NUMERICAL ANALYSIS AND PROGRAMMING

CAT II

b)

i)Differentiation

import numpy as npy

def f(x):

return x\*\*3 + 2\*x\*\*2 + x + 1

x = np.linspace(0, 10, 100)

y = f(x)

dy\_dx = np.gradient(y, x)

A screenshot of a computer

Description automatically generatedprint(f"The numerical derivative is: {dy\_dx}")

ii)Numerical Integration

import numpy as np

def f(x):

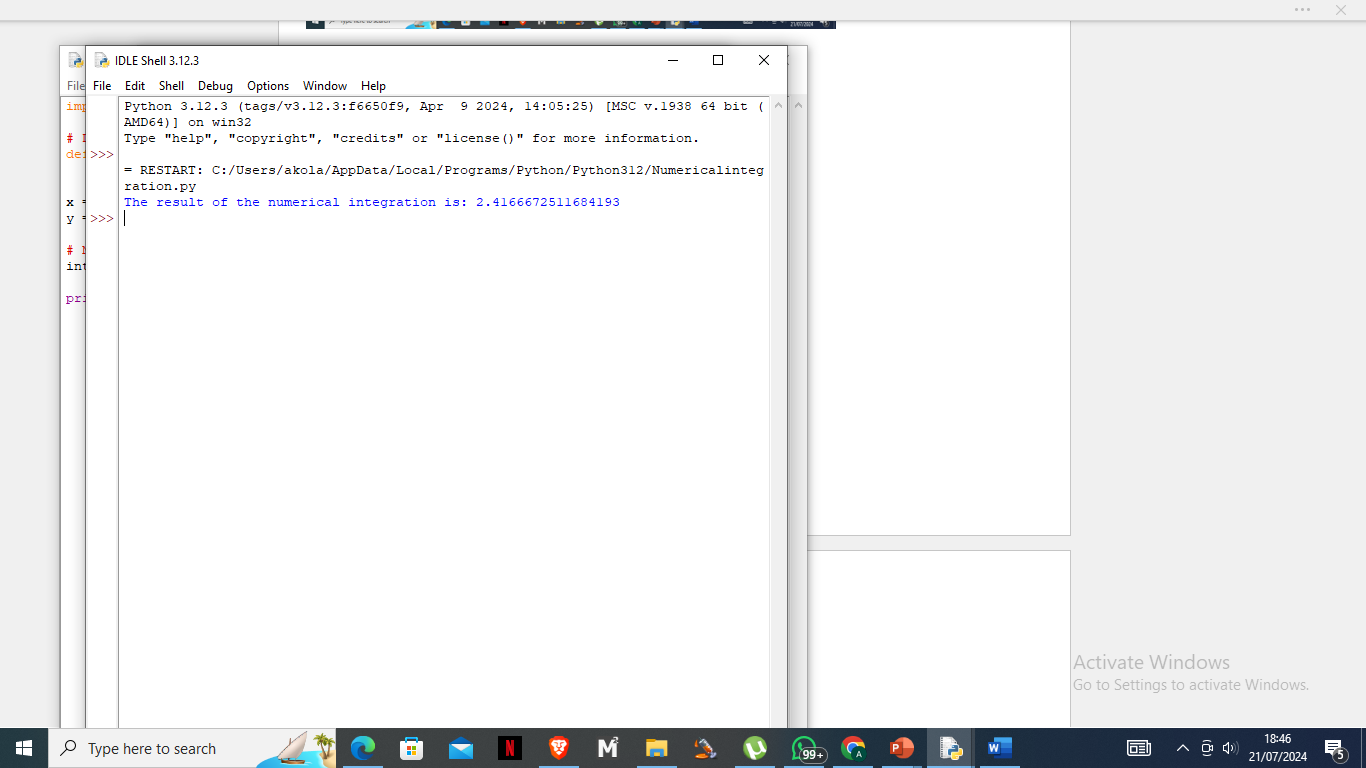
return x\*\*3 + 2\*x\*\*2 + x + 1

x = np.linspace(0, 1, 1000)

y = f(x)

integral = np.trapz(y, x)

print(f"The result of the numerical integration is: {integral}")



iii)Curve Fitting

import numpy as np

import matplotlib.pyplot as plt

# Define the function to fit

def func(x, a, b, c):

return a \* np.exp(b \* x) + c

# Generate some data

xdata = np.linspace(0, 4, 50)

ydata = func(xdata, 2.5, 1.3, 0.5) + 0.2 \* np.random.normal(size=len(xdata))

# Define the cost function for curve fitting

def cost(params, x, y):

a, b, c = params

return np.sum((y - func(x, a, b, c))\*\*2)

# Initial guess for the parameters

initial\_guess = [1, 1, 1]

# Perform the curve fitting using gradient descent

from scipy.optimize import minimize

result = minimize(cost, initial\_guess, args=(xdata, ydata))

popt = result.x

print(f"Fitted parameters: {popt}")

# Plot the data and the fit

plt.scatter(xdata, ydata, label='Data')

plt.plot(xdata, func(xdata, \*popt), label='Fit', color='red')

plt.legend()

plt.show()

iv)Linear Regression

import numpy as np

import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)

y = 3 \* x + 7 + np.random.normal(size=x.size)

A = np.vstack([x, np.ones\_like(x)]).T

slope, intercept = np.linalg.lstsq(A, y, rcond=None)[0]

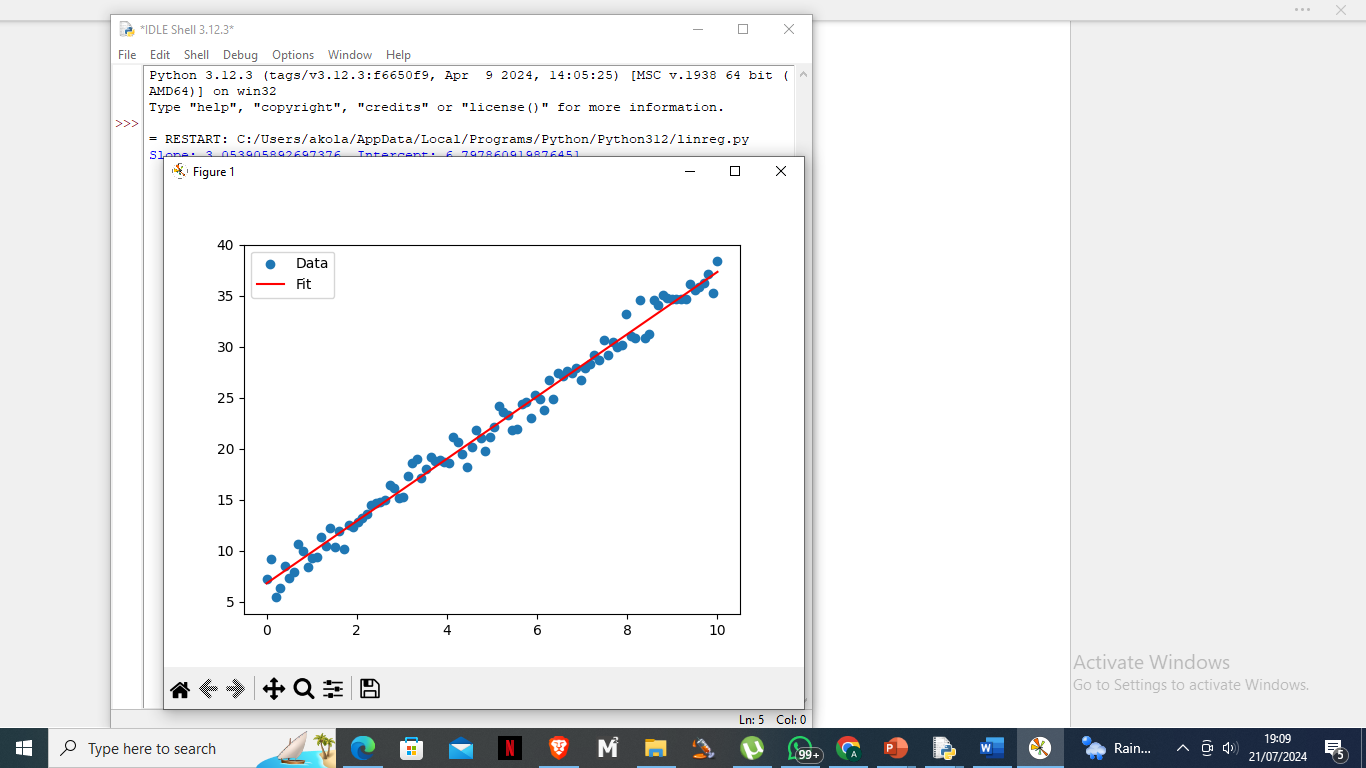
print(f"Slope: {slope}, Intercept: {intercept}")

plt.scatter(x, y, label='Data')

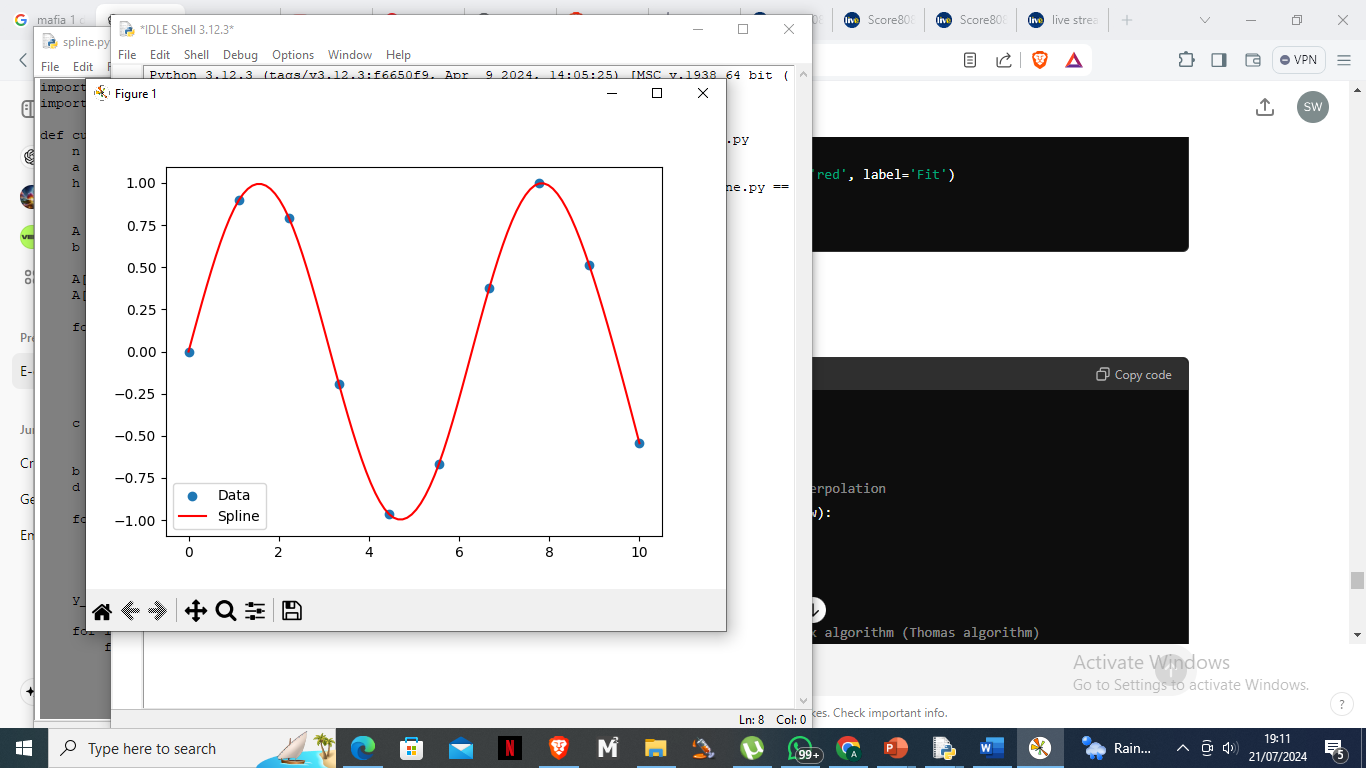
plt.plot(x, slope \* x + intercept, color='red', label='Fit')

plt.legend()

plt.show()



v)Spline Interpolation



import numpy as np

import matplotlib.pyplot as plt

def cubic\_spline\_interpolation(x, y, x\_new):

n = len(x)

a = y

h = np.diff(x)

A = np.zeros((n, n))

b = np.zeros(n)

A[0, 0] = 1

A[-1, -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 \* (a[i+1] - a[i]) / h[i] - 3 \* (a[i] - a[i-1]) / h[i-1]

c = np.linalg.solve(A, b)

b = np.zeros(n-1)

d = np.zeros(n-1)

for i in range(n-1):

b[i] = (a[i+1] - a[i]) / h[i] - h[i] \* (c[i+1] + 2\*c[i]) / 3

d[i] = (c[i+1] - c[i]) / (3 \* h[i])

y\_new = np.zeros\_like(x\_new)

for i in range(len(x\_new)):

for j in range(n-1):

if x[j] <= x\_new[i] <= x[j+1]:

dx = x\_new[i] - x[j]

y\_new[i] = a[j] + b[j]\*dx + c[j]\*dx\*\*2 + d[j]\*dx\*\*3

return y\_new

x = np.linspace(0, 10, 10)

y = np.sin(x)

x\_new = np.linspace(0, 10, 100)

y\_new = cubic\_spline\_interpolation(x, y, x\_new)

plt.scatter(x, y, label='Data')

plt.plot(x\_new, y\_new, label='Spline', color='red')

plt.legend()

plt.show()

c) import numpy as np

# Given coordinates

x\_coords = np.array([2.00, 4.25])

y\_coords = np.array([7.2, 7.1])

# Point to interpolate

x\_to\_find = 4.0

# Linear interpolation

x1, x2 = x\_coords

y1, y2 = y\_coords

# Calculate the slope (y2 - y1) / (x2 - x1)

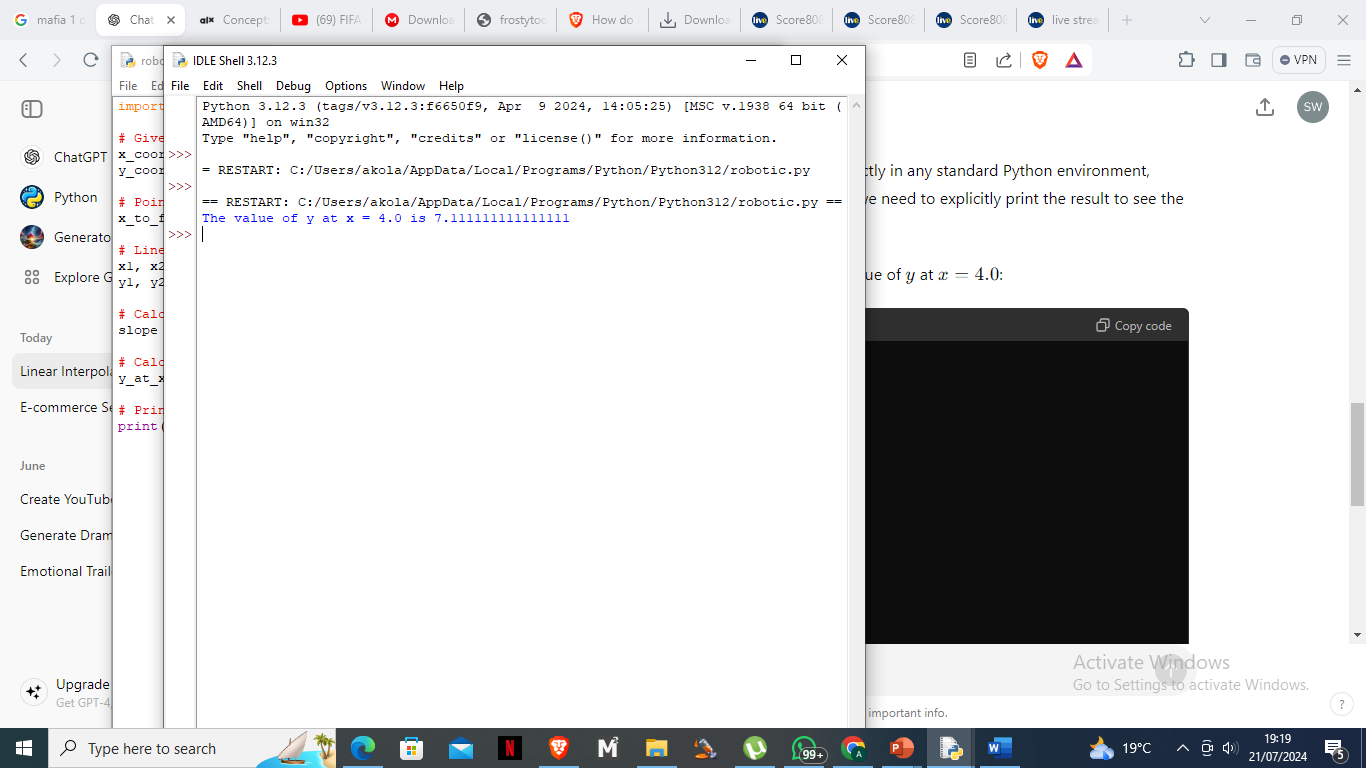
slope = (y2 - y1) / (x2 - x1)

# Calculate the interpolated y value

y\_at\_x = y1 + slope \* (x\_to\_find - x1)

# Print the result

print(f"The value of y at x = {x\_to\_find} is {y\_at\_x}")



e)

import numpy as np

import matplotlib.pyplot as plt

# Define the parameters

f1 = 50 # Frequency 1 in Hz

f2 = 120 # Frequency 2 in Hz

fs = 1000 # Sampling frequency in Hz

t = np.arange(0, 1, 1/fs) # Time vector for 1 second

# Define the signal

s = np.sin(2 \* np.pi \* f1 \* t) + np.sin(2 \* np.pi \* f2 \* t)

# Compute the FFT

S = np.fft.fft(s)

freq = np.fft.fftfreq(len(s), 1/fs)

# Plot the signal

plt.figure(figsize=(12, 6))

plt.subplot(2, 1, 1)

plt.plot(t, s)

plt.title('Signal in Time Domain')

plt.xlabel('Time [s]')

plt.ylabel('Amplitude')

plt.subplot(2, 1, 2)

plt.plot(freq[:len(freq)//2], np.abs(S)[:len(S)//2]) # Plot only the positive frequencies

plt.title('Signal in Frequency Domain')

plt.xlabel('Frequency [Hz]')

plt.ylabel('Magnitude')

plt.tight\_layout()

plt.show()

g)

import numpy as np

def trapezoidal\_rule(func, a, b, n):

"""

Compute the integral of a function using the trapezoidal rule.

Parameters:

func : function

The function to integrate.

a : float

The start of the interval.

b : float

The end of the interval.

n : int

The number of sub-intervals.

Returns:

float

The approximate integral of the function.

"""

x = np.linspace(a, b, n+1)

y = func(x)

h = (b - a) / n

integral = (h/2) \* (y[0] + 2 \* np.sum(y[1:-1]) + y[-1])

return integral

# Define a sample function to integrate

def sample\_func(x):

return x\*\*2

# Set the interval and number of sub-intervals

a = 0

b = 1

n = 100

# Compute the integral

integral = trapezoidal\_rule(sample\_func, a, b, n)

print(f"The integral of the function from {a} to {b} is approximately {integral:.6f}")

# For visualization

import matplotlib.pyplot as plt

x = np.linspace(a, b, 1000)

y = sample\_func(x)

plt.plot(x, y, label='f(x) = x^2')

# Trapezoids

x\_trap = np.linspace(a, b, n+1)

y\_trap = sample\_func(x\_trap)

plt.fill\_between(x\_trap, y\_trap, alpha=0.3, label='Trapezoids')

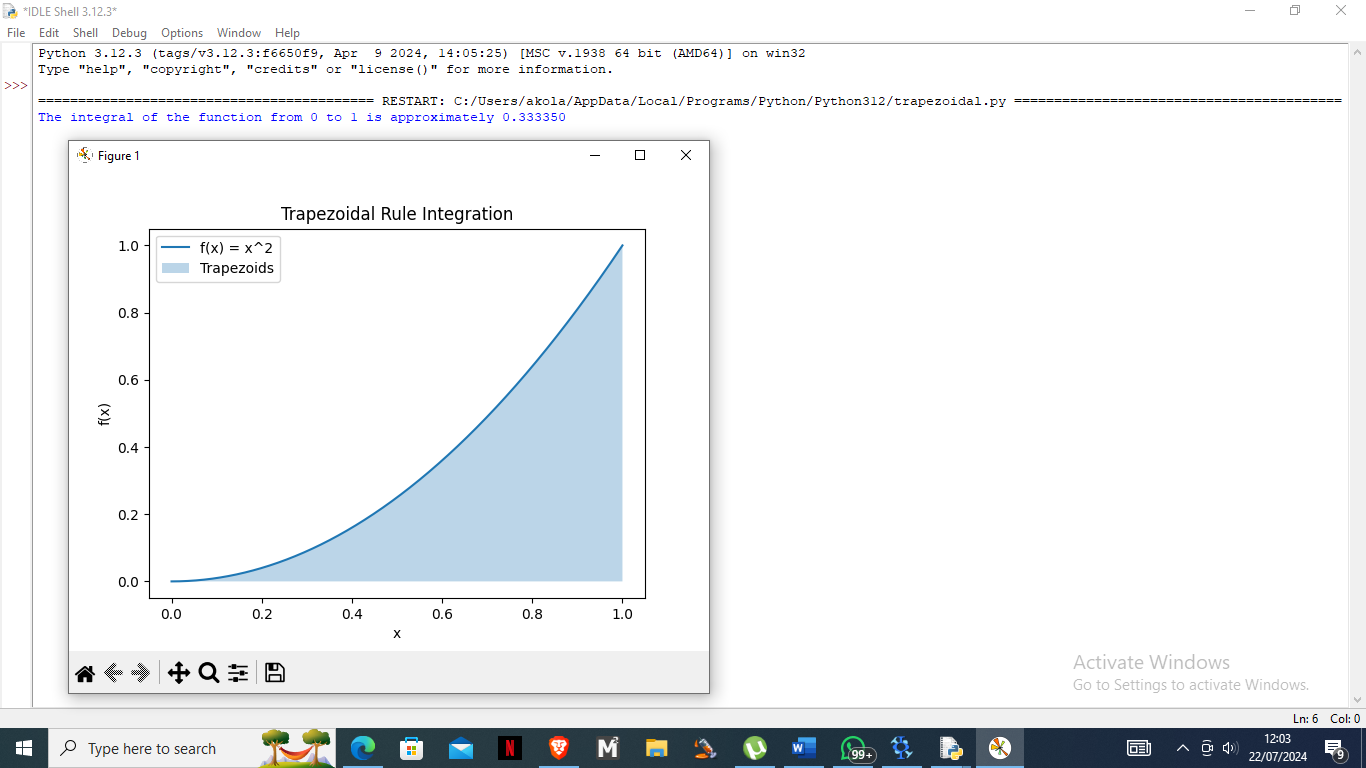
plt.title('Trapezoidal Rule Integration')

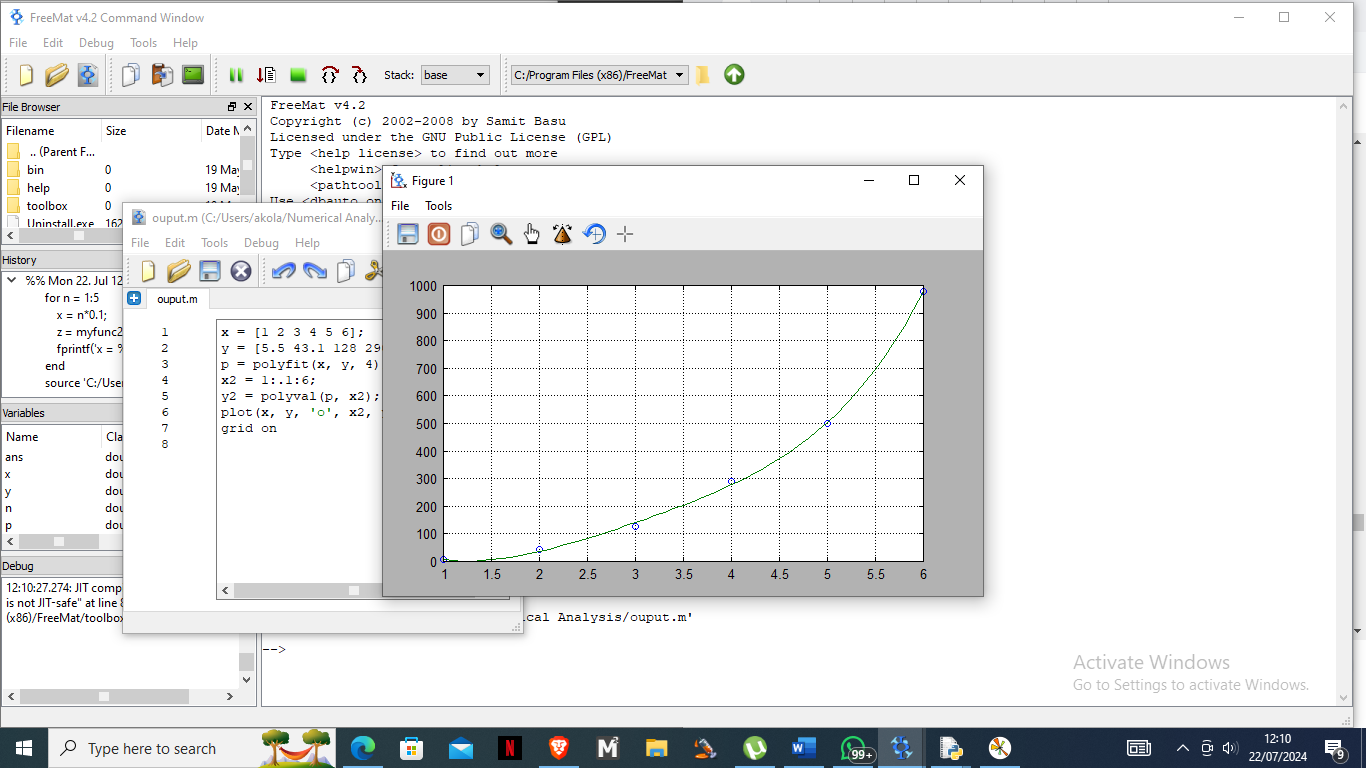
plt.xlabel('x')

plt.ylabel('f(x)')

plt.legend()

plt.show()



h)

i) x and y define the data points.

ii) polyfit(x, y, 4) computes the coefficients of a 4th-degree polynomial that fits the data points (x, y).

iii) x2 creates a vector from 1 to 6 with increments of 0.1.

iv) polyval(p, x2) evaluates the polynomial with coefficients p at each point in x2.

v) plot(x, y, 'o', x2, y2) plots the original data points as circles and the polynomial fit as a line.

vi) grid on adds a grid to the plot.

i)

i) import numpy as np

def lagrange\_interpolation(x, y, x\_val):

n = len(x)

result = 0

for i in range(n):

term = y[i]

for j in range(n):

if j != i:

term = term \* (x\_val - x[j]) / (x[i] - x[j])

result += term

return result

# Given data points

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

y = np.array([1, 4, 9, 16])

# Evaluate at several points

x\_vals = np.linspace(1, 4, 100)

y\_vals = [lagrange\_interpolation(x, y, xv) for xv in x\_vals]

# Plot the results

import matplotlib.pyplot as plt

plt.plot(x, y, 'o', label='Data points')

plt.plot(x\_vals, y\_vals, '-', label='Lagrange Interpolation')

plt.xlabel('x')

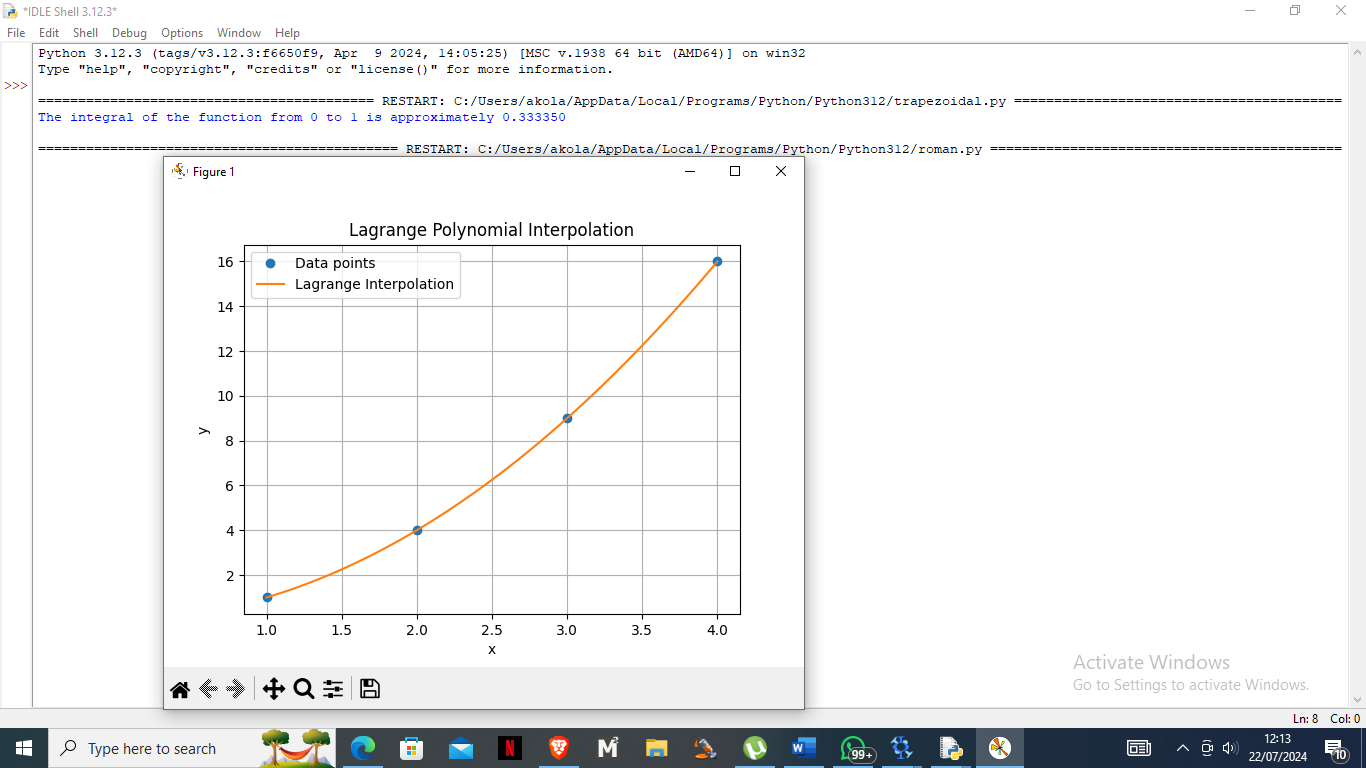
plt.ylabel('y')

plt.legend()

plt.grid(True)

plt.title('Lagrange Polynomial Interpolation')

plt.show()



ii) def divided\_diff(x, y):

n = len(y)

coef = np.zeros([n, n])

coef[:, 0] = y

for j in range(1, n):

for i in range(n - j):

coef[i, j] = (coef[i + 1, j - 1] - coef[i, j - 1]) / (x[i + j] - x[i])

return coef[0, :]

def newton\_poly(coef, x\_data, x):

n = len(coef) - 1

p = coef[n]

for k in range(1, n + 1):

p = coef[n - k] + (x - x\_data[n - k]) \* p

return p

# Given data points

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

y = np.array([1, 4, 9, 16])

# Calculate coefficients

coef = divided\_diff(x, y)

# Evaluate at several points

x\_vals = np.linspace(1, 4, 100)

y\_vals = [newton\_poly(coef, x, xv) for xv in x\_vals]

# Plot the results

plt.plot(x, y, 'o', label='Data points')

plt.plot(x\_vals, y\_vals, '-', label='Newton Interpolation')

plt.xlabel('x')

plt.ylabel('y')

plt.legend()

plt.grid(True)

plt.title('Newton Polynomial Interpolation')

plt.show()

iii)

j)

import numpy as np

def power\_iteration(A, num\_simulations: int):

# Choose a random vector to start with

b\_k = np.random.rand(A.shape[1])

for \_ in range(num\_simulations):

# Calculate the matrix-by-vector product Ab

b\_k1 = np.dot(A, b\_k)

# Re normalize the vector

b\_k1\_norm = np.linalg.norm(b\_k1)

b\_k = b\_k1 / b\_k1\_norm

eigenvalue = np.dot(b\_k.T, np.dot(A, b\_k)) / np.dot(b\_k.T, b\_k)

eigenvector = b\_k

return eigenvalue, eigenvector

# Define the matrix

A = np.array([[4, 1, 1],

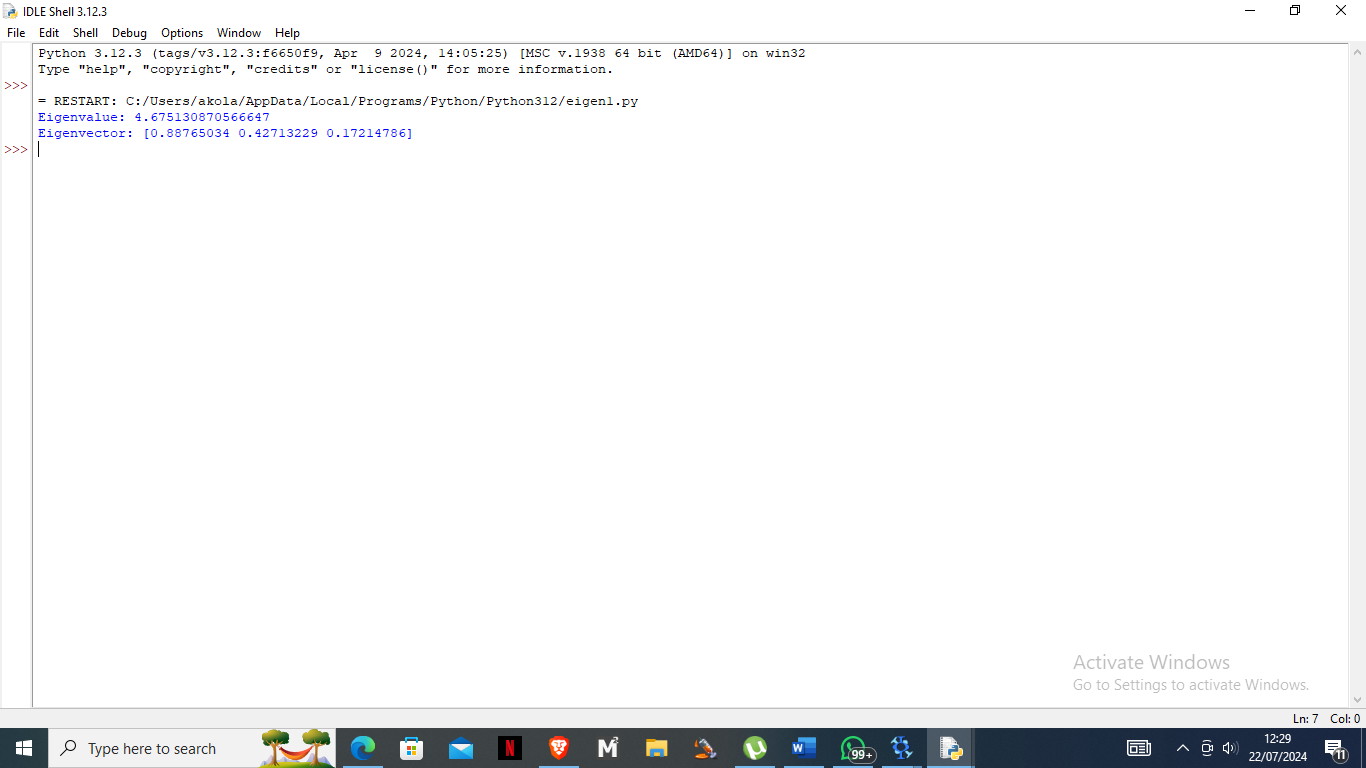
[1, 3, -1],

[1, -1, 2]])

eigenvalue, eigenvector = power\_iteration(A, 1000)

print("Eigenvalue:", eigenvalue)

print("Eigenvector:", eigenvector)



ii) def qr\_algorithm(A, num\_simulations: int):

Ak = A.copy()

Q\_prod = np.eye(A.shape[0])

for \_ in range(num\_simulations):

Q, R = np.linalg.qr(Ak)

Ak = np.dot(R, Q)

Q\_prod = np.dot(Q\_prod, Q)

eigenvalues = np.diag(Ak)

eigenvectors = Q\_prod

return eigenvalues, eigenvectors

# Define the matrix

A = np.array([[4, 1, 1],

[1, 3, -1],

[1, -1, 2]])

eigenvalues, eigenvectors = qr\_algorithm(A, 100)

print("Eigenvalues:", eigenvalues)

print("Eigenvectors:")

print(eigenvectors)

k)

import matplotlib.pyplot as plt

import numpy as np

def f(x, y):

return x\*\*2 + y\*\*2 - x\*y + x - y + 1

def grad\_f(x, y):

df\_dx = 2\*x - y + 1

df\_dy = 2\*y - x - 1

return [df\_dx, df\_dy]

def gradient\_descent(starting\_point, learning\_rate, num\_iterations):

point = list(starting\_point)

points = [point.copy()]

for \_ in range(num\_iterations):

grad = grad\_f(point[0], point[1])

point[0] -= learning\_rate \* grad[0]

point[1] -= learning\_rate \* grad[1]

points.append(point.copy())

return points

# Parameters

starting\_point = [0, 0]

learning\_rate = 0.1

num\_iterations = 100

# Perform Gradient Descent

points = gradient\_descent(starting\_point, learning\_rate, num\_iterations)

# Extract points for plotting

x\_points = [point[0] for point in points]

y\_points = [point[1] for point in points]

# Create a grid of x and y values

x = np.linspace(-1, 2, 400)

y = np.linspace(-1, 2, 400)

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

Z = f(X, Y)

# Plot the results

plt.contour(X, Y, Z, levels=50)

plt.plot(x\_points, y\_points, 'r.-')

plt.title('Gradient Descent Optimization')

plt.xlabel('x')

plt.ylabel('y')

plt.show()

