ADRIAN KIPRUTO KIRUI

INTE/MG/3002/09/22

INTE 316

NUMERICAL ANALYSIS AND DESIGN

a) Define the function f(x)=x2−x−2f(x) = x^2 - x - 2f(x)=x2−x−2.

x=

f(b)·a−f(a)·b / f(b)−f(a)

a=1,b=3

f(a)=f(1)=12−1−2=−2

f(b)=f(3)=32−3−2=4

Iteration 1

x1= f(b)·a−f(a)·b / f(b)−f(a)

4·1−(−2)·3/ 4-(-2)

=(4+6)/6=10/6≈1.6667

f(x1)=f(1.6667)=(1.6667)2−1.6667−2≈−0.5556

Since f(a)·f(x1)<0f(a) update b=x1

b=1.6667,f(b)=−0.5556

Iteration 2

X2= f(b)·a−f(a)·b / f(b)−f(a)

(−0.5556)·1−(−2)·1.6667) / −0.5556−(−2)

=−0.5556+3.3334 / 1.4444

≈1.9286

f(x2)=f(1.9286)=(1.9286)2−1.9286−2≈0.3927

Since f(a)·f(x2)<0 update b=x2

b=1.9286,f(b)=0.3927

Iteration 3

X3= f(b)·a−f(a)·b / f(b)−f(a)

0.3927·1−(−2)·1.9286 / 0.3927-(-2)

=0.3927+3.8572 /2.3927

≈1.7692

f(x3)=f(1.7692)=(1.7692)2−1.7692−2≈−0.1003

Since f(a)·f(x3)<0 update b=x3

b=1.7692,f(b)=−0.1003

After three iterations, the root is approximately

x≈1.7692

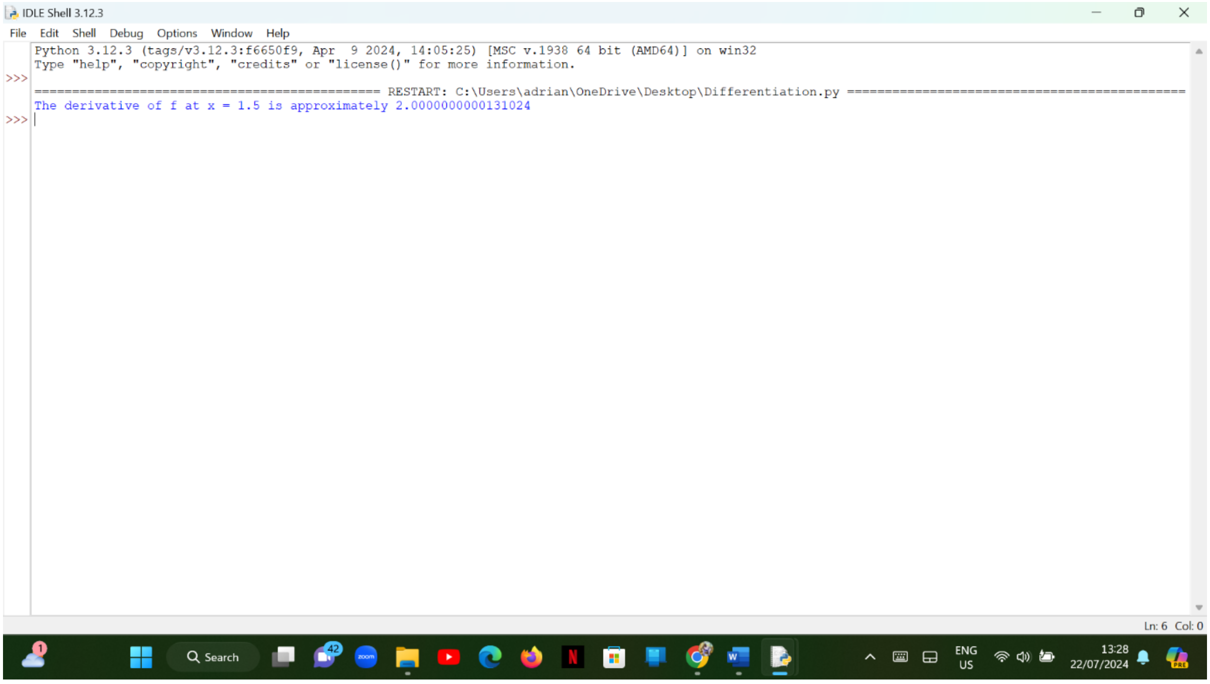
b)Using PYTHON show how the following is achieved**(PRACTICAL)**

i. Differentiation

code

```
import numpy as np

# Define the function
def f(x):
   return x**2 - x - 2

# Define the point at which to differentiate and the step size
x = 1.5
h = 1e-5

# Numerical differentiation (finite difference method)
df_dx = (f(x + h) - f(x - h)) / (2 * h)
print(f"The derivative of f at x = {x} is approximately {df_dx}")
```

output

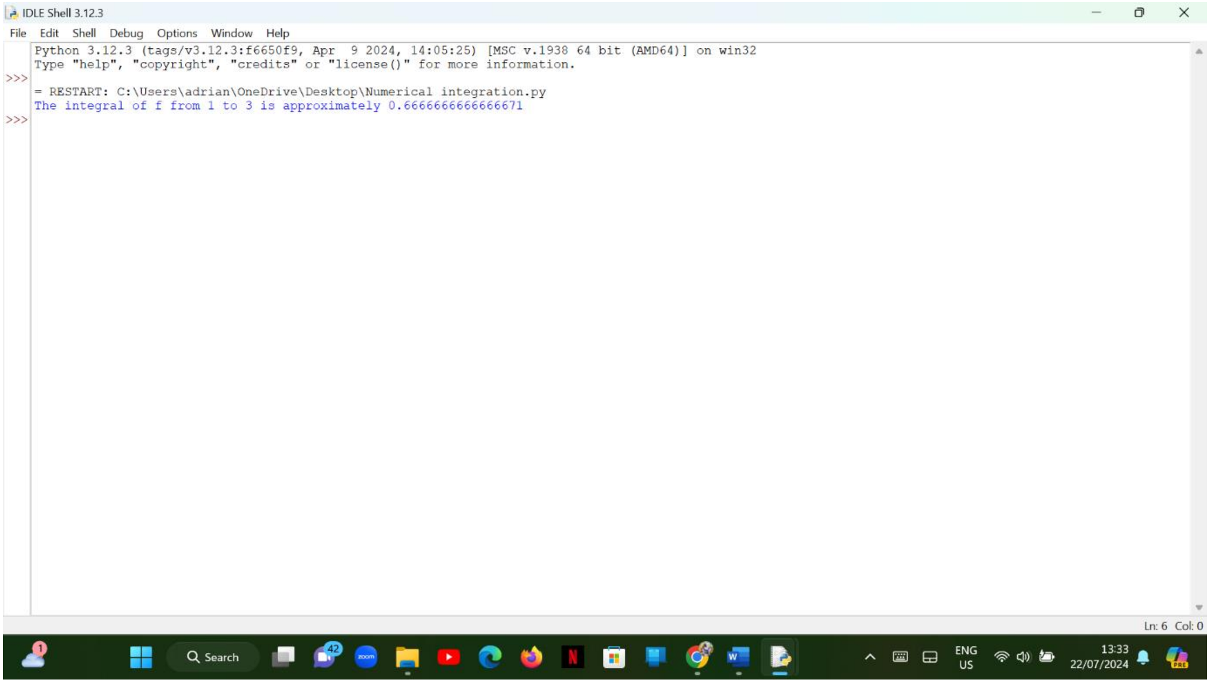


ii)Numerical integration

code

```
import scipy.integrate as spi

# Define the function
def f(x):
    return x**2 - x - 2

# Define the limits of integration
a = 1
b = 3

# Numerical integration
integral, error = spi.quad(f, a, b)
print(f"The integral of f from {a} to {b} is approximately {integral}")
```

output



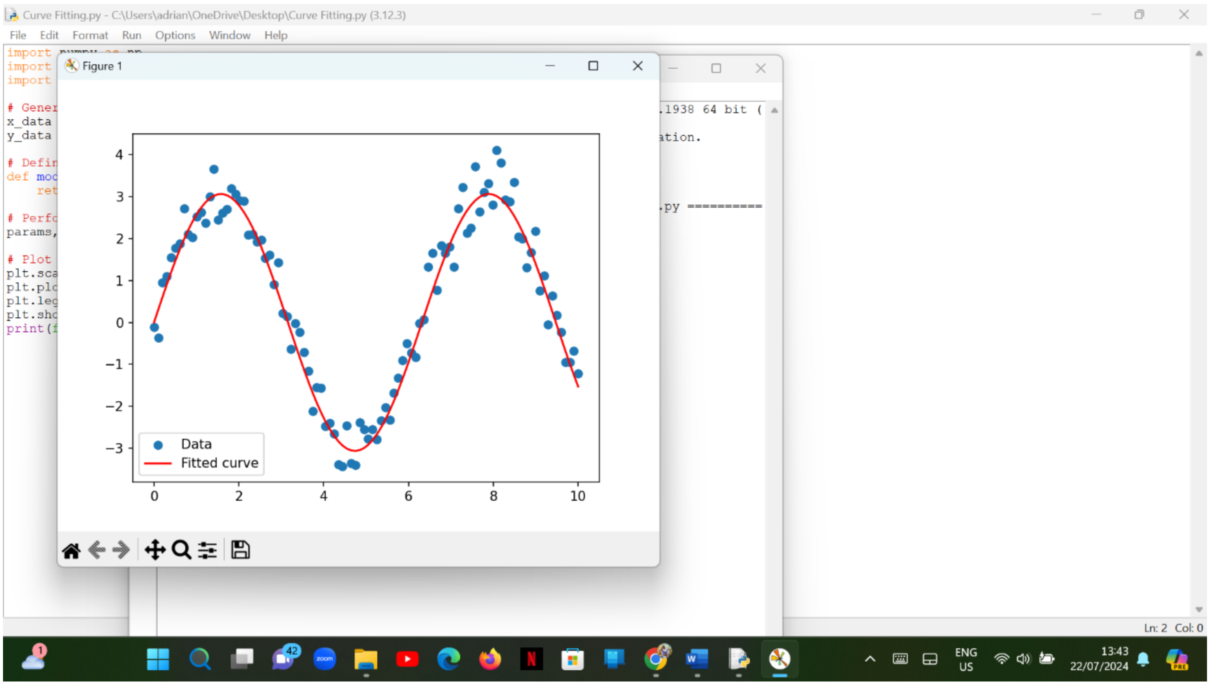
iii)Curve Fitting

code

```
import numpy as np

import scipy.optimize as spo

import matplotlib.pyplot as plt


# Generate some data points

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

y_data = 3 * np.sin(x_data) + np.random.normal(0, 0.5, x_data.shape)


# Define the model function

def model(x, a, b):
```

```
    return a * np.sin(b * x)


# Perform curve fitting
params, params_covariance = spo.curve_fit(model, x_data, y_data, p0=[2, 1])


# Plot the data and the fitted curve
plt.scatter(x_data, y_data, label='Data')
plt.plot(x_data, model(x_data, *params), label='Fitted curve', color='red')
plt.legend()
plt.show()
print(f"Fitted parameters: {params}")
```

output

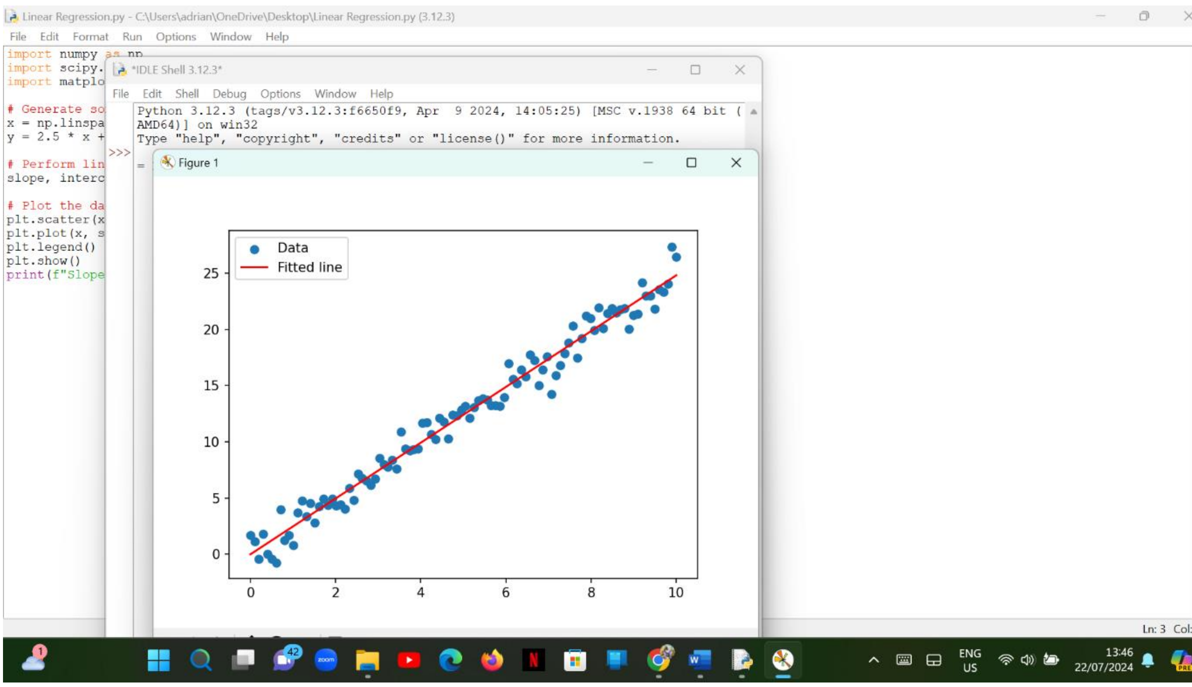


iv) Linear Regression

code

```
import numpy as np

import scipy.stats as sps

import matplotlib.pyplot as plt


# Generate some data points

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

y = 2.5 * x + np.random.normal(0, 1, x.shape)


# Perform linear regression

slope, intercept, r_value, p_value, std_err = sps.linregress(x, y)


# Plot the data and the fitted line

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

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

plt.legend()

plt.show()

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

output

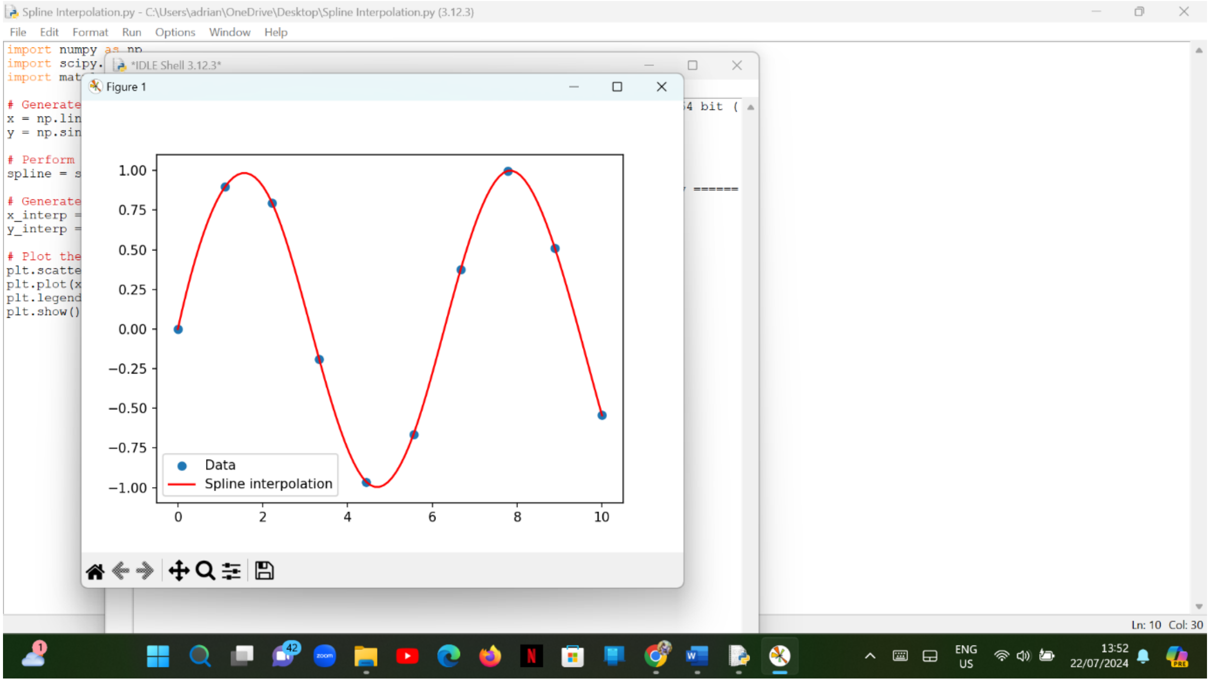


v)Spline Interpolation

code

```
import numpy as np
import scipy.interpolate as spi
import matplotlib.pyplot as plt

# Generate some data points
x = np.linspace(0, 10, 10)
y = np.sin(x)

# Perform spline interpolation
spline = spi.CubicSpline(x, y)

# Generate points for interpolation
x_interp = np.linspace(0, 10, 100)
y_interp = spline(x_interp)

# Plot the data and the spline interpolation
plt.scatter(x, y, label='Data')
plt.plot(x_interp, y_interp, label='Spline interpolation', color='red')
plt.legend()
plt.show()
```

OUTPUT



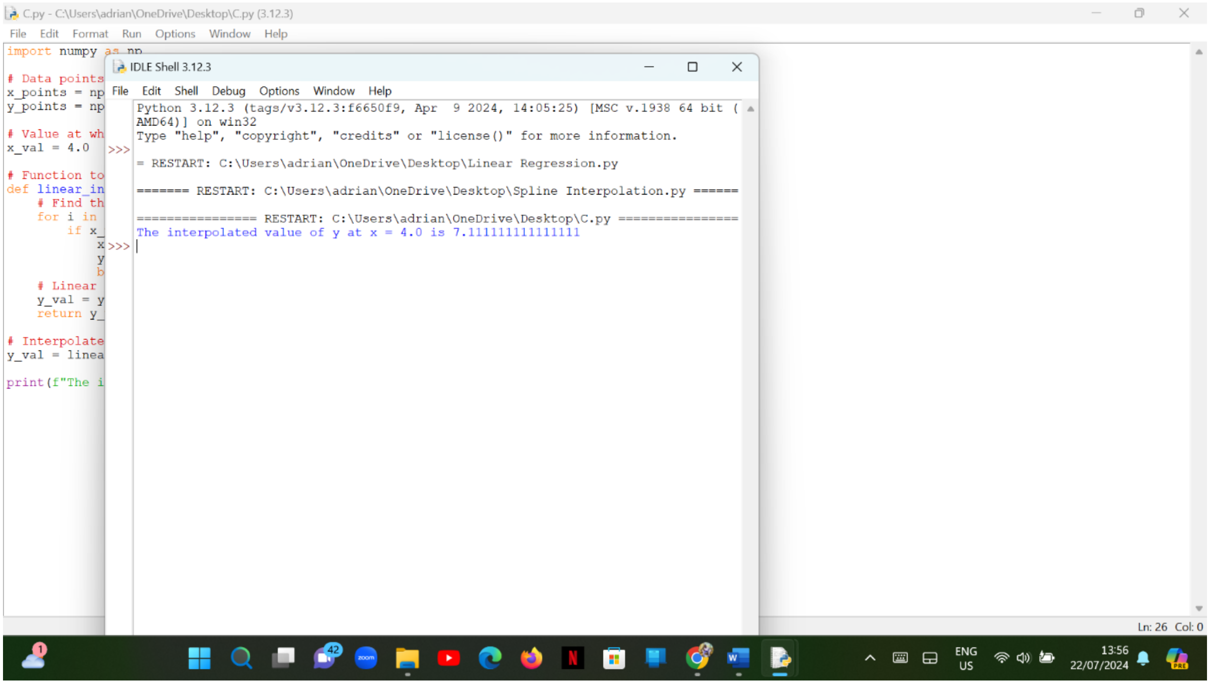
c)

code

```
import numpy as np

# Data points from the table
x_points = np.array([2.00, 4.25, 5.25, 7.81, 9.20, 10.60])
y_points = np.array([7.2, 7.1, 6.0, 5.0, 3.5, 5.0])

# Value at which we want to interpolate
x_val = 4.0

# Function to perform linear spline interpolation
def linear_interpolation(x_points, y_points, x_val):
    # Find the interval in which x_val lies
    for i in range(len(x_points) - 1):
        if x_points[i] <= x_val <= x_points[i + 1]:
            x0, x1 = x_points[i], x_points[i + 1]
            y0, y1 = y_points[i], y_points[i + 1]
            break
    # Linear interpolation formula
```

```
    y_val = y0 + (y1 - y0) * (x_val - x0) / (x1 - x0)

    return y_val


# Interpolated value of y at x_val

y_val = linear_interpolation(x_points, y_points, x_val)


print(f"The interpolated value of y at x = {x_val} is {y_val}")
```

output



d)

f(x)=0

xn+1=xn-(f(xn)/f’(xn))

f(x)=x3-0.165x2+3.993*10-4

f’(x)=3x2-0.33x

x1=xo-f(xo) /f’(xo)

f(0.05)=(0.05)3 – 0.165(0.05)2 + 3.993 * 10-4 = 1.25 * 10-4 – 0.004125 + 3.993 * 10-4= 0.00024975

f’(0.05)= 3(0.05)2 – 0.33(0.05) = 0.0075 – 0.016 = -0.009

x1 = 0.05 – 0.00024975/ -0.009 = 0.05 + 0.02775 = 0.07775

€1=|(x1 – x0)/x1| * 100 =35.68%

Iteration 2

X2=x1 – f(x1)/f’(x1)

f(x1)=-0.000137

f’(x1)=-0.00755

x2 =0.07775 – (-0.000137)/(-0.00755)=0.07775+0.01815=0.0959

€= ∣(x2 – x1)/x2∣* 100 = 18.88%

Iteration 3

X3 = x2 – (f(x2))/(f’(x2))

f(x2)= (0.0959)3 – 0.165(0.0959)2 + 3.993 * 10-4 =-0.000226

f’(0.0959)= 3(0.0959)2 – 0.33(0.0959) = 0.02755 – 0.03165 = -0.0041

x3 = 0.0959 – (-0.000226)/(-0.0041) =0.0959 + 0.05512 = 0.15102

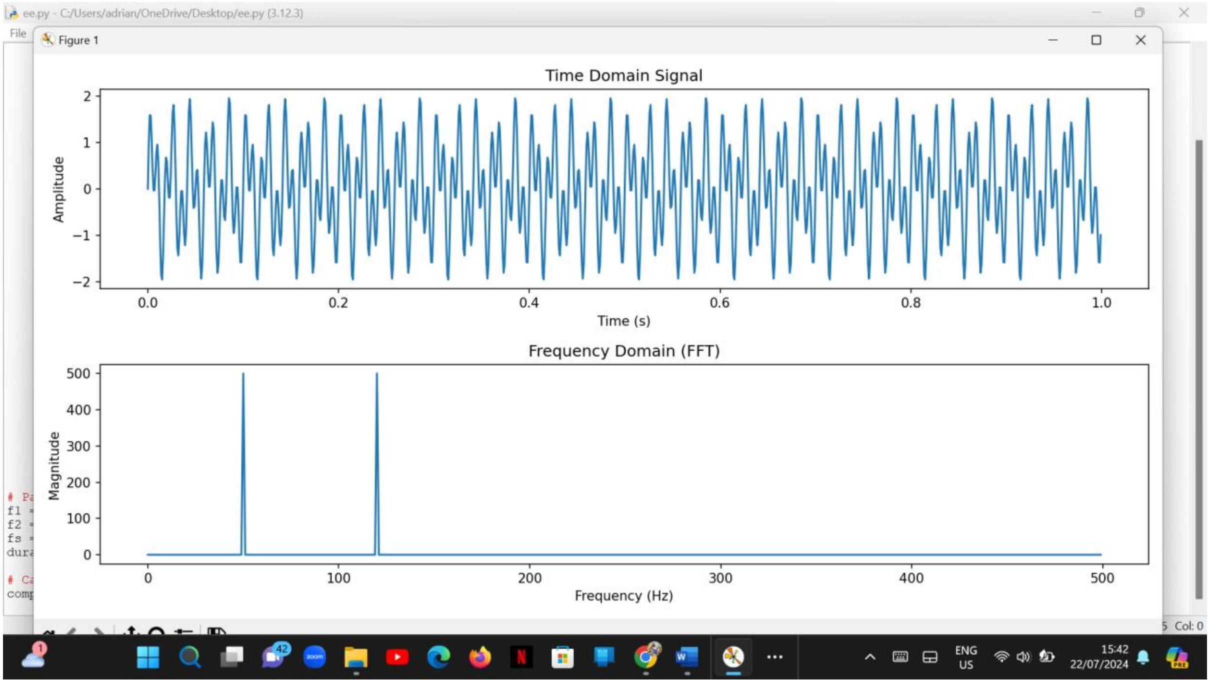
€∣(x3 – x2)/x3∣* 100= 36.52%

e)

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

def compute_fft(f1, f2, fs, duration):
    # Generate time vector
    t = np.arange(0, duration, 1/fs)

    # Generate the signal
    s_t = np.sin(2 * np.pi * f1 * t) + np.sin(2 * np.pi * f2 * t)

    # Compute FFT
    fft_result = np.fft.fft(s_t)

    # Frequency vector
    freqs = np.fft.fftfreq(len(fft_result), 1/fs)

    # Only take the positive frequencies
    positive_freqs = freqs[:len(freqs)//2]
    positive_fft = np.abs(fft_result)[:len(fft_result)//2]

    # Plotting the signal
```

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

    # Time domain plot
    plt.subplot(2, 1, 1)
    plt.plot(t, s_t)
    plt.title('Time Domain Signal')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')

    # Frequency domain plot
    plt.subplot(2, 1, 2)
    plt.plot(positive_freqs, positive_fft)
    plt.title('Frequency Domain (FFT)')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude')

    plt.tight_layout()
    plt.show()

# Parameters
f1 = 50   # Frequency 1 in Hz
f2 = 120  # Frequency 2 in Hz
fs = 1000 # Sampling frequency in Hz
duration = 1 # Duration in seconds

# Call the function
compute_fft(f1, f2, fs, duration)
```

output



f)

loop: The code runs a loop from n = 1 to n = 5.

Variable Calculation: In each iteration, it calculates x as n * 0.1.

Function Call: It calls a function myfunc2 with arguments x, 2, 3, and 7, and stores the result in z.

Output Formatting: It prints the values of x and z using fprintf, formatting x to 2 decimal places and z to 4 decimal places.

g

```
import numpy as np

import matplotlib.pyplot as plt



def trapezoidal_rule(f, a, b, n):
    """

    Calculate the integral of f from a to b using the trapezoidal rule.



    Parameters:
```

```
    f : function - The function to integrate.
    a : float - The lower limit of integration.
    b : float - The upper limit of integration.
    n : int - The number of trapezoids.

    Returns:
    float - The approximate value of the integral.
    """
    h = (b - a) / n  # Width of each trapezoid
    integral = 0.5 * (f(a) + f(b))  # Start with the first and last terms

    for i in range(1, n):
        integral += f(a + i * h)  # Add the middle terms

    integral *= h  # Multiply by the width of the trapezoids
    return integral

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

# Integration limits
```

```
a = 0  # Lower limit

b = 1  # Upper limit

n = 10  # Number of trapezoids


# Calculate the integral

result = trapezoidal_rule(f, a, b, n)

print(f"Approximate value of the integral from {a} to {b} is: {result}")


# Visualization

x = np.linspace(a, b, 100)

y = f(x)


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

plt.fill_between(x, y, color='lightblue', alpha=0.5, label='Area under curve')


# Draw trapezoids

for i in range(n):

   x0 = a + i * (b - a) / n

   x1 = a + (i + 1) * (b - a) / n

   plt.fill_between([x0, x0, x1, x1], [0, f(x0), f(x1), 0], color='orange', 
alpha=0.5)


plt.title('Trapezoidal Rule Integration')
```

plt.xlabel('x')

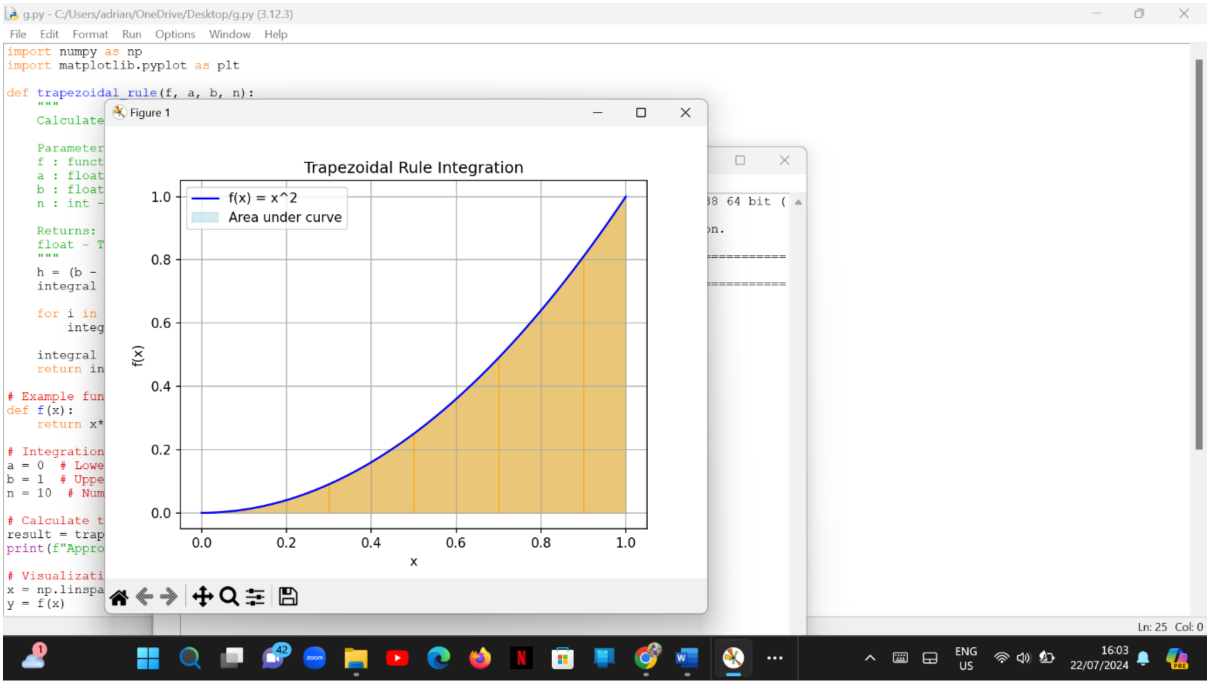
plt.ylabel('f(x)')

plt.legend()

plt.grid()

plt.show()

output



h)

original Data Points: Shown as circles at coordinates defined by the vectors x and y.

Fitted Polynomial Curve: A smooth curve representing the 4th degree polynomial that fits the data points.

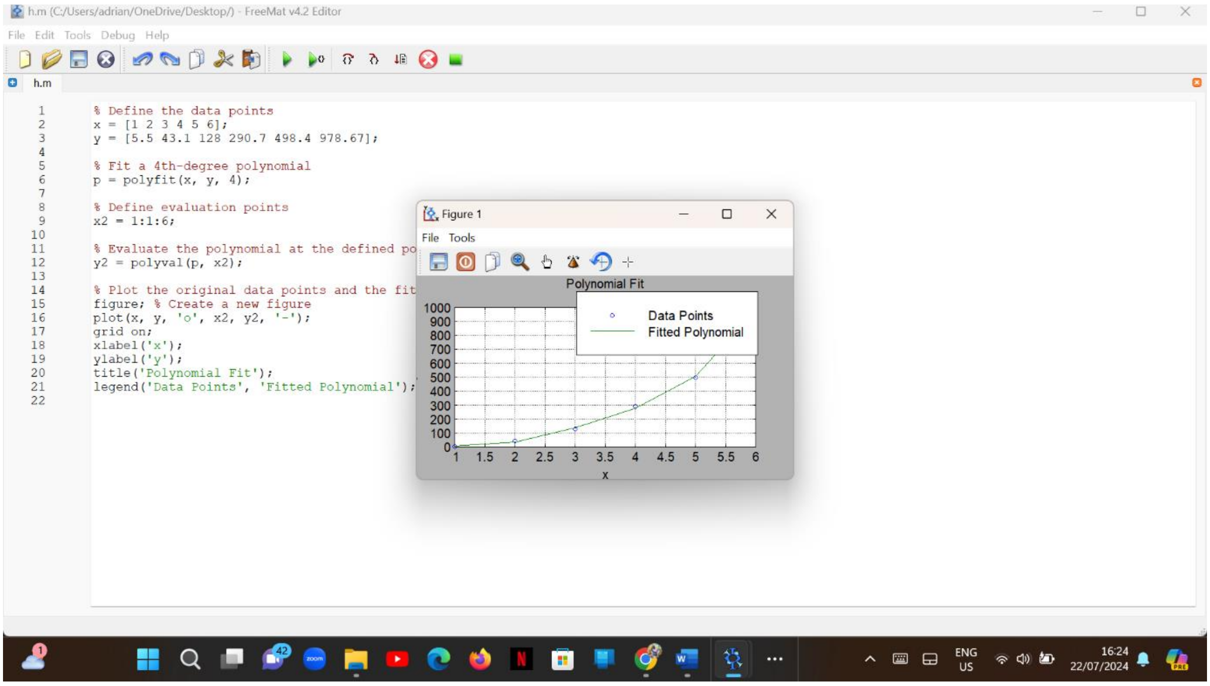
Grid: A grid overlay on the plot for better visualization.

Overall, the plot illustrates how well the polynomial fits the given data

The original data points marked as circles.

A smooth curve representing the fitted 4th-degree polynomial that goes through or near the data points.

Grid lines visible on the plot for better readability

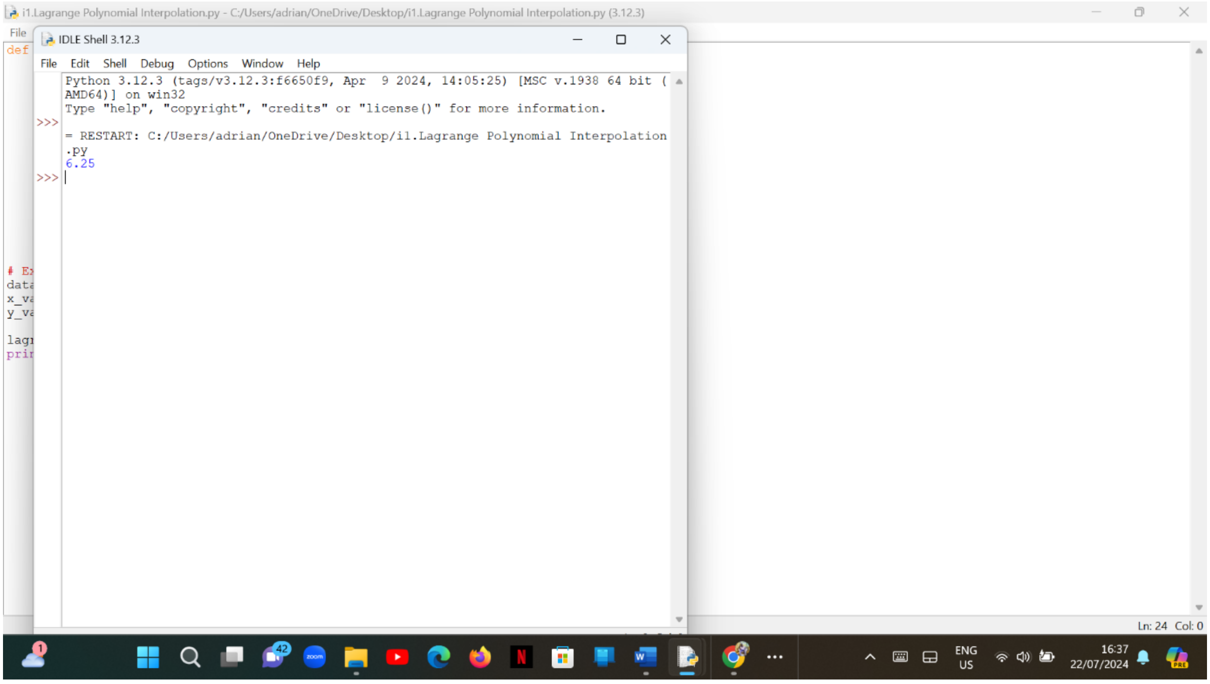


i.1) Lagrange Polynomial Interpolation

```
def lagrange_interpolation(x, y):
    def L(k, x_val):
        result = 1
        for i in range(len(x)):
            if i != k:
                result *= (x_val - x[i]) / (x[k] - x[i])
        return result

    def P(x_val):
        total = 0
```

```
    for k in range(len(x)):
        total += y[k] * L(k, x_val)
    return total

  return P

# Example usage:
data_points = [(1, 1), (2, 4), (3, 9), (4, 16)]
x_vals = [point[0] for point in data_points]
y_vals = [point[1] for point in data_points]

lagrange_poly = lagrange_interpolation(x_vals, y_vals)
print(lagrange_poly(2.5))  # Evaluate at x = 2.5
```

output

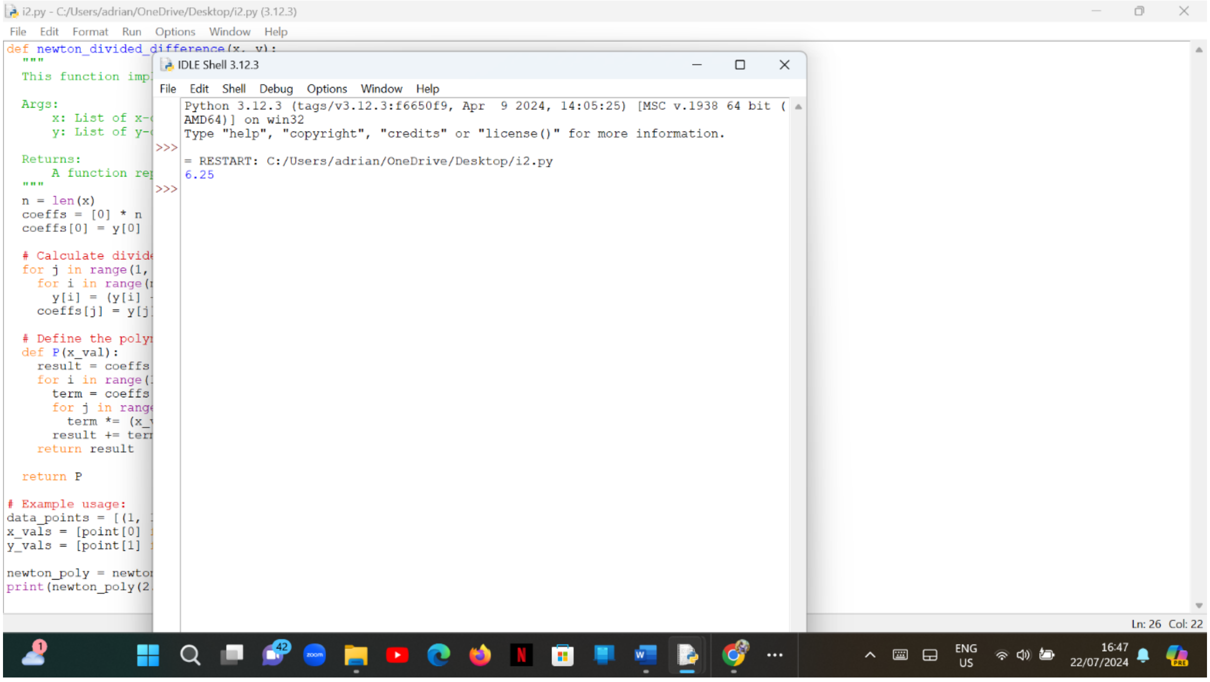


i 2)

```
def newton_divided_difference(x, y):
  """
  This function implements Newton's divided difference method for
polynomial interpolation.

  Args:
    x: List of x-coordinates of the data points.
    y: List of y-coordinates of the data points.

  Returns:
    A function representing the Newton polynomial.
  """
  n = len(x)
  coeffs = [0] * n
  coeffs[0] = y[0]

  # Calculate divided differences
  for j in range(1, n):
    for i in range(n - 1, j - 1, -1):
      y[i] = (y[i] - y[i - 1]) / (x[i] - x[i - j])
    coeffs[j] = y[j]
```

```
  # Define the polynomial function
  def P(x_val):
    result = coeffs[0]
    for i in range(1, n):
      term = coeffs[i]
      for j in range(i):
        term *= (x_val - x[j])
      result += term
    return result

  return P

# Example usage:
data_points = [(1, 1), (2, 4), (3, 9), (4, 16)]
x_vals = [point[0] for point in data_points]
y_vals = [point[1] for point in data_points]

newton_poly = newton_divided_difference(x_vals, y_vals)
print(newton_poly(2.5))  # Evaluate at x = 2.5
```

output



i 3)

Comparison of Lagrange and Newton's Methods:

Formulation:

Lagrange uses a single formula that combines all data points, while Newton builds the polynomial incrementally using divided differences.

Computational Efficiency:

Lagrange can be computationally expensive for large datasets since it recalculates the basis polynomials for each evaluation. Newton's method is generally more efficient, especially for larger datasets, as it allows for easier updates when new points are added.

Numerical Stability:

Newton's method tends to be more stable and less prone to numerical errors compared to Lagrange, particularly for closely spaced points.

Ease of Use:

Lagrange is straightforward and easy to understand, making it suitable for small datasets, while Newton's method may require a deeper understanding of divided differences.

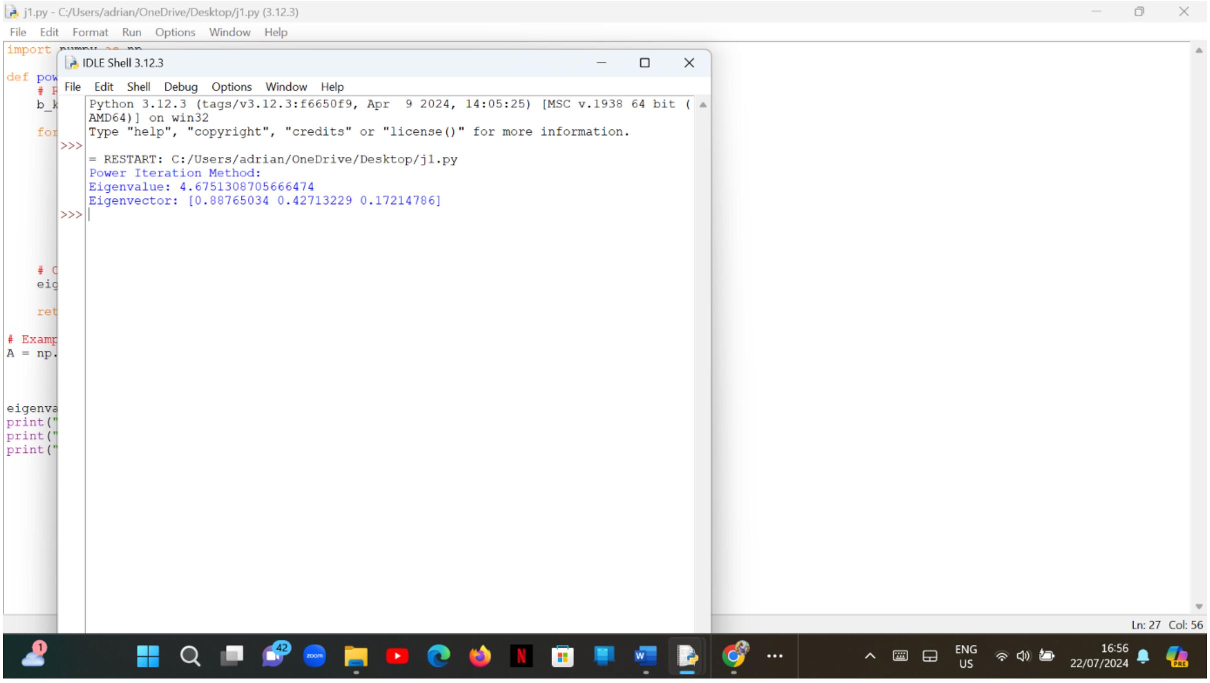
In practice, the choice between the two methods depends on the specific requirements of the problem, such as the size of the dataset and the need for numerical stability.

j1 Power Iteration Method

```
import numpy as np


def power_iteration(A, num_iterations: int = 1000):

    # Random initial vector

    b_k = np.random.rand(A.shape[1])


    for _ in range(num_iterations):

        # Calculate the matrix-by-vector product

        b_k1 = np.dot(A, b_k)


        # Calculate the norm

        b_k1_norm = np.linalg.norm(b_k1)


        # Re-normalize the vector
```

```
        b_k = b_k1 / b_k1_norm

    # Calculate the eigenvalue
    eigenvalue = np.dot(b_k.T, np.dot(A, b_k)) / np.dot(b_k.T, b_k)

    return eigenvalue, b_k

# Example usage
A = np.array([[4, 1, 1],
              [1, 3, -1],
              [1, -1, 2]])

eigenvalue_power, eigenvector_power = power_iteration(A)
print("Power Iteration Method:")
print("Eigenvalue:", eigenvalue_power)
print("Eigenvector:", eigenvector_power)
```

output



j 2) QR Algorithm

```
import numpy as np


def qr_algorithm(A, num_iterations: int = 1000):
    A_k = A.copy()


    for _ in range(num_iterations):
        Q, R = np.linalg.qr(A_k)  # QR decomposition
        A_k = R @ Q  # Update A_k for the next iteration


    eigenvalues = np.diag(A_k)  # Extract eigenvalues from the diagonal
    return eigenvalues, Q  # Return eigenvalues and the orthogonal matrix Q


# Example matrix
```

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

[1, 3, -1],

[1, -1, 2]])

Example usage

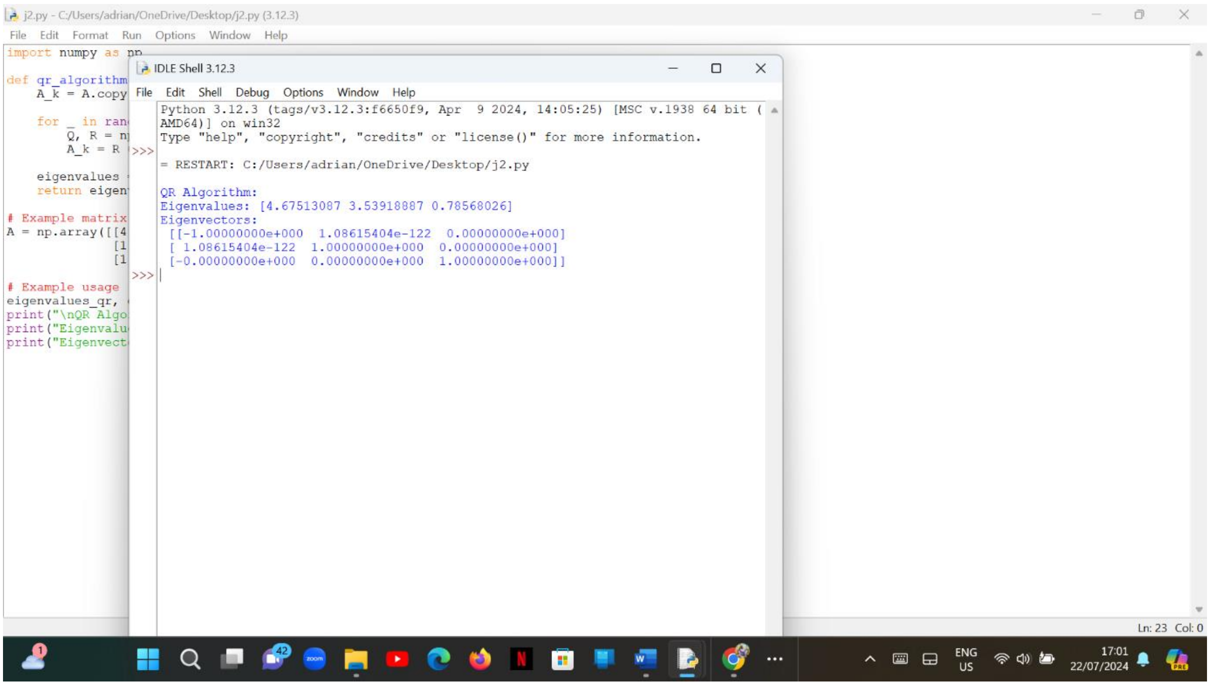
eigenvalues_qr, eigenvector_qr = qr_algorithm(A)

print("\nQR Algorithm:")

print("Eigenvalues:", eigenvalues_qr)

print("Eigenvectors:\n", eigenvector_qr)

output



J 3)

3. Comparison

After running the above implementations, you can compare the results obtained from both methods.

Power Iteration:

Finds the dominant eigenvalue and its corresponding eigenvector.

QR Algorithm:

Provides all eigenvalues and their corresponding eigenvectors.

Discussion of Differences:

Methodology:

The Power Iteration method focuses on the largest eigenvalue, while the QR Algorithm can find all eigenvalues.

Convergence:

The Power Iteration may require many iterations to converge, especially if the largest eigenvalue is not well-separated from the others. The QR Algorithm generally converges faster for all eigenvalues.

Output:

The output from the Power Iteration method is a single eigenvalue and eigenvector, while the QR Algorithm outputs a list of eigenvalues and a matrix of eigenvectors.

K

import numpy as np

```
def gradient_descent(learning_rate=0.1, initial_guess=(0, 0),
max_iterations=1000, tolerance=1e-6):
    x, y = initial_guess


    def f(x, y):
        return x**2 + y**2 - x*y + x - y + 1


    def gradient(x, y):
        df_dx = 2*x - y + 1  # Partial derivative with respect to x
        df_dy = 2*y - x - 1  # Partial derivative with respect to y
        return np.array([df_dx, df_dy])


    for i in range(max_iterations):
        grad = gradient(x, y)
        x_new = x - learning_rate * grad[0]
        y_new = y - learning_rate * grad[1]


        # Check for convergence
        if np.linalg.norm([x_new - x, y_new - y]) < tolerance:
            break


        x, y = x_new, y_new
```

```python
    return (x, y), f(x, y)


# Example usage
optimal_point, optimal_value = gradient_descent()
print("Optimal point:", optimal_point)
print("Optimal value:", optimal_value)
```

output