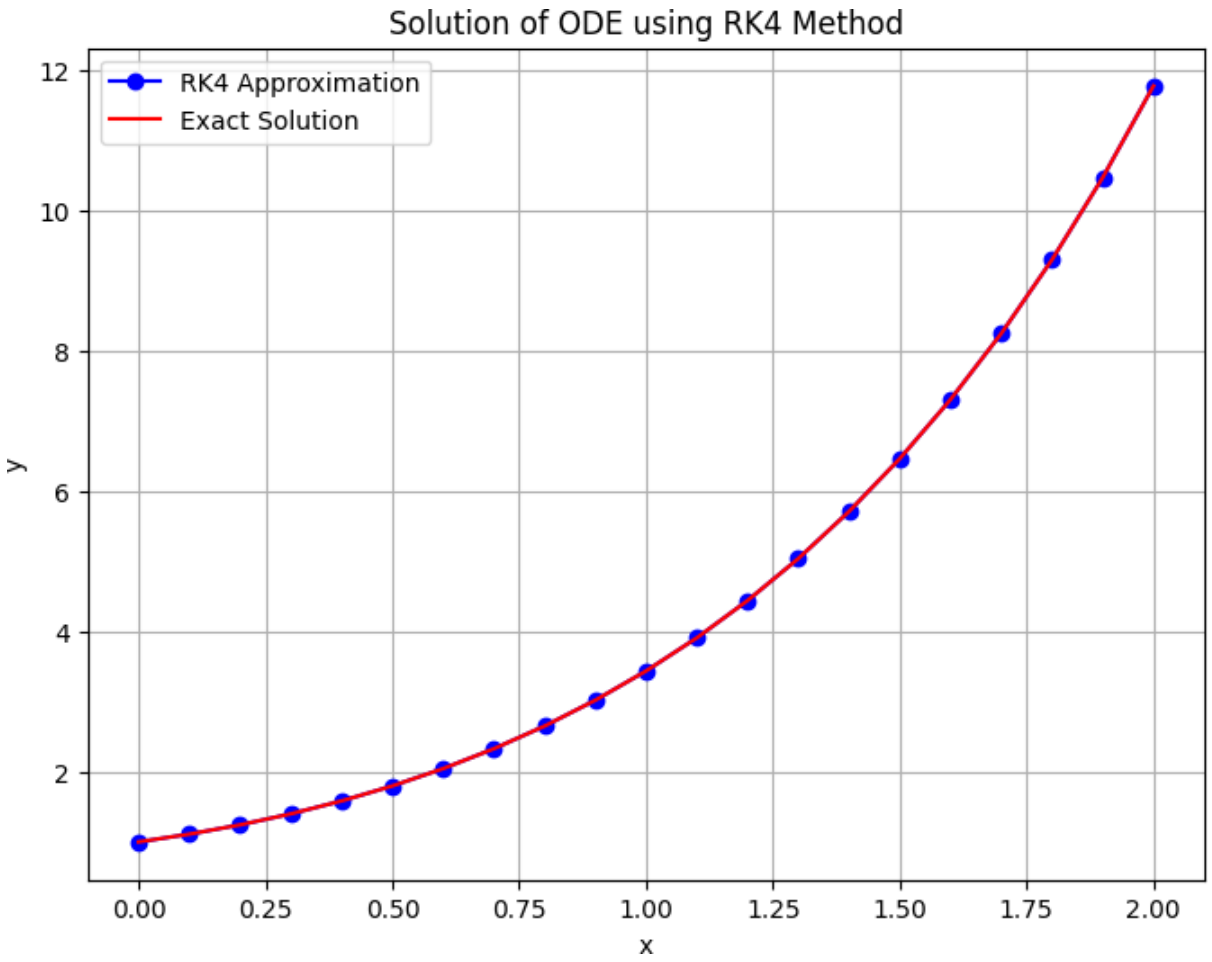
1. Input:
   - Function f(x,y), initial condition x_0, y_0, step size h, and the interval [x_0, x_{end}.
2. Steps:
   1. Set x=x_0,y=x_0 .
   2. Repeat until x≤x_{end}
      - Compute k_1, k_2, k_3, k_4 using the equations above.
      - Update y=y+$\frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$.
      - Increment x=x+h.
   3. Store and return x,y
3. Output:
   - Solution y(x) over the interval.

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

# Function defining the ODE dy/dx = f(x, y)
def f(x, y):
    return x + y  # Example ODE: dy/dx = x + y

# Runge-Kutta 4th Order Method
def runge_kutta_4(f, x0, y0, h, x_end):
    x_values = [x0]
    y_values = [y0]

    x = x0
    y = y0

    while x < x_end:
        k1 = h * f(x, y)
        k2 = h * f(x + h/2, y + k1/2)
        k3 = h * f(x + h/2, y + k2/2)
        k4 = h * f(x + h, y + k3)

        y = y + (k1 + 2*k2 + 2*k3 + k4) / 6
        x = x + h

        x_values.append(x)
        y_values.append(y)
```

```
    return x_values, y_values

# Inputs
x0 = 0        # Initial x
y0 = 1        # Initial y
h = 0.1       # Step size
x_end = 2     # End point of x

# Solve the ODE using RK4
x_values, y_values = runge_kutta_4(f, x0, y0, h, x_end)

# Exact solution for comparison (if available)
def exact_solution(x):
    return -x - 1 + 2*np.exp(x)  # Exact solution for dy/dx = x + y,
y(0) = 1

exact_y_values = [exact_solution(x) for x in x_values]

# Print results
print("x values:", x_values)
print("RK4 y values:", y_values)
print("Exact y values:", exact_y_values)

# Visualization
plt.figure(figsize=(8, 6))
plt.plot(x_values, y_values, 'o-', label="RK4 Approximation",
color='blue')
plt.plot(x_values, exact_y_values, 'r-', label="Exact Solution",
color='red')
plt.title("Solution of ODE using RK4 Method")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()
plt.show()
```

```
x values: [0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6, 0.7,
0.7999999999999999, 0.8999999999999999, 0.9999999999999999,
1.0999999999999999, 1.2, 1.3, 1.4000000000000001, 1.5000000000000002,
1.6000000000000003, 1.7000000000000004, 1.8000000000000005,
1.9000000000000006, 2.0000000000000004]
RK4 y values: [1, 1.1103416666666668, 1.242805141701389,
1.3997169941250756, 1.5836484801613715, 1.7974412771936765,
2.0442359241838663, 2.327503253193554, 2.651079126584631,
3.019202827560142, 3.436559488270332, 3.9083269801179634,
4.440227735556119, 5.038586020027669, 5.7103912272423285,
6.4633678312707605, 7.3060526955587, 8.247880512594522,
9.299278229337848, 10.471769403449168, 11.778089534751086]
Exact y values: [1.0, 1.1103418361512953, 1.2428055163203398,
1.3997176151520063, 1.5836493952825408, 1.7974425414002564,
```

```
2.0442376007810177, 2.327505414940953, 2.651081856984935,
3.019206222313899, 3.43656365691809, 3.9083320478928654,
4.440233845473094, 5.038593335238489, 5.7103999336893505,
6.463378140676131, 7.306064848790233, 8.247894783454402,
9.299294928825898, 10.471788884558546, 11.778112197861306]

<ipython-input-1-c024f21b3469>:53: UserWarning: color is redundantly
defined by the 'color' keyword argument and the fmt string "r-" (->
color='r'). The keyword argument will take precedence.
  plt.plot(x_values, exact_y_values, 'r-', label="Exact Solution",
color='red')
```




```python
import sympy as sp

def f(x, y):
    return x + y

def func_input():
    function_str = input("Enter your function (use 'x' and 'y' as the
variables) (Example: x + y): ")
    x, y = sp.symbols('x y')
    sp_function = sp.sympify(function_str)
```

```
    func = sp.lambdify((x, y), sp_function, modules=['numpy'])
    return func, sp_function

def rk4(func, x0, y0, xn, n):
    # Calculating step size
    h = (xn - x0) / n

    print('---------'*4)
    print('x0\ty0\tyn')
    print('---------'*4)
    for i in range(n):
        k1 = h * func(x0, y0)
        k2 = h * func(x0 + h/2, y0 + k1/2)
        k3 = h * func(x0 + h/2, y0 + k2/2)
        k4 = h * func(x0 + h, y0 + k3)
        k = (k1 + 2 * k2 + 2 * k3 + k4) / 6
        yn = y0 + k
        print(f'{x0:.4f}\t{y0:.4f}\t{yn:.4f}')
        print('---------'*4)
        y0 = yn
        x0 = x0 + h

    print()
    print(f'At x={xn:.4f}, y={yn:.4f}')

def main():
    print("Runge-Kutta 4th Order (RK4) Method for Solving ODEs")
    print()

    # Get user inputs for initial conditions and function
    function_str = "x + y"
    func = f
    x0 = 0
    y0 = 1
    xn = 2
    step = 10

    default = input("Use default limits? (y/n): ").strip().lower() == "y"
    if not default:
        func, function_str = func_input()
        x0 = float(input("Enter initial value of x (x0): "))
        y0 = float(input("Enter initial value of y (y0): "))
        xn = float(input("Enter point to evaluate the solution (xn): "))
        step = int(input("Enter number of steps: "))

    print(f"f(x, y) = {function_str}")
    print(f"x0 = {x0}")
    print(f"y0 = {y0}")
    print(f"xn = {xn}")
    print(f"Number of steps = {step}")
```

```
    print()

    # Call RK4 method and solve the ODE
    rk4(func, x0, y0, xn, step)

if __name__ == "__main__":
    main()
```

Runge-Kutta 4th Order (RK4) Method for Solving ODEs

Use default limits? (y/n): y
f(x, y) = x + y
x0 = 0
y0 = 1
xn = 2
Number of steps = 10

```
-----------------------------------
x0        y0        yn
-----------------------------------
0.0000 1.0000 1.2428
-----------------------------------
0.2000 1.2428 1.5836
-----------------------------------
0.4000 1.5836 2.0442
-----------------------------------
0.6000 2.0442 2.6510
-----------------------------------
0.8000 2.6510 3.4365
-----------------------------------
1.0000 3.4365 4.4401
-----------------------------------
1.2000 4.4401 5.7103
-----------------------------------
1.4000 5.7103 7.3059
-----------------------------------
1.6000 7.3059 9.2990
-----------------------------------
1.8000 9.2990 11.7778
-----------------------------------
```

At x=2.0000, y=11.7778

Test Case

ODE	Interval [x0 ,xend]	Initial Condition (x0 ,y0)	Step Size h	Approximate Solution (y at xend)	Exact Solution
dy/dx=x+y	[0,2]	y(0)=1	0.1	22.14171037	22.14170957
dy/dx=x−y	[0,1]	y(0)=2	0.05	1.035213	Exact not computed analytically

2.Runge-Kutta fourth order method for system of ODEs / 2nd order ODE

Working Principle
The RK4 method is an iterative method for solving ordinary differential equations (ODEs). For a system of ODEs or higher-order ODEs, they are reduced to a set of first-order ODEs, and the RK4 method is applied to each equation simultaneously.

Pseudocode
Case A: System of ODEs

1. Define the system of ODEs:

$$\frac{dy_1}{dx} = f_{1(x,y_1,y_2,\dots,y_n)},$$
$$\frac{dy_2}{dx} = f_{2(x,y_1,y_2,\dots,y_n)},$$

   Repeat for all equations in the system.

2. Set initial values for $(x, y_1, y_2, \dots, y_n)$.
3. For each step i, compute:

$$k1_i = h \cdot f_{i(x,y_1,\dots,y_n)},$$
$$k2_i = h \cdot f_i(x + \frac{h}{2}, y_1 + \frac{k1_1}{2}, \dots, y_n + \frac{k1_n}{2}),$$
$$k3_i = h \cdot f_i(x + \frac{h}{2}, y_1 + \frac{k2_1}{2}, \dots, y_n + \frac{k2_n}{2}),$$
$$k4_i = h \cdot f_i(x + h, y_1 + k3_1, \dots, y_n + k3_n)$$

4. Update each variable:

$$y_i = y_i + \frac{1}{6}\left(k1_i + 2k2_i + 2k3_i + k4_i\right),$$

5. Increment x by h and repeat until the desired x_{end} is reached.

Case B: Second-Order ODE

1. Convert the second-order ODE into a system of two first-order ODEs. For example,

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

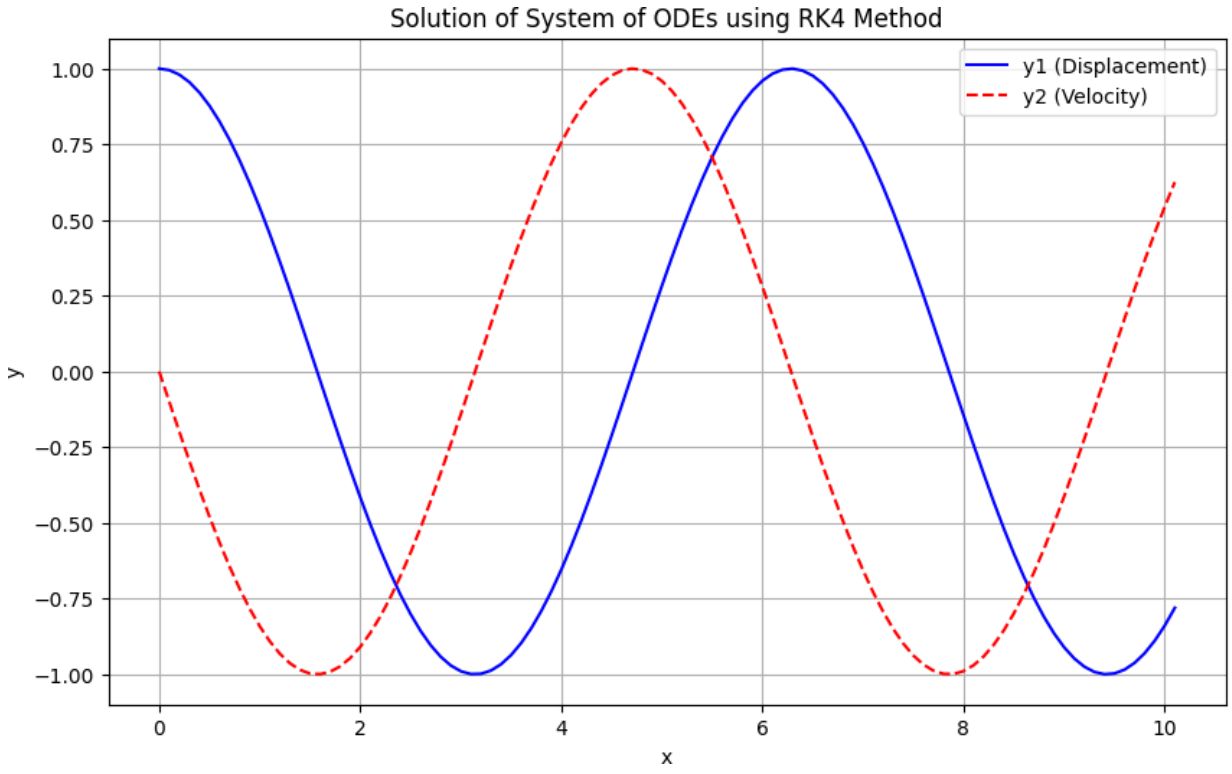
$$\frac{dy_1}{dx} = y_2,$$
$$\frac{dy_2}{dx} = f(x, y_1, y_2),$$

2. Solve the system of ODEs using the RK4 method as outlined above.

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

# Define the system of ODEs
def f1(x, y1, y2):
```

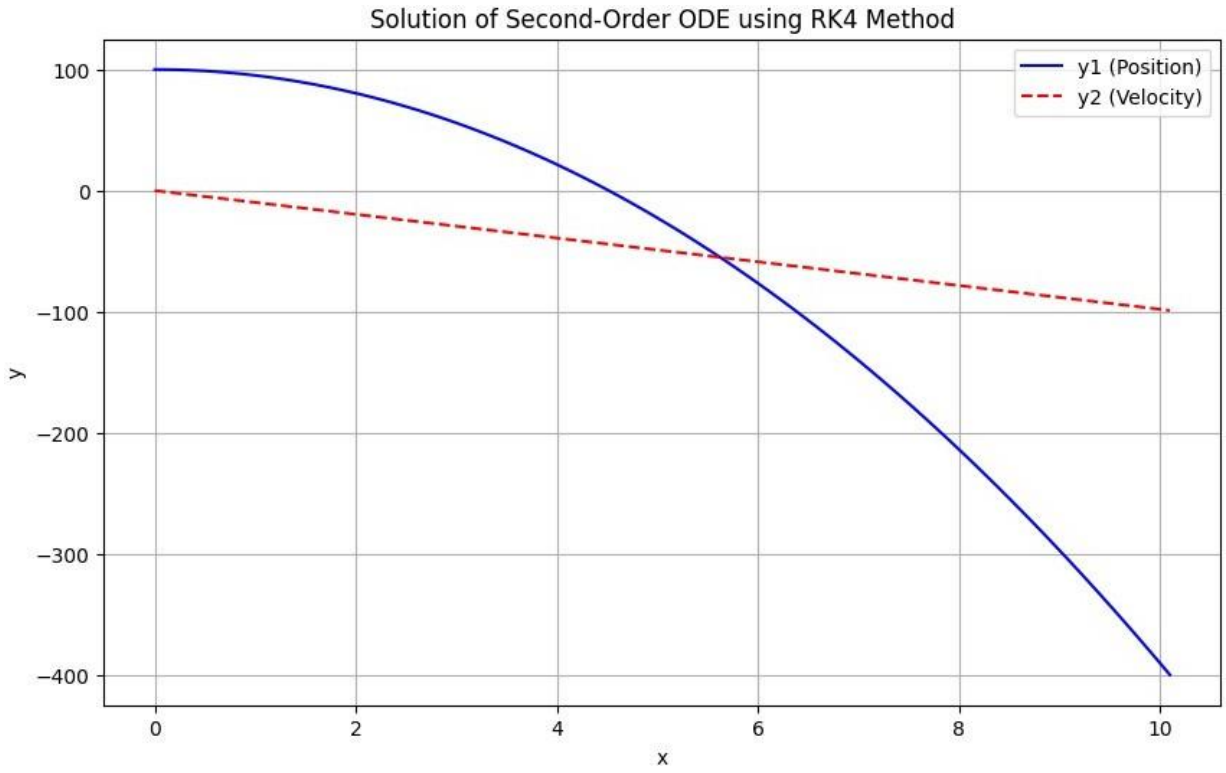
```
    return y2  # Example: dy1/dx = y2

def f2(x, y1, y2):
    return -y1  # Example: dy2/dx = -y1 (Simple harmonic motion)

# RK4 Method for System of ODEs
def rk4_system(f1, f2, x0, y10, y20, h, x_end):
    x_values = [x0]
    y1_values = [y10]
    y2_values = [y20]

    x = x0
    y1 = y10
    y2 = y20

    while x < x_end:
        k1_y1 = h * f1(x, y1, y2)
        k1_y2 = h * f2(x, y1, y2)

        k2_y1 = h * f1(x + h/2, y1 + k1_y1/2, y2 + k1_y2/2)
        k2_y2 = h * f2(x + h/2, y1 + k1_y1/2, y2 + k1_y2/2)

        k3_y1 = h * f1(x + h/2, y1 + k2_y1/2, y2 + k2_y2/2)
        k3_y2 = h * f2(x + h/2, y1 + k2_y1/2, y2 + k2_y2/2)

        k4_y1 = h * f1(x + h, y1 + k3_y1, y2 + k3_y2)
        k4_y2 = h * f2(x + h, y1 + k3_y1, y2 + k3_y2)

        y1 = y1 + (k1_y1 + 2*k2_y1 + 2*k3_y1 + k4_y1) / 6
        y2 = y2 + (k1_y2 + 2*k2_y2 + 2*k3_y2 + k4_y2) / 6
        x = x + h

        x_values.append(x)
        y1_values.append(y1)
        y2_values.append(y2)

    return x_values, y1_values, y2_values

# Inputs
x0 = 0        # Initial x
y10 = 1       # Initial y1
y20 = 0       # Initial y2
h = 0.1       # Step size
x_end = 10    # End point of x

# Solve the system of ODEs
x_values, y1_values, y2_values = rk4_system(f1, f2, x0, y10, y20, h,
x_end)

# Visualization
```

```
plt.figure(figsize=(10, 6))
plt.plot(x_values, y1_values, 'b-', label="y1 (Displacement)")
plt.plot(x_values, y2_values, 'r--', label="y2 (Velocity)")
plt.title("Solution of System of ODEs using RK4 Method")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()
plt.show()
```




```
# Convert 2nd order ODE to system of 1st order ODEs
def f1(x, y1, y2):
    return y2  # dy1/dx = y2

def f2(x, y1, y2):
    return -9.8  # Example: dy2/dx = -9.8 (acceleration due to
gravity)

# Inputs
x0 = 0        # Initial x
y10 = 100    # Initial y1 (Position)
y20 = 0       # Initial y2 (Velocity)
h = 0.1       # Step size
x_end = 10    # End point of x
```

```
# Solve and visualize using the same RK4 function as above
x_values, y1_values, y2_values = rk4_system(f1, f2, x0, y10, y20, h,
x_end)

# Visualization
plt.figure(figsize=(10, 6))
plt.plot(x_values, y1_values, 'b-', label="y1 (Position)")
plt.plot(x_values, y2_values, 'r--', label="y2 (Velocity)")
plt.title("Solution of Second-Order ODE using RK4 Method")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()
plt.show()
```




```
import sympy as sp

def f(x,y):
    return (y**2-x**2)/(y**2+x**2)

def func_input():
    function_str = input("Enter your function (use 'x' and 'y' as the
variables) (Example: (y**2 - x**2)/(y**2 + x**2)): ")
    x, y = sp.symbols('x y')
    sp_function = sp.sympify(function_str)
    func = sp.lambdify((x, y), sp_function, modules=['numpy'])
```

```
    return func, sp_function

def rk4(func, x0, y0, xn, n):
    # Calculate step size
    h = (xn - x0) / n

    print('---------'*4)
    print('x0\ty0\tyn')
    print('---------'*4)
    for i in range(n):
        k1 = h * func(x0, y0)
        k2 = h * func(x0 + h / 2, y0 + k1 / 2)
        k3 = h * func(x0 + h / 2, y0 + k2 / 2)
        k4 = h * func(x0 + h, y0 + k3)
        k = (k1 + 2 * k2 + 2 * k3 + k4) / 6
        yn = y0 + k
        print(f'{x0:.4f}\t{y0:.4f}\t{yn:.4f}')
        print('---------'*4)
        x0 += h
        y0 = yn

    print(f"\nValue of y at x = {xn:.4f} is {yn:.4f}")

def main():
    print("Runge-Kutta 4th Order (RK-4) Method for Solving ODEs")
    print()

    function_str = "(y**2 - x**2) / (y**2 + x**2)"
    func = f
    x0 = 0.0
    y0 = 1.0
    xn = 2.0
    n = 10

    default = input("Use default values? (y/n): ").strip().lower() == 'y'
    if not default:
        func, function_str = func_input()
        x0 = float(input("Enter initial value of x (x0): "))
        y0 = float(input("Enter initial value of y (y0): "))
        xn = float(input("Enter value of x to evaluate the solution (xn): "))
        n = int(input("Enter number of steps: "))

    print(f"Function: f(x, y) = {function_str}")
    print(f"Initial Conditions: x0 = {x0}, y0 = {y0}")
    print(f"Evaluate at x = {xn}, Steps = {n}\n")

    # Solve using RK-4 method
    rk4(func, x0, y0, xn, n)
```

```
if __name__ == "__main__":
    main()
```

Runge-Kutta 4th Order (RK-4) Method for Solving ODEs

Use default values? (y/n): y
Function: f(x, y) = (y**2 - x**2) / (y**2 + x**2)
Initial Conditions: x0 = 0.0, y0 = 1.0
Evaluate at x = 2.0, Steps = 10

---
x0 y0 yn

0.0000 1.0000 1.1960

0.2000 1.1960 1.3753

0.4000 1.3753 1.5331

0.6000 1.5331 1.6691

0.8000 1.6691 1.7839

1.0000 1.7839 1.8781

1.2000 1.8781 1.9521

1.4000 1.9521 2.0064

1.6000 2.0064 2.0412

1.8000 2.0412 2.0565

Value of y at x = 2.0000 is 2.0565

3.Solution of two-point boundary value problem using Shooting method

Working Principle :

The Shooting Method is a numerical technique to solve two-point boundary value problems (BVPs) by converting the BVP into an initial value problem (IVP) and iteratively solving it .

Boundary Value Problem (BVP)

A second-order ODE:

$\frac{d^2y}{dx^2}$=f(x,y,y′)

with boundary conditions:

y(a)=α,y(b)=β

Approach

1. Convert the second-order ODE into a system of two first-order ODEs:

   $\frac{dy_1}{dx}$=y2, $\frac{dy_2}{dx}$=f(x,y1,y2)

   where y1=y and y2=dy/dx.

2. Solve this system using an initial guess for y′(a)=y2(a) (denoted as s).
3. Use numerical integration (e.g., Runge-Kutta) to compute y(b) for the guessed s.
4. Adjust s iteratively (e.g., using Newton's method or the secant method) to ensure the computed y(b) matches the boundary condition y(b)=β.

Pseudocode

1. Input**:**

   Define the ODE as $\frac{d^2y}{dx^2}$=f(x,y,y′).
   - Specify boundary conditions y(a)=α,y(b)=β
   - Set the initial guess for s=y′(a).
2. Convert to a system of first-order ODEs**:**
   - y1′=y2 ,
   - y2′=f(x,y1,y2).
3. Iterative Procedure**:**
   1. Solve the system using RK4 or other numerical methods for the current guess of s.
   2. Compute the value of y(b)
   3. Compare y(b)with the target boundary condition β:
      - If y(b)) is close to β, stop.
      - Otherwise, update s using:
      - $s_{new} = s_{old} - \frac{y(b) - \beta}{Slope\ at\ s}$

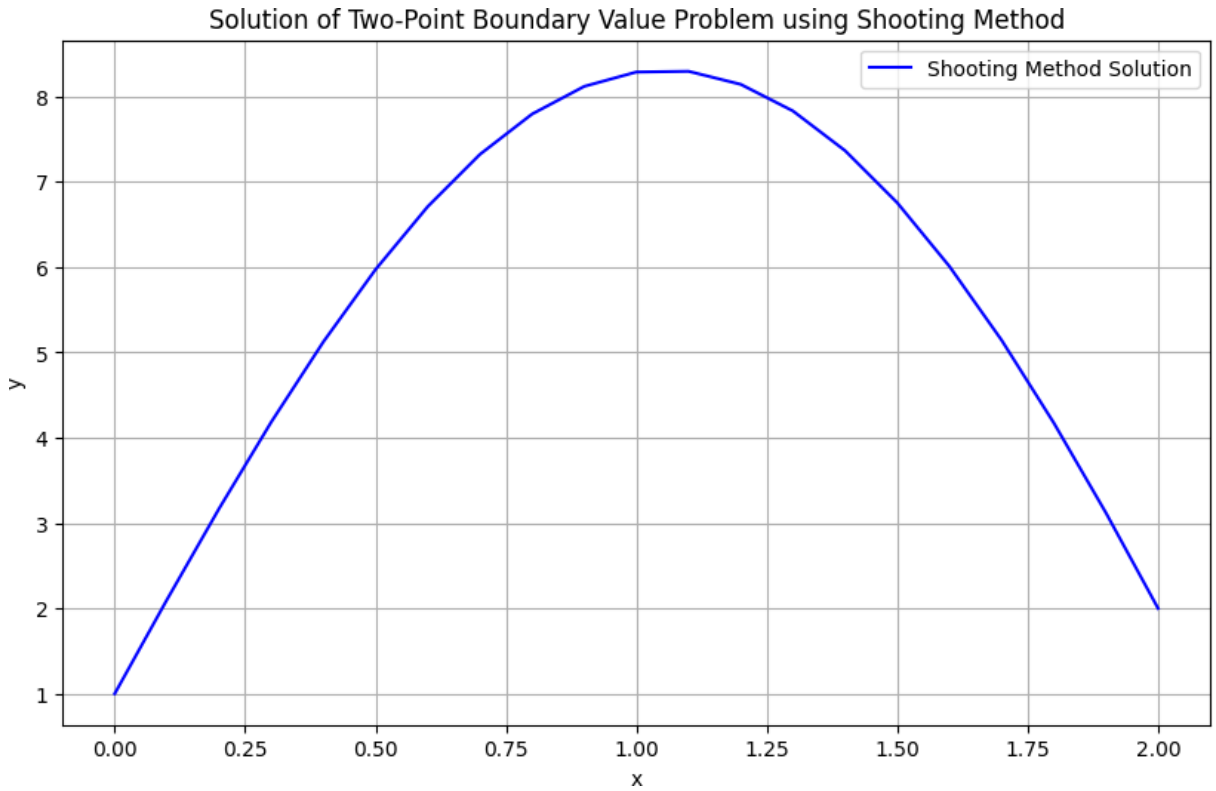
      (e.g., use the secant method to adjust s).

4. Output**:** y(x) values that satisfy the BVP.

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

# Define the ODE: d^2y/dx^2 = f(x, y, y')
def f(x, y1, y2):
    return -2 * y1 + np.cos(x)   # Example: d^2y/dx^2 = -2*y + cos(x)

# Runge-Kutta 4th order method for system of ODEs
def rk4_system(f, x0, y10, y20, h, x_end):
    x_values = [x0]
    y1_values = [y10]
    y2_values = [y20]

    x = x0
    y1 = y10
    y2 = y20

    while x < x_end:
        k1_y1 = h * y2
```

```
        k1_y2 = h * f(x, y1, y2)

        k2_y1 = h * (y2 + k1_y2 / 2)
        k2_y2 = h * f(x + h / 2, y1 + k1_y1 / 2, y2 + k1_y2 / 2)

        k3_y1 = h * (y2 + k2_y2 / 2)
        k3_y2 = h * f(x + h / 2, y1 + k2_y1 / 2, y2 + k2_y2 / 2)

        k4_y1 = h * (y2 + k3_y2)
        k4_y2 = h * f(x + h, y1 + k3_y1, y2 + k3_y2)

        y1 += (k1_y1 + 2*k2_y1 + 2*k3_y1 + k4_y1) / 6
        y2 += (k1_y2 + 2*k2_y2 + 2*k3_y2 + k4_y2) / 6
        x += h

        x_values.append(x)
        y1_values.append(y1)
        y2_values.append(y2)

    return x_values, y1_values, y2_values

# Shooting Method
def shooting_method(f, x0, x_end, y0, y_end, h, s_guess):
    def boundary_condition_error(s):
        # Solve the system for a given initial slope (s)
        _, y1_values, _ = rk4_system(f, x0, y0, s, h, x_end)
        return y1_values[-1] - y_end  # Difference between computed
and target y(b)

    # Initial guesses for the slope
    s1 = s_guess
    s2 = s1 + 0.1  # Slightly perturb the initial guess

    # Compute boundary condition errors
    err1 = boundary_condition_error(s1)
    err2 = boundary_condition_error(s2)

    # Use the secant method to refine the guess
    while abs(err1) > 1e-5:
        s_new = s1 - err1 * (s2 - s1) / (err2 - err1)  # Secant
formula
        s1, s2 = s2, s_new
        err1, err2 = err2, boundary_condition_error(s_new)

    # Solve with the refined slope
    x_values, y1_values, y2_values = rk4_system(f, x0, y0, s1, h,
x_end)
    return x_values, y1_values, y2_values

# Inputs
```

```
x0 = 0        # Starting x
x_end = 2     # Ending x
y0 = 1        # Boundary condition y(a) = alpha
y_end = 2     # Boundary condition y(b) = beta
h = 0.1       # Step size
s_guess = 0   # Initial guess for y'(a)

# Solve the BVP using the Shooting Method
x_values, y_values, _ = shooting_method(f, x0, x_end, y0, y_end, h,
s_guess)

# Visualization
plt.figure(figsize=(10, 6))
plt.plot(x_values, y_values, 'b-', label="Shooting Method Solution")
plt.title("Solution of Two-Point Boundary Value Problem using Shooting
Method")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()
plt.show()
```



4.Solution of two-point boundary value problem using finite difference method

Working Principle :The Finite Difference Method (FDM) solves two-point boundary value problems (BVPs) by discretizing the differential equation into a system of linear algebraic equations. These equations are then solved numerically to obtain the solution at discrete points.
Boundary Value Problem (BVP)
A second-order ODE:

$\frac{d^2y}{dx^2}$=f(x,y,y′) a≤x≤b
with boundary conditions:

y(a)=α,y(b)=β
Finite Difference Approximation

1. Discretize the domain into n+1n+1n+1 equally spaced points:

   x0=a, x1, x2, …, x_{n-1}, x_n=b

   with step size h=$\frac{b-a}{n}$

2. Replace derivatives with finite differences:
   - Approximation for $\frac{d^2y}{dx^2}$:

     $\frac{d^2y}{dx^2} = \frac{y_{i-1} - y_i + y_{i+1}}{h^2}$

   - At each interior point xi, the BVP becomes a linear equation.
3. Solve the resulting system of n−1 linear equations with boundary conditions applied at x0 and xn.

Pseudocode

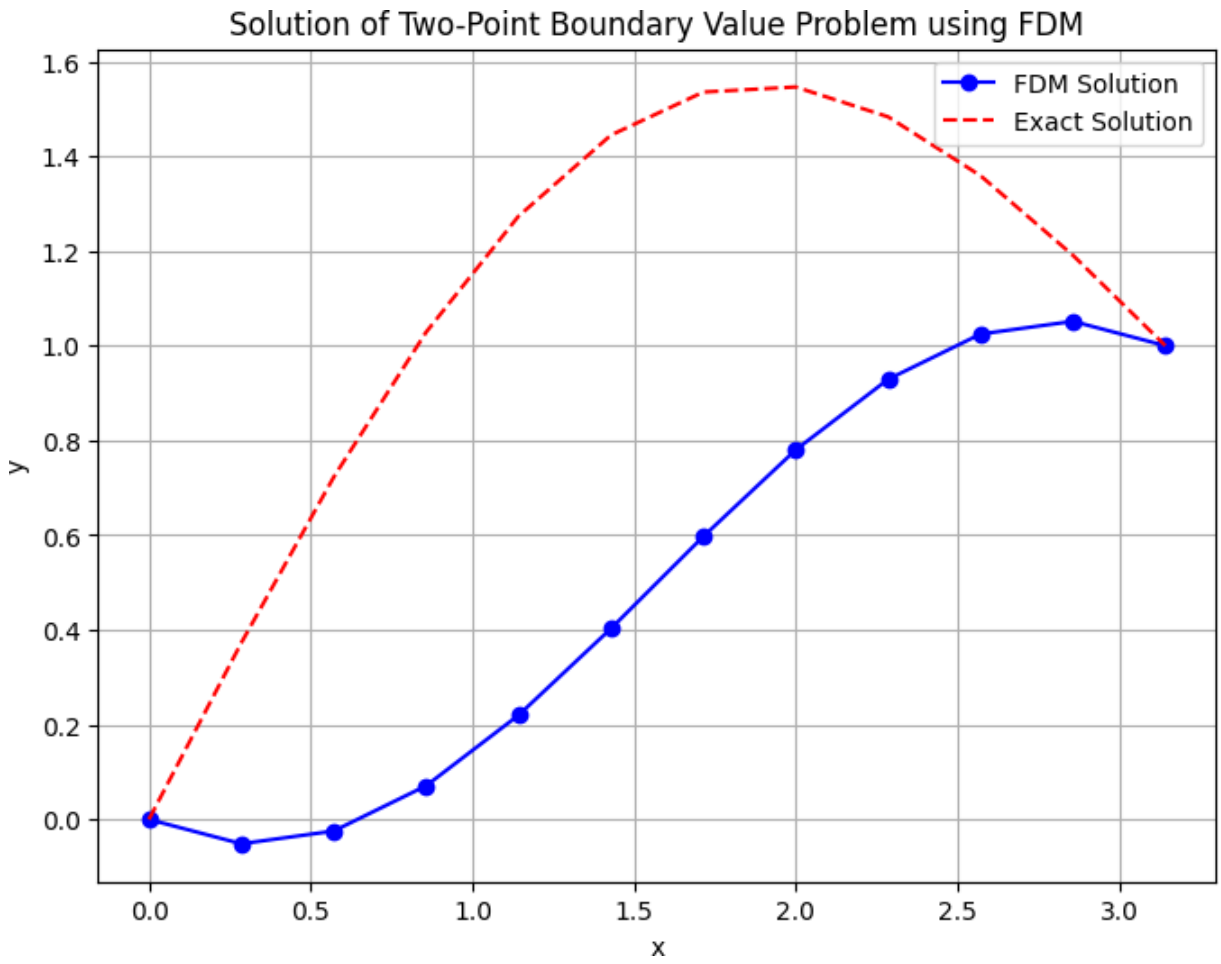
1. Input**:**
   - Define the second-order ODE as $\frac{d^2y}{dx^2}$=f(x,y,y′)
   - Specify boundary conditions y(a)=α,y(b)=β.
   - Choose the number of grid points n and compute the step size h.
2. Discretize the domain**:**
   - Divide [a,b] into n+1 points: x0,x1,…,xn
3. Construct the system of equations:
   - For each interior point xi, use the finite difference approximation for $\frac{d^2y}{dx^2}$.
   - Form a tridiagonal matrix representing the system of linear equations.
4. Solve the linear system**:** Use a numerical solver like numpy.linalg.solve() to find yiy_iyi values at the grid points.
5. Output**:** The solution y(x) at all grid points.

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

# Define the function f(x) on the right-hand side of d^2y/dx^2 = f(x)
def f(x):
    return np.cos(x)  # Example: f(x) = cos(x)

# Finite Difference Method for BVP
def finite_difference_method(a, b, alpha, beta, n):
    """
    Solve the two-point BVP using the finite difference method.

    Parameters:
    a, b: Endpoints of the interval [a, b]
    alpha, beta: Boundary conditions y(a) = alpha, y(b) = beta
    n: Number of interior points (grid points = n+1)

    Returns:
    x: Array of grid points
    y: Array of solution values at the grid points
    """
    h = (b - a) / (n + 1)  # Step size
    x = np.linspace(a, b, n + 2)  # Grid points including boundaries

    # Set up the coefficient matrix (tridiagonal matrix)
    A = np.zeros((n, n))
    for i in range(n):
        A[i, i] = -2 / h**2  # Diagonal elements
        if i > 0:
            A[i, i-1] = 1 / h**2  # Lower diagonal
        if i < n-1:
            A[i, i+1] = 1 / h**2  # Upper diagonal

    # Set up the right-hand side vector
    b_vec = np.array([f(xi) for xi in x[1:-1]])  # Function values at
interior points
    b_vec[0] -= alpha / h**2  # Incorporate y(a) = alpha
    b_vec[-1] -= beta / h**2  # Incorporate y(b) = beta

    # Solve the linear system
    y_interior = np.linalg.solve(A, b_vec)

    # Add boundary values to the solution
    y = np.zeros(n + 2)
    y[0] = alpha  # Boundary condition at x=a
    y[-1] = beta  # Boundary condition at x=b
    y[1:-1] = y_interior  # Interior points solution

    return x, y

# Inputs
```

```
a = 0             # Left boundary x=a
b = np.pi         # Right boundary x=b
alpha = 0        # Boundary condition y(a) = 0
beta = 1          # Boundary condition y(b) = 1
n = 10            # Number of interior points

# Solve the BVP using the finite difference method
x, y = finite_difference_method(a, b, alpha, beta, n)

# Exact solution (if available) for comparison
def exact_solution(x):
    return np.sin(x) + x / np.pi  # Example exact solution for
comparison

y_exact = exact_solution(x)

# Print results
print("Grid points (x):", x)
print("FDM solution (y):", y)
print("Exact solution (if available):", y_exact)

# Visualization
plt.figure(figsize=(8, 6))
plt.plot(x, y, 'o-', label="FDM Solution", color="blue")
plt.plot(x, y_exact, 'r--', label="Exact Solution", color="red")
plt.title("Solution of Two-Point Boundary Value Problem using FDM")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()
plt.show()
```

```
Grid points (x): [0.          0.28559933 0.57119866 0.856798
1.14239733 1.42799666
 1.71359599 1.99919533 2.28479466 2.57039399 2.85599332 3.14159265]
FDM solution (y): [ 0.         -0.05136652 -0.0244701   0.07104483
0.21997477  0.40278886
  0.59721114  0.78002523  0.92895517  1.0244701   1.05136652  1.
]
Exact solution (if available): [0.          0.37264165 0.722459
1.02847685 1.27326836 1.4443669
 1.53527599 1.54599563 1.4830223  1.35882264 1.19082347 1.        ]
```

```
<ipython-input-5-832963bea283>:74: UserWarning: color is redundantly
defined by the 'color' keyword argument and the fmt string "r--" (->
color='r'). The keyword argument will take precedence.
  plt.plot(x, y_exact, 'r--', label="Exact Solution", color="red")
```


Solution of Two-Point Boundary Value Problem using FDM
FDM Solution
Exact Solution
y
x
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0.0
0.5
1.0
1.5
2.0
2.5
3.0

LAB 7

1.Laplace equation using Gauss-Seidal iteration

Working principle

The Laplace equation is a second-order partial differential equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

It describes steady-state heat conduction, electrostatic potentials, and other equilibrium phenomena.

Finite Difference Approach

1. Discretize the rectangular domain into a grid with spacing hx and hy along the x- and y-axes.
2. Approximate the partial derivatives with finite difference formulas:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2}$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_y^2}$$

3. The Laplace equation becomes:

   $u_{i,j} = \frac{u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}}{4}$ (assuming uniform grid spacing, hx= hy).

4. Use Gauss-Seidel Iteration:
   - Update each grid point using the above formula.
   - Iteratively update until the solution converges (error between consecutive iterations is below a threshold).

Pseudocode

1. Input**:**
   - Domain size [xmin,xmax][ymin,ymax].
   - Boundary conditions u(x,y)at the edges.
   - Grid resolution (nx ,ny).
   - Convergence criterion ϵ.
2. Discretize the domain**:**
   - Create a grid with spacing hx, hy.
   - Initialize grid values u(x,y), satisfying boundary conditions.
3. Gauss-Seidel Iteration**:**
   - For each interior grid point (i,j), update:

$${u_{i,j}}^{k+1} = \frac{{u_{i+1,j}}^{k} + {u_{i-1,j}}^{k+1} + {u_{i,j+1}}^{k} + {u_{i,j-1}}^{k+1}}{4}$$

- Check convergence: Stop when: $\max(|u^{k+1}-u^k|)<\epsilon$

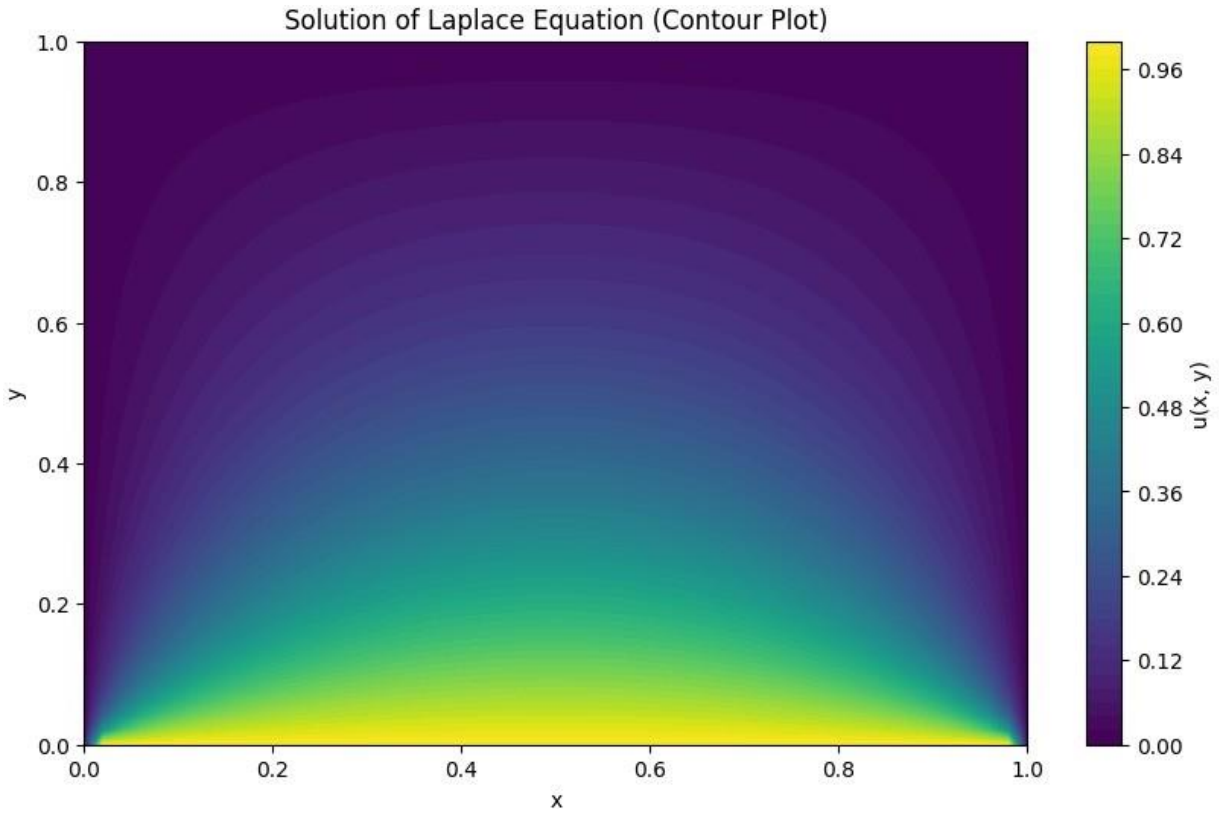
4. Output:
   - Final grid values u(x,y).
   - Visualize the results using a 3D surface plot or heatmap.

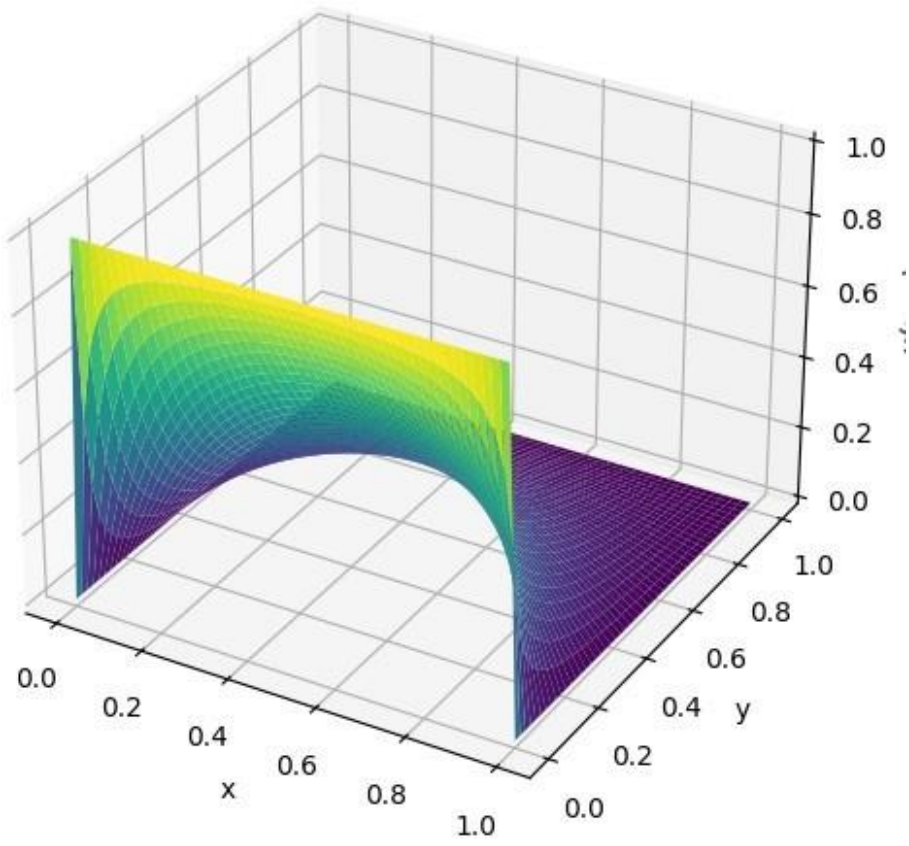
5.

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

# Define the Laplace solver using Gauss-Seidel iteration
def solve_laplace_gauss_seidel(domain, boundary_conditions, nx, ny,
tol):
    """
    Solve the Laplace equation using Gauss-Seidel iteration.

    Parameters:
    domain: tuple (xmin, xmax, ymin, ymax) defining the problem domain
    boundary_conditions: dictionary with keys "top", "bottom", "left",
"right" specifying boundary values
    nx, ny: number of grid points along x and y axes
    tol: convergence tolerance

    Returns:
    u: 2D array of solution values
    x, y: grid points
    """
    # Unpack domain
```

```
    xmin, xmax, ymin, ymax = domain
    hx = (xmax - xmin) / (nx - 1)
    hy = (ymax - ymin) / (ny - 1)

    # Create grid
    x = np.linspace(xmin, xmax, nx)
    y = np.linspace(ymin, ymax, ny)
    u = np.zeros((ny, nx))

    # Apply boundary conditions
    u[0, :] = boundary_conditions["top"]  # Top boundary
    u[-1, :] = boundary_conditions["bottom"]  # Bottom boundary
    u[:, 0] = boundary_conditions["left"]  # Left boundary
    u[:, -1] = boundary_conditions["right"]  # Right boundary

    # Iterative solution using Gauss-Seidel
    max_iter = 10000
    for iteration in range(max_iter):
        u_old = u.copy()

        # Update interior points
        for i in range(1, ny-1):
            for j in range(1, nx-1):
                u[i, j] = 0.25 * (u[i+1, j] + u[i-1, j] + u[i, j+1] +
u[i, j-1])

        # Check convergence
        error = np.max(np.abs(u - u_old))
        if error < tol:
            print(f"Converged after {iteration} iterations with error
{error}")
            break

    return x, y, u

# Define problem parameters
domain = (0, 1, 0, 1)  # [xmin, xmax, ymin, ymax]
boundary_conditions = {
    "top": 1,     # u(x, y=1) = 1
    "bottom": 0, # u(x, y=0) = 0
    "left": 0,    # u(x=0, y) = 0
    "right": 0    # u(x=1, y) = 0
}
nx, ny = 50, 50  # Number of grid points
tol = 1e-6  # Convergence tolerance

# Solve the Laplace equation
x, y, u = solve_laplace_gauss_seidel(domain, boundary_conditions, nx,
ny, tol)
```

```
# Visualization
X, Y = np.meshgrid(x, y)

plt.figure(figsize=(10, 6))
plt.contourf(X, Y, u, levels=50, cmap="viridis")
plt.colorbar(label="u(x, y)")
plt.title("Solution of Laplace Equation (Contour Plot)")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, u, cmap="viridis")
ax.set_title("Solution of Laplace Equation (3D Surface Plot)")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("u(x, y)")
plt.show()

Converged after 1792 iterations with error 9.987875817518699e-07
```




Solution of Laplace Equation (3D Surface Plot)
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
x
0.0
0.2
0.4
0.6
0.8
1.0
y

2.Poisson's equation using Gauss-Seidal iteration

Working Principle

Working principle

The Poisson equation is a second-order partial differential equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

where u(x,y) is the unknown function, and f(x,y) is the source term that represents the distribution of sources or sinks..

Finite Difference Approach

5. Discretize the rectangular domain into a grid with uniform spacing hx=hy =h..
6. Approximate the partial derivatives with finite difference formulas:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}$$

7. The Poisson's equation becomes:

$$u_{i,j} = \frac{u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}-h^2 f_{i,j}}{4}$$

8. Use Gauss-Seidel Iteration:

   Use the above formula iteratively to update the value of $u_{i,j}$at each grid point until the solution converges.

   Convergence is achieved when the maximum difference between successive iterations is less than a specified tolerance (ϵ\epsilonϵ).

Pseudocode

6. Input**:**
   - Domain size [xmin,xmax][ymin,ymax].
   - Boundary conditions u(x,y)at the edges.
   - Source term f(x,y)
   - Grid resolution (nx ,ny).
   - Convergence criterion ϵ.
7. Discretize the domain**:**
   - Create a grid with spacing hx, hy.
   - Initialize grid values u(x,y), satisfying boundary conditions.
8. Gauss-Seidel Iteration**:**
   - For each interior grid point (i,j), update:

$$u_{i,j}{}^{k+1} = \frac{u_{i+1,j}{}^{k} + u_{i-1,j}{}^{k+1} + u_{i,j+1}{}^{k} + u_{i,j-1}{}^{k+1} - h^2 f_{i,j}}{4}$$

- Check convergence: Stop when: $\max(|u^{k+1} - u^k|) < \epsilon$

9. Output**:**
   - Final grid values u(x,y).
   - Visualize the results using contour plots and 3D surface plot.

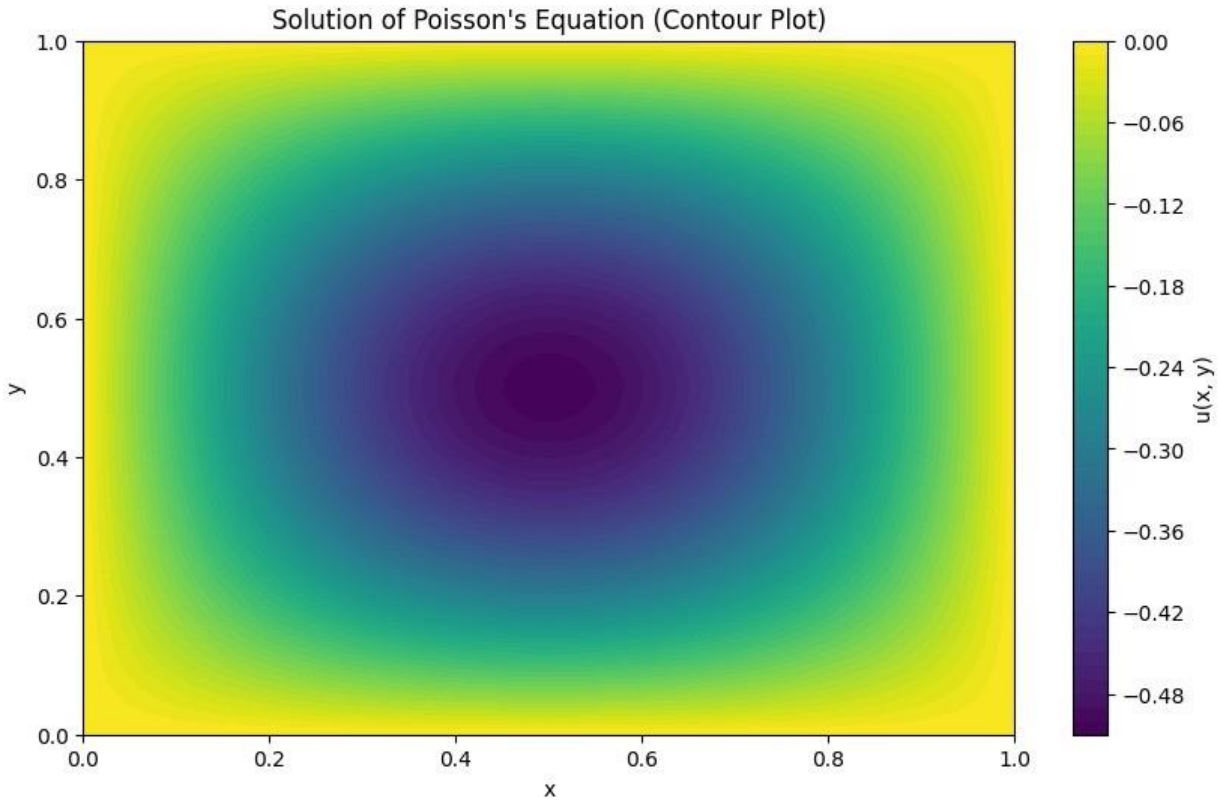
Final grid values u(x,y). Visualize the results using contour plots and 3D surface plots.

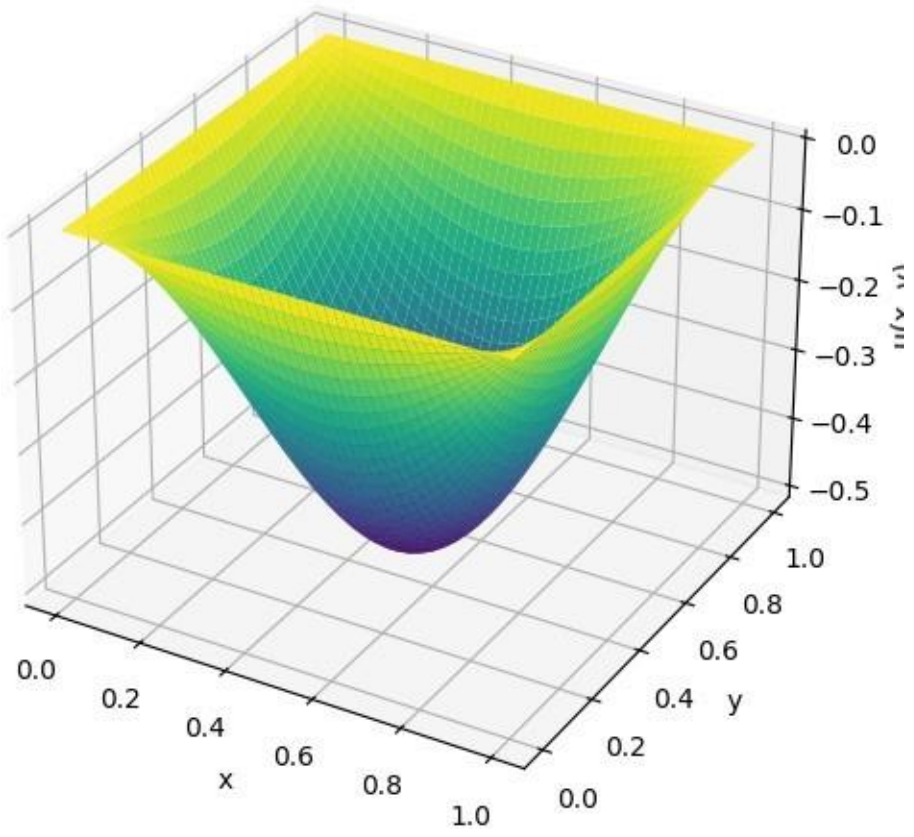
```
import numpy as np
import matplotlib.pyplot as plt

# Define the Poisson solver using Gauss-Seidel iteration
def solve_poisson_gauss_seidel(domain, boundary_conditions, f, nx, ny,
tol):
    """
    Solve the Poisson equation using Gauss-Seidel iteration.

    Parameters:
    domain: tuple (xmin, xmax, ymin, ymax) defining the problem domain
    boundary_conditions: dictionary with keys "top", "bottom", "left",
"right" specifying boundary values
    f: function representing the source term f(x, y)
    nx, ny: number of grid points along x and y axes
    tol: convergence tolerance

    Returns:
    u: 2D array of solution values
    x, y: grid points
```

```
    """
    # Unpack domain
    xmin, xmax, ymin, ymax = domain
    hx = (xmax - xmin) / (nx - 1)
    hy = (ymax - ymin) / (ny - 1)
    h = hx  # Assuming uniform grid spacing

    # Create grid
    x = np.linspace(xmin, xmax, nx)
    y = np.linspace(ymin, ymax, ny)
    u = np.zeros((ny, nx))

    # Apply boundary conditions
    u[0, :] = boundary_conditions["top"]  # Top boundary
    u[-1, :] = boundary_conditions["bottom"]  # Bottom boundary
    u[:, 0] = boundary_conditions["left"]  # Left boundary
    u[:, -1] = boundary_conditions["right"]  # Right boundary

    # Compute source term on the grid
    F = np.zeros((ny, nx))
    for i in range(ny):
        for j in range(nx):
            F[i, j] = f(x[j], y[i])

    # Iterative solution using Gauss-Seidel
    max_iter = 10000
    for iteration in range(max_iter):
        u_old = u.copy()

        # Update interior points
        for i in range(1, ny-1):
            for j in range(1, nx-1):
                u[i, j] = 0.25 * (u[i+1, j] + u[i-1, j] + u[i, j+1] +
u[i, j-1] - h**2 * F[i, j])

        # Check convergence
        error = np.max(np.abs(u - u_old))
        if error < tol:
            print(f"Converged after {iteration} iterations with error
{error}")
            break

    return x, y, u

# Define problem parameters
domain = (0, 1, 0, 1)  # [xmin, xmax, ymin, ymax]
boundary_conditions = {
    "top": 0,    # u(x, y=1) = 0
    "bottom": 0, # u(x, y=0) = 0
    "left": 0,   # u(x=0, y) = 0
```

```
    "right": 0    # u(x=1, y) = 0
}
nx, ny = 50, 50  # Number of grid points
tol = 1e-6  # Convergence tolerance

# Define source term f(x, y)
def source_term(x, y):
    return 10 * np.sin(np.pi * x) * np.sin(np.pi * y)

# Solve the Poisson equation
x, y, u = solve_poisson_gauss_seidel(domain, boundary_conditions,
source_term, nx, ny, tol)

# Visualization
X, Y = np.meshgrid(x, y)

plt.figure(figsize=(10, 6))
plt.contourf(X, Y, u, levels=50, cmap="viridis")
plt.colorbar(label="u(x, y)")
plt.title("Solution of Poisson's Equation (Contour Plot)")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, u, cmap="viridis")
ax.set_title("Solution of Poisson's Equation (3D Surface Plot)")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("u(x, y)")
plt.show()
```

```
Converged after 1858 iterations with error 9.998938609312447e-07
```


Solution of Poisson's Equation (Contour Plot)
1.0
0.8
0.6
0.4
0.2
0.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x
0.00
−0.06
−0.12
−0.18
−0.24
−0.30
−0.36
−0.42
−0.48
u(x, y)


Solution of Poisson's Equation (3D Surface Plot)
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
0.0
0.2
0.4
0.6
0.8
1.0
x
0.0
0.2
0.4
0.6
0.8
1.0
y

3.One-dimensional heat equation using Bendre- Schmidt method

Working Principle

The one-dimensional heat equation is a parabolic partial differential equation that describes the distribution of heat (or temperature) in a given region over time. The general form of the heat equation is:

$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial y^2}$where:

- u(x,t)is the temperature at position x and time t,
- α is the thermal diffusivity constant,
- $\frac{\partial^2 u}{\partial y^2}$ is the spatial second derivative of the temperature.

Finite Difference Approach

We discretize both the time and space domains to solve the heat equation using the finite difference method. The spatial domain is divided into grid points, and the time domain is divided into time steps.

The Bendre-Schmidt method is a specific finite difference approach used to solve the heat equation. It involves discretizing both the time and space domains using explicit schemes.

- For space discretization, we use a grid with spacing Δx.
- For time discretization, we use a time step Δt.

The method leads to an update formula for the temperature at the next time step ${u_i}^{n+1}$:

${u_i}^{n+1} = {u_i}^n + \lambda({u_{i-1}}^n - 2{u_i}^n + {u_{i+1}}^n)$where:

- $\lambda = \frac{\alpha \Delta t}{\Delta x^2}$
- ${u_i}^n$ is the temperature at grid point i and time step n.

Pseudocode:

Input:

- Length of the rod, L,
- Time duration, T,
- Number of grid points nx for space and time steps nt,
- Thermal diffusivity constant α,
- Initial temperature distribution u(x,0)
- Boundary conditions u(0,t)and u(L,t).

Discretization:

- Define spatial grid points x0,x1,...,xn, and time steps t0,t1,...,tm .
- Choose time step size Δt and spatial step size Δx.

Initialization:

- Set the initial temperature profile u(x,0)at all spatial points.
- Apply boundary conditions at x=0and x =L.

Time-stepping:

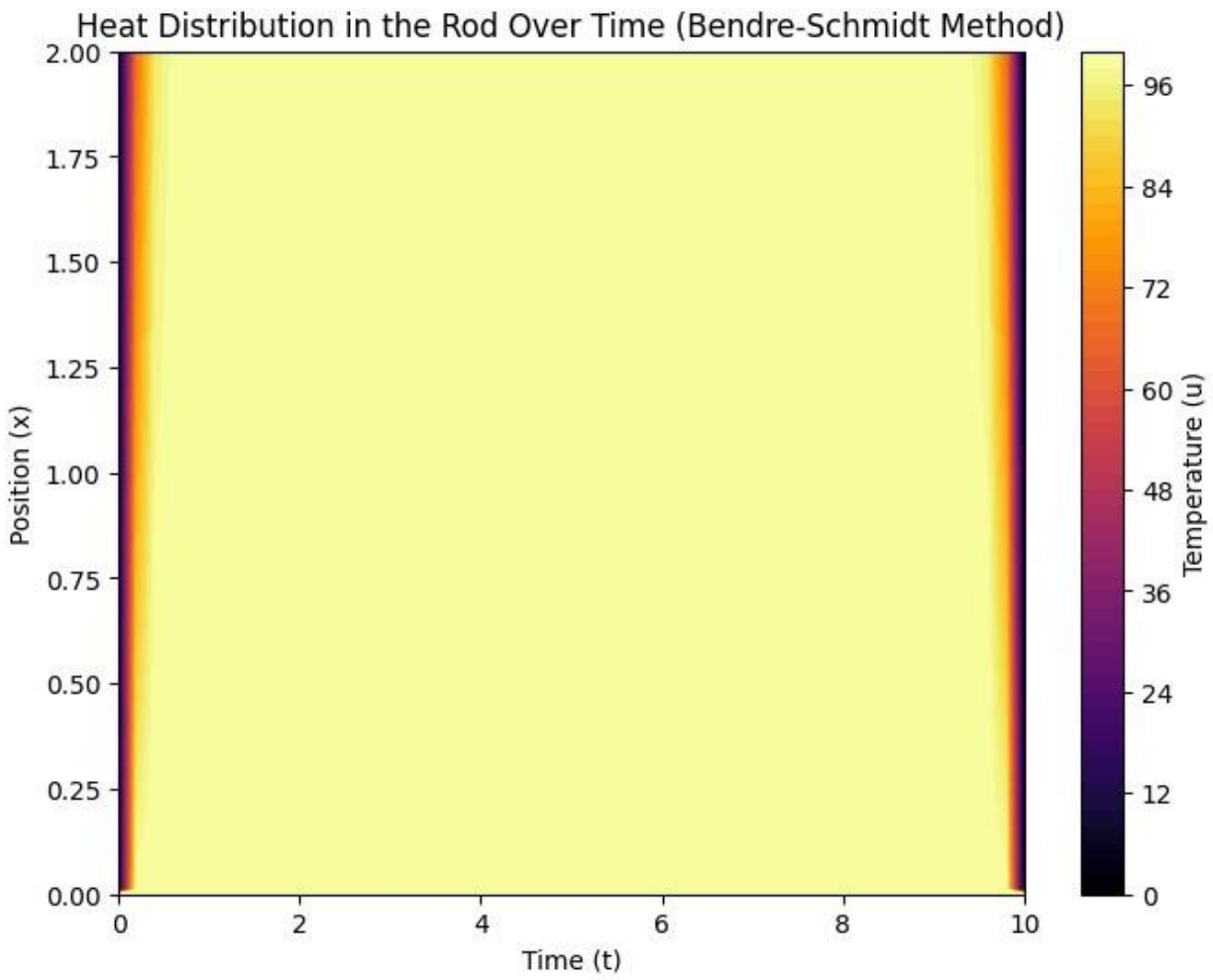
- For each time step nnn, update the temperature profile using the Bendre-Schmidt update rule: $u_i{}^{n+1} = u_i{}^n + \lambda(u_{i-1}{}^n - 2u_i{}^n + u_{i+1}{}^n)$

Output:

- Plot the temperature distribution at different time steps.
- Display the results using a graphical representation (line plot, 3D plot).

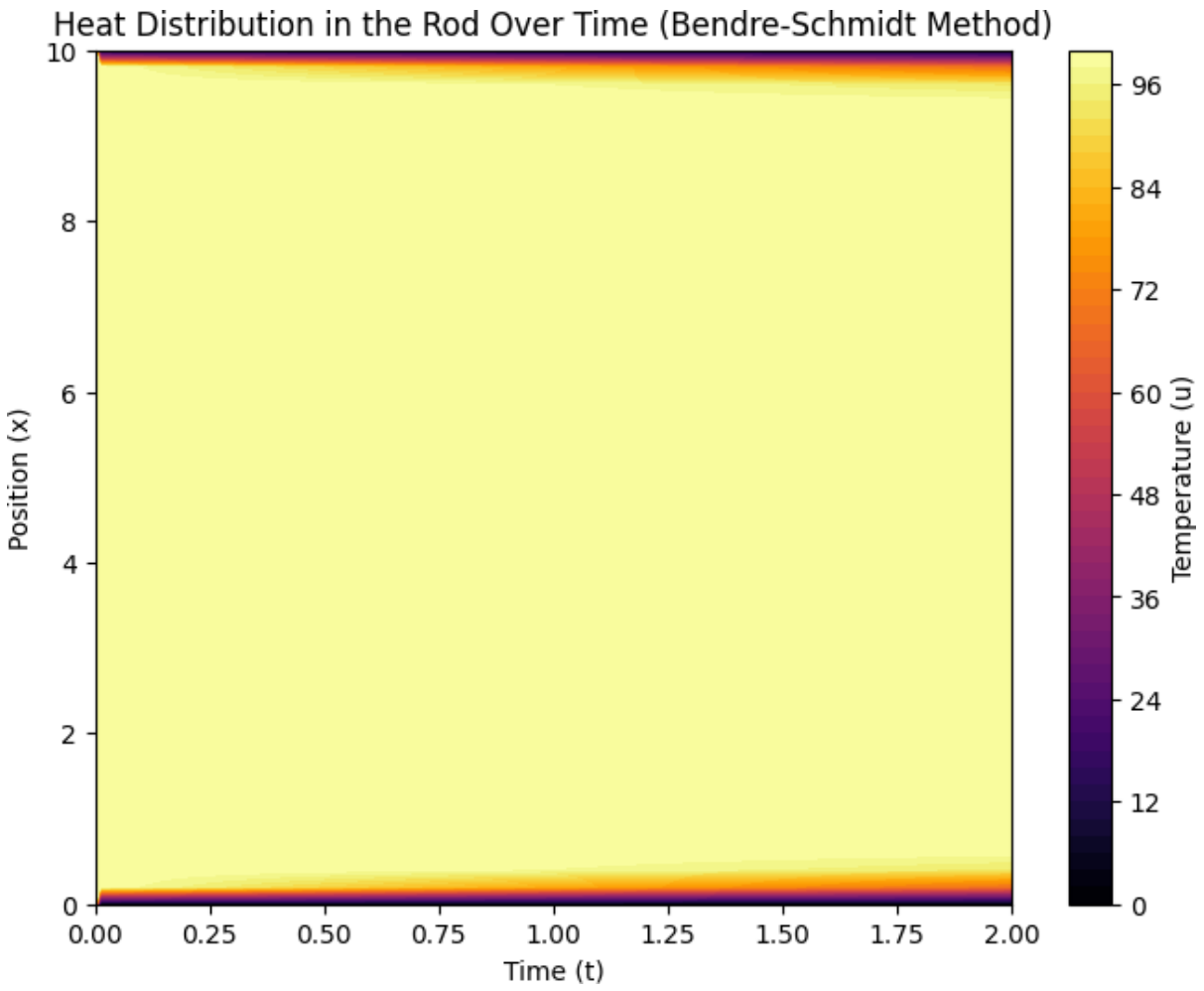
```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
L = 10      # Length of the rod
Tmax = 2    # Total time
```

```
Nx = 50      # Number of spatial steps
Nt = 200     # Number of time steps
alpha = 0.01  # Thermal diffusivity

# Discretization
dx = L / (Nx - 1)
dt = Tmax / (Nt - 1)

# Stability condition for the explicit method
if alpha * dt / dx**2 > 0.5:
    raise ValueError("The stability condition is not satisfied.
Decrease dt or increase dx.")

# Initial and boundary conditions
u = np.zeros((Nt, Nx))  # Temperature distribution
u[:, 0] = 0  # Left boundary
u[:, -1] = 0  # Right boundary
u[0, :] = 100  # Initial temperature distribution

# Bendre-Schmidt method
for n in range(0, Nt-1):
    for i in range(1, Nx-1):
        u[n+1, i] = u[n, i] + alpha * dt / dx**2 * (u[n, i+1] - 2*u[n,
i] + u[n, i-1])

# Visualization: Contour plot
plt.figure(figsize=(8, 6))
t = np.linspace(0, Tmax, Nt)
x = np.linspace(0, L, Nx)

# Create meshgrid for plotting using x and t
X, T = np.meshgrid(x, t)

# Corrected call to contourf with X and T as the axes
plt.contourf(X, T, u, 50, cmap='inferno')  # Contour plot

plt.colorbar(label='Temperature (u)')
plt.title("Heat Distribution in the Rod Over Time (Bendre-Schmidt
Method)")
plt.xlabel("Time (t)")
plt.ylabel("Position (x)")
plt.show()
```

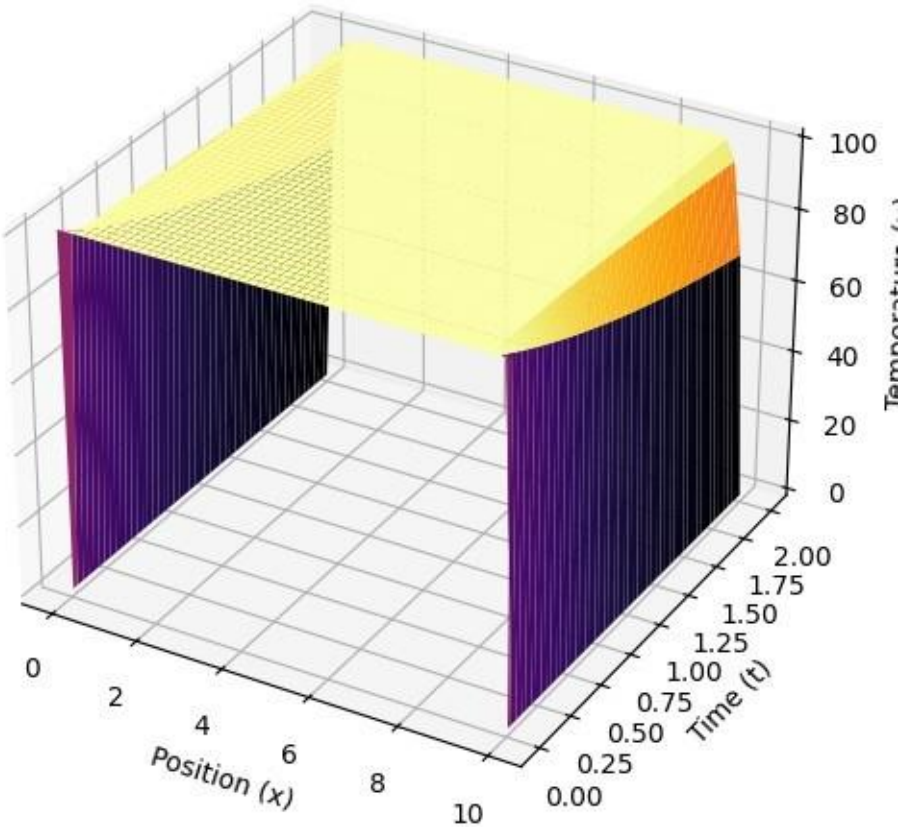



```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Parameters
L = 10       # Length of the rod
Tmax = 2     # Total time
Nx = 50      # Number of spatial steps
Nt = 200     # Number of time steps
alpha = 0.01  # Thermal diffusivity

# Discretization
dx = L / (Nx - 1)
dt = Tmax / (Nt - 1)

# Stability condition for the explicit method
if alpha * dt / dx**2 > 0.5:
    raise ValueError("The stability condition is not satisfied.
Decrease dt or increase dx.")
```

```
# Initial and boundary conditions
u = np.zeros((Nt, Nx))  # Temperature distribution
u[:, 0] = 0  # Left boundary
u[:, -1] = 0  # Right boundary
u[0, :] = 100  # Initial temperature distribution

# Bendre-Schmidt method (Explicit)
for n in range(0, Nt-1):
    for i in range(1, Nx-1):
        u[n+1, i] = u[n, i] + alpha * dt / dx**2 * (u[n, i+1] - 2*u[n,
i] + u[n, i-1])

# Visualization: Contour plot
plt.figure(figsize=(8, 6))
t = np.linspace(0, Tmax, Nt)
x = np.linspace(0, L, Nx)
plt.contourf(t, x, u.T, 50, cmap='inferno')  # Transpose u for correct
dimensions
plt.colorbar(label='Temperature (u)')
plt.title("Heat Distribution in the Rod Over Time (Bendre-Schmidt
Method)")
plt.xlabel("Time (t)")
plt.ylabel("Position (x)")
plt.show()

# 3D Surface Plot
X, T = np.meshgrid(x, t)  # Create meshgrid for plotting
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, T, u, cmap='inferno')  # 3D Surface plot
ax.set_title("Heat Distribution in the Rod Over Time (3D Surface
Plot)")
ax.set_xlabel("Position (x)")
ax.set_ylabel("Time (t)")
ax.set_zlabel("Temperature (u)")
plt.show()
```


Heat Distribution in the Rod Over Time (Bendre-Schmidt Method)
Position (x)
Time (t)
Temperature (u)
0
2
4
6
8
10
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0
12
24
36
48
60
72
84
96

Heat Distribution in the Rod Over Time (3D Surface Plot)



4.One-dimensional heat equation using Crank- Nicholson method

Working principle

The one-dimensional heat equation is given as:

$\frac{\partial u(x,t)}{\partial t} = \alpha \frac{\partial^2 u(x,t)}{\partial y^2}$where:

- u(x,t) is the temperature distribution function over space and time.
- α is the thermal diffusivity constant.
- x represents spatial coordinates, and t represents time.

Crank-Nicholson Method

The Crank-Nicholson method is an implicit finite difference method that is numerically stable and accurate. It is a combination of the forward Euler method (in time) and the central difference method (in space), making it second-order accurate both in space and time.

The Crank-Nicholson method discretizes the heat equation as follows:

$$\frac{1}{\Delta t}\left(u_i^{n+1} - u_i^n\right) = \frac{\alpha}{2}\left(\frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2} + \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}\right)$$

Rearranging the terms results in the following tridiagonal system:

$$-\frac{\alpha\Delta t}{2\Delta x^2}u_{i-1}^{n+1} + \left(1 + \frac{\alpha\Delta t}{\Delta x^2}\right)u_i^{n+1} - \frac{\alpha\Delta t}{2\Delta x^2}u_{i+1}^{n+1} = \frac{\alpha\Delta t}{2\Delta x^2}u_{i-1}^n + \left(1 - \frac{\alpha\Delta t}{\Delta x^2}\right)u_i^n + \frac{\alpha\Delta t}{2\Delta x^2}u_{i+1}^n$$

Where:

- u_i^n is the value of the solution at spatial grid point iii and time step nnn.
- Δt is the time step, and Δx is the spatial grid spacing.

The method involves iterating over time steps and solving the system of equations for each time step to evolve the temperature distribution over time.

Pseudocode:

1. Input:
   - Initial conditions u(x, 0)
   - Boundary conditions u(0, t) and u(L, t)
   - Time step Δt
   - Spatial step Δx
   - Thermal diffusivity constant α

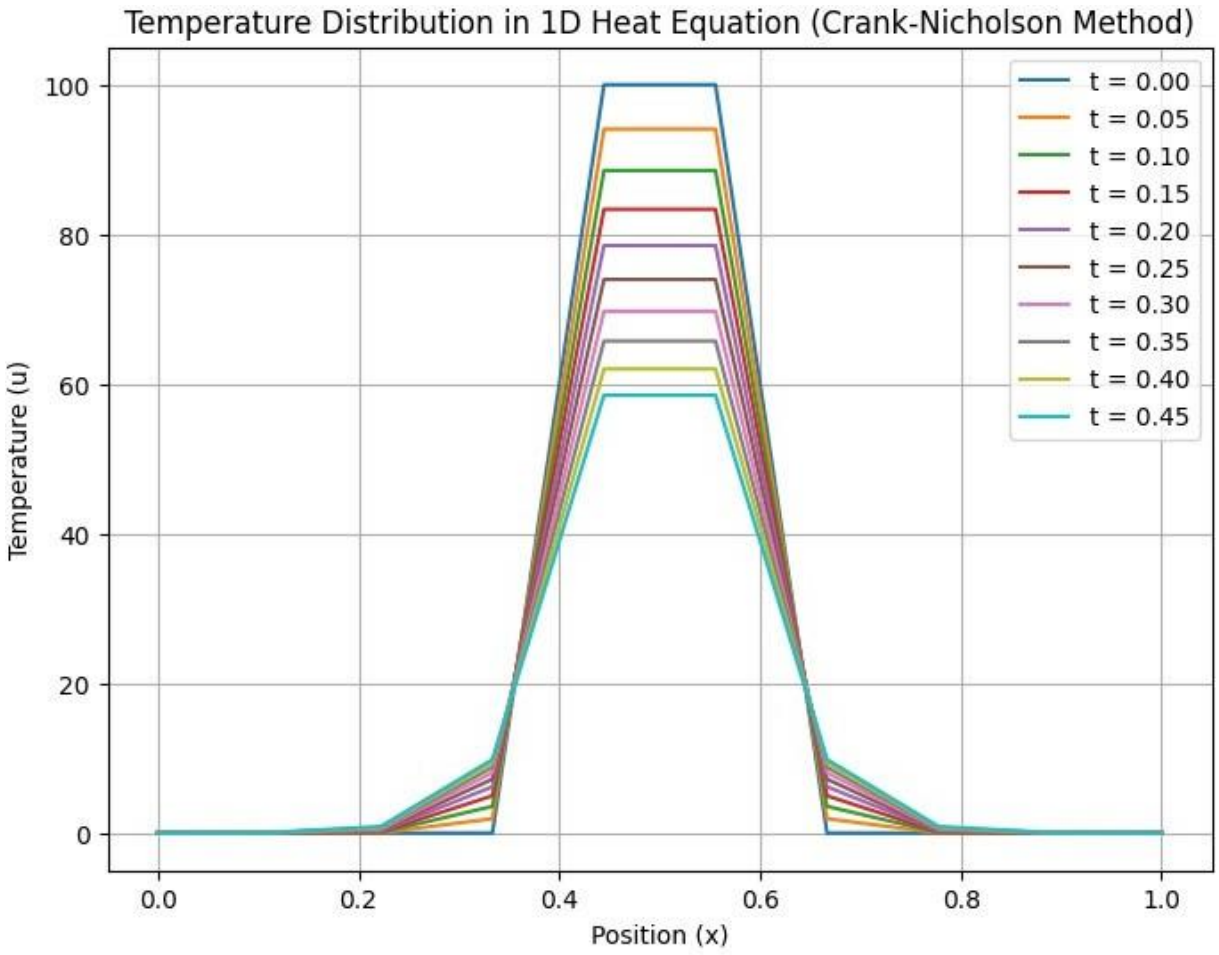
- Total time T

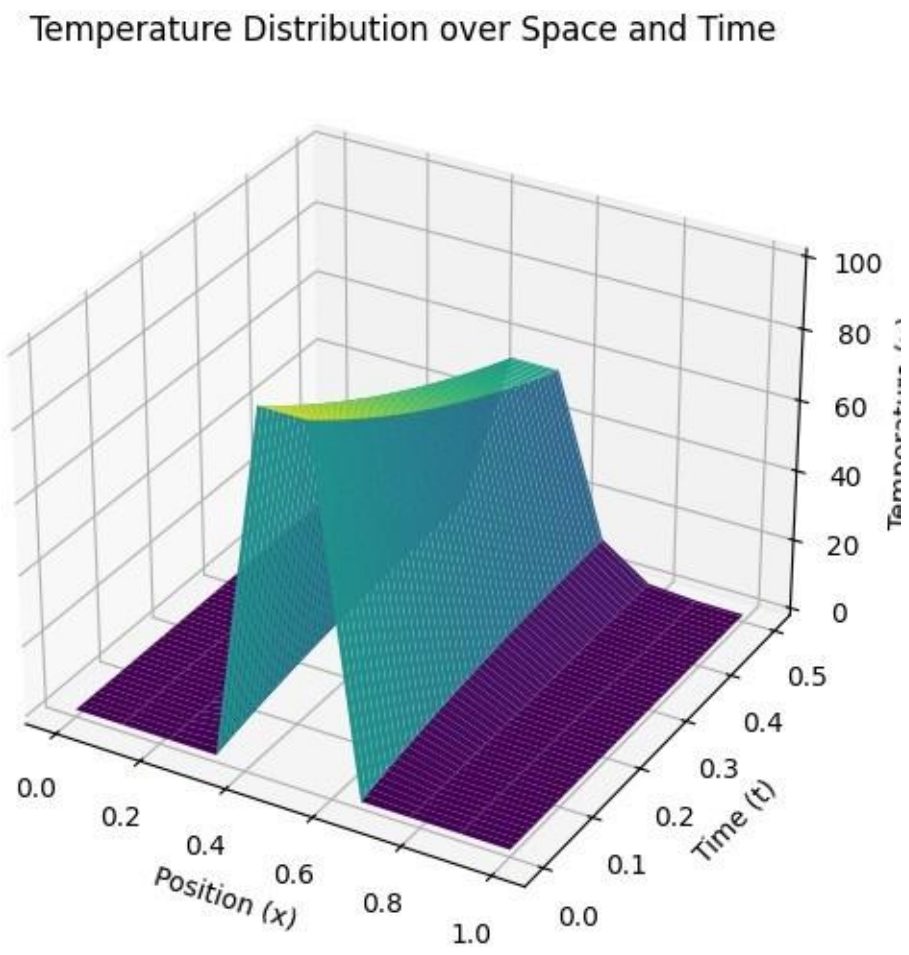
2. Discretize the domain:
   - Define spatial grid with points: x0, x1, ..., xN
   - Define time grid with points: t0, t1, ..., tN
3. Set initial condition for the temperature distribution at t=0.
4. For each time step from t=0 to T:
   - Compute the Crank-Nicholson finite difference scheme for all interior points.
   - Solve the resulting tridiagonal system using Gaussian elimination or other methods.
5. Output: The temperature distribution at each time step.
6. Visualization:
   - Plot the temperature distribution at various time steps using matplotlib.
   - Generate 2D surface plot of the temperature distribution over space and time.
7. Stop when the final time step is reached or maximum iterations are completed.

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

# Crank-Nicholson method for 1D heat equation
def crank_nicholson_heat_eq(L, T, alpha, nx, nt, u_initial,
boundary_conditions):
    """
    Solves the 1D heat equation using the Crank-Nicholson method.

    Parameters:
    - L: length of the rod
    - T: total time
    - alpha: thermal diffusivity constant
    - nx: number of spatial grid points
    - nt: number of time steps
    - u_initial: initial temperature distribution
    - boundary_conditions: boundary conditions at x=0 and x=L

    Returns:
    - x: spatial grid points
    - t: time points
    - u: solution array (temperature distribution over time and space)
    """
    # Discretize the space and time
    dx = L / (nx - 1)  # space step
    dt = T / (nt - 1)  # time step
    r = alpha * dt / (2 * dx**2)  # Crank-Nicholson parameter

    # Initialize the solution array
```

```
    u = np.zeros((nt, nx))

    # Set initial condition
    u[0, :] = u_initial

    # Apply boundary conditions
    u[:, 0] = boundary_conditions[0]  # u(0, t)
    u[:, -1] = boundary_conditions[1]  # u(L, t)

    # Coefficients for the tridiagonal matrix
    A = np.diag((1 + 2 * r) * np.ones(nx - 2))  # Main diagonal
    B = np.diag(-r * np.ones(nx - 3), 1)  # Upper diagonal
    C = np.diag(-r * np.ones(nx - 3), -1)  # Lower diagonal

    for n in range(0, nt - 1):
        # Construct the right-hand side vector
        b = np.zeros(nx - 2)
        for i in range(1, nx - 1):
            b[i - 1] = r * (u[n, i + 1] + u[n, i - 1]) + (1 - 2 * r) *
u[n, i]

        # Solve the system of equations (Ax = b)
        # Using np.linalg.solve to solve the tridiagonal system
        u_next = np.linalg.solve(A, b)

        # Update the solution for the next time step
        u[n + 1, 1:-1] = u_next

    # Return the grid and solution
    x = np.linspace(0, L, nx)
    t = np.linspace(0, T, nt)
    return x, t, u

# Problem parameters
L = 1.0      # length of the rod
T = 0.5      # total time
alpha = 0.01 # thermal diffusivity
nx = 10      # number of spatial points
nt = 100     # number of time steps

# Initial condition: Temperature distribution at t = 0
u_initial = np.zeros(nx)
u_initial[4:6] = 100  # Hot spot in the middle

# Boundary conditions: u(0, t) and u(L, t)
boundary_conditions = [0, 0]  # u(0, t) = 0, u(L, t) = 0

# Solve the heat equation
x, t, u = crank_nicholson_heat_eq(L, T, alpha, nx, nt, u_initial,
boundary_conditions)
```

```
# Visualization: Plot temperature distribution at various time steps
plt.figure(figsize=(8, 6))
for n in range(0, nt, int(nt/10)):  # Plot every 10th time step
    plt.plot(x, u[n, :], label=f"t = {t[n]:.2f}")

plt.title("Temperature Distribution in 1D Heat Equation (Crank-
Nicholson Method)")
plt.xlabel("Position (x)")
plt.ylabel("Temperature (u)")
plt.legend()
plt.grid(True)
plt.show()

# 3D Surface plot of the temperature distribution
X, T = np.meshgrid(x, t)
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, T, u, cmap='viridis')
ax.set_title("Temperature Distribution over Space and Time")
ax.set_xlabel("Position (x)")
ax.set_ylabel("Time (t)")
ax.set_zlabel("Temperature (u)")
plt.show()
```


Temperature Distribution in 1D Heat Equation (Crank-Nicholson Method)
Temperature (u)
Position (x)
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0
t = 0.00
t = 0.05
t = 0.10
t = 0.15
t = 0.20
t = 0.25
t = 0.30
t = 0.35
t = 0.40
t = 0.45


Temperature Distribution over Space and Time
100
80
60
40
20
0
0.0
0.2
0.4
0.6
0.8
1.0
Position (x)
0.0
0.1
0.2
0.3
0.4
0.5
Time (t)

Test Case

Test Case 1:

- Problem Setup**:**
  - Length of rod L=1.0
  - Total time T=0.5
  - Thermal diffusivity α=0.01
  - Grid size nx=10, nt=100
  - Initial temperature distribution: A hot spot in the middle of the rod.
  - Boundary conditions: u(0,t)=0, u(L,t)=0
- Expected Results**:**
  - The temperature in the middle of the rod will decrease over time.
  - Boundary points will remain at zero temperature.

Test Case 2:

- Problem Setup**:**
  - Length of rod L=2.0
  - Total time T=1.0
  - Thermal diffusivity α=0.02
  - Grid size nx=20, nt=200
  - Initial temperature distribution: Heat distributed across the rod.
  - Boundary conditions: u(0,t)=0, u(L,t)=0
- Expected Results**:**
  - Temperature distribution will evolve, and heat will dissipate over time.