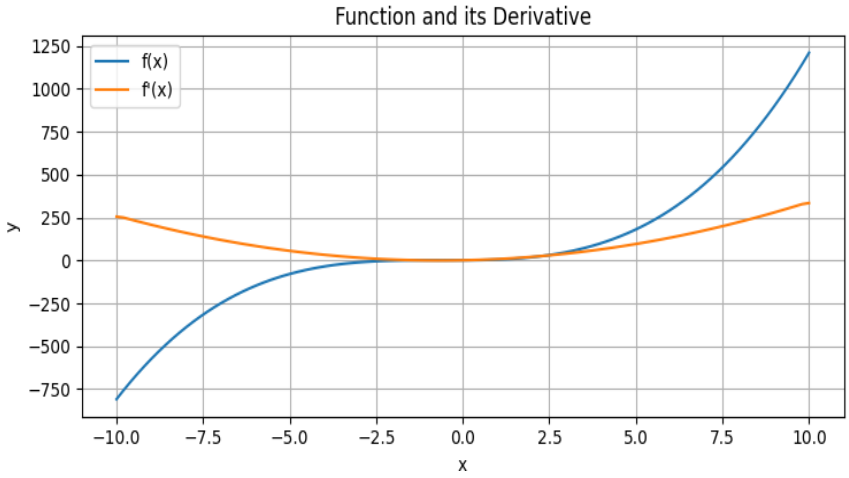
LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2
NUMERICAL ANALYSIS AND PROGRAMMING PRACTICAL WORK

b.

i. Differentiation

```
import numpy as np
import matplotlib.pyplot as plt
# Define a function
def f(x):
    return x**3 + 2*x**2 + x + 1
# Generate x values
x = np.linspace(-10, 10, 100)
# Compute y values
y = f(x)
# Compute the derivative
dy_dx = np.gradient(y, x)
# Plot the function and its derivative
plt.figure(figsize=(10, 6))
plt.plot(x, y, label='f(x)')
plt.plot(x, dy_dx, label="f'(x)")
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Function and its Derivative')
plt.grid(True)
plt.show()
```

output:



LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2

ii. Numerical integration

```
from scipy.integrate import quad
# Define the function to integrate
def integrand(x):
    return x**3 + 2*x**2 + x + 1
# Integrate from a to b
a, b = 0, 10
integral, error = quad(integrand, a, b)
print(f"Integral of the function from {a} to {b} is {integral} with error
{error}")

```

output:

```
Integral of the function from 0 to 10 is 3226.666666666667 with error
3.5823196261238387e-11
```

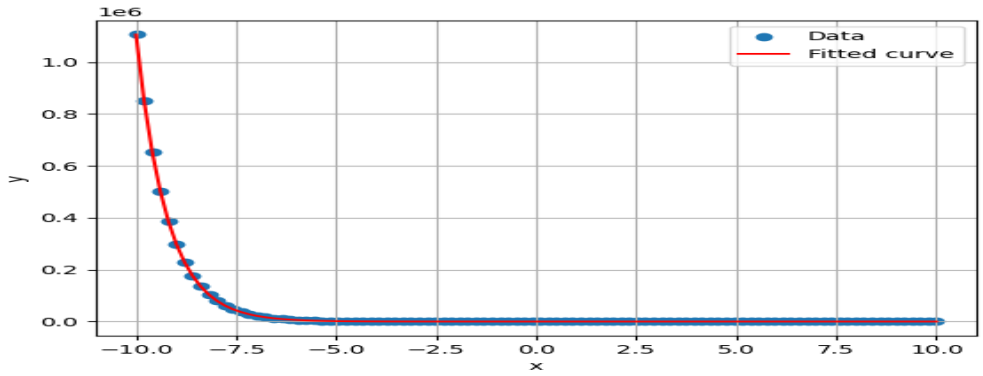
iii. Curve Fitting

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
# Define the function to fit
def func(x, a, b, c):
    return a * np.exp(-b * x) + c
# Generate example data
x_data = np.linspace(-10, 10, 100)  # Generate 100 points between -10 and 10
y_data = func(x_data, 2.5, 1.3, 0.5) + 0.2 *
np.random.normal(size=len(x_data))  # Add some noise to the data
# Perform curve fitting
popt, pcov = curve_fit(func, x_data, y_data)
# Print the optimized parameters
print("Optimized parameters:", popt)
# Generate data points for plotting the fitted curve
x_fit = np.linspace(-10, 10, 100)
y_fit = func(x_fit, *popt)
# Plot the data and the fitted curve
plt.scatter(x_data, y_data, label='Data')
plt.plot(x_fit, y_fit, label='Fitted curve', color='red')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2

Output:

```
Optimized parameters: [2.49998762 1.30000051 0.47435706]
```

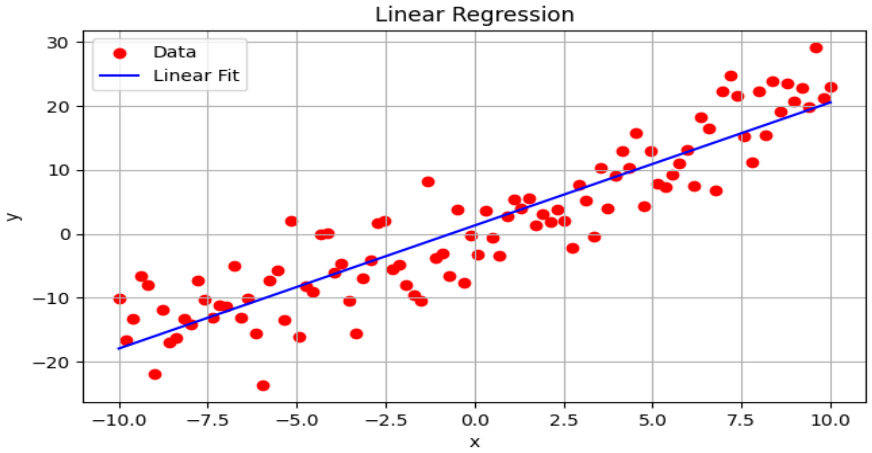


iv. Linear Regression

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
# Generate some data points with noise
np.random.seed(0)
x_data = np.linspace(-10, 10, 100)
y_data = 2*x_data + 1 + np.random.normal(0, 5, x_data.size)
# Perform linear regression
coefficients = np.polyfit(x_data, y_data, 1)  # 1 means linear
slope, intercept = coefficients
# Plot the data and the linear regression line
plt.figure(figsize=(10, 6))
plt.scatter(x_data, y_data, label='Data', color='red')
plt.plot(x_data, slope*x_data + intercept, label='Linear Fit', color='blue')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Linear Regression')
plt.grid(True)
plt.show()
print(f"Slope: {slope}, Intercept: {intercept}")
```

Output:

LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2

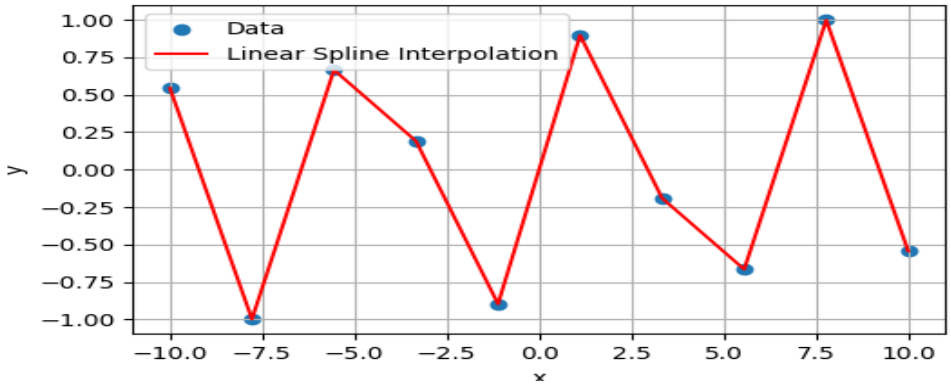


v. Spline Interpolation

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

# Generate example data
x_data = np.linspace(-10, 10, 10)  # Generate 10 points between -10 and 10
y_data = np.sin(x_data)  # Example data using the sine function

# Perform linear spline interpolation
linear_interp = interp1d(x_data, y_data, kind='linear')

# Generate data points for plotting the interpolation
x_interp = np.linspace(-10, 10, 100)
y_interp = linear_interp(x_interp)

# Plot the original data points and the linear spline interpolation
plt.scatter(x_data, y_data, label='Data')
plt.plot(x_interp, y_interp, label='Linear Spline Interpolation', color='red')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

Output:

LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2



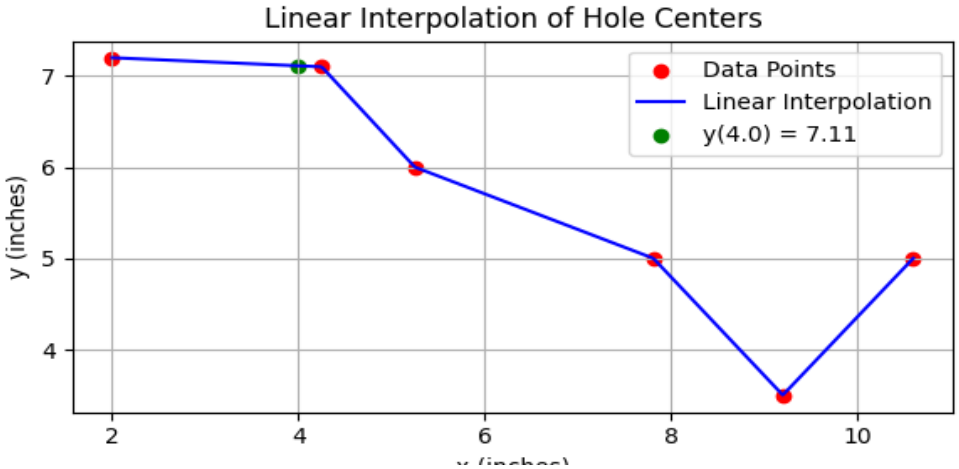
C.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
# Given data points
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])
# Create a linear interpolation function
linear_interp = interp1d(x_points, y_points, kind='linear')
# Find the y-value at x = 4.0
x_value = 4.0
y_value = linear_interp(x_value)
print(f'The value of y at x = {x_value} is {y_value:.2f}')
# Plot the data points and the linear interpolation
x_dense = np.linspace(min(x_points), max(x_points), 400)
y_dense = linear_interp(x_dense)
plt.figure(figsize=(10, 6))
plt.scatter(x_points, y_points, label='Data Points', color='red')
plt.plot(x_dense, y_dense, label='Linear Interpolation', color='blue')
plt.scatter(x_value, y_value, label=f'y({x_value}) = {y_value:.2f}',
color='green')
plt.legend()
plt.xlabel('x (inches)')
plt.ylabel('y (inches)')
plt.title('Linear Interpolation of Hole Centers')
plt.grid(True)
plt.show()
```

Output:

```
The value of y at x = 4.0 is 7.1
```

LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2



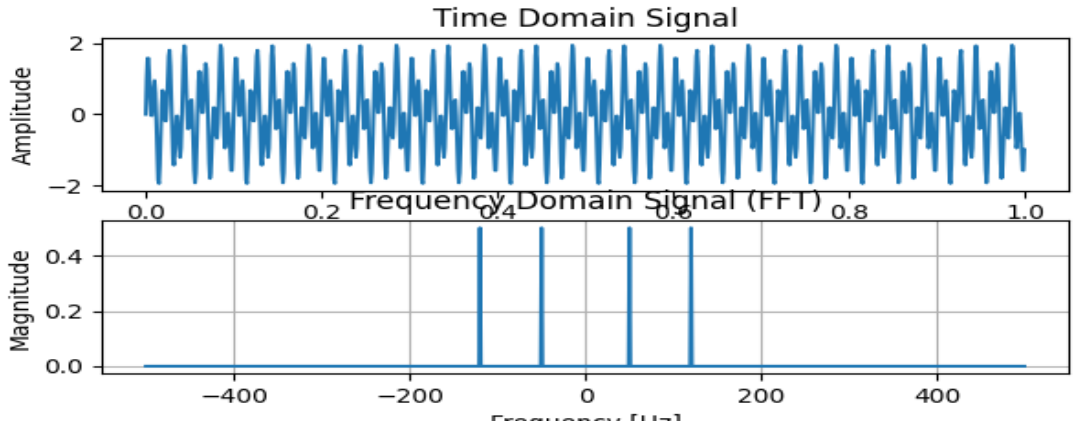
e. Fast Fourier Transform (FFT) of signal .

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

def generate_signal(f1, f2, fs, duration):
    t = np.linspace(0, duration, int(fs * duration), endpoint=False)
    s = np.sin(2 * np.pi * f1 * t) + np.sin(2 * np.pi * f2 * t)
    return t, s

def compute_fft(signal, fs):
    N = len(signal)
    fft_values = np.fft.fft(signal)
    fft_values = np.fft.fftshift(fft_values)  # Shift zero frequency component to
center
    frequencies = np.fft.fftfreq(N, 1/fs)
    frequencies = np.fft.fftshift(frequencies)  # Shift zero frequency component
to center

    # Compute the magnitude of the FFT and normalize
    magnitude = np.abs(fft_values) / N

    return frequencies, magnitude

# Signal parameters
f1 = 50  # Frequency of the first sine wave
f2 = 120  # Frequency of the second sine wave
fs = 1000  # Sampling frequency
duration = 1  # Duration in seconds
```

LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2

```
# Generate the signal
t, signal = generate_signal(f1, f2, fs, duration)

# Compute the FFT
frequencies, magnitude = compute_fft(signal, fs)

# Plot the signal
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(t, signal)
plt.title('Time Domain Signal')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')

# Plot the FFT magnitude spectrum
plt.subplot(2, 1, 2)
plt.plot(frequencies, magnitude)
plt.title('Frequency Domain Signal (FFT)')
plt.xlabel('Frequency [Hz]')
plt.ylabel('Magnitude')
plt.grid()
plt.show()
```

Output:



g. Trapezoidal rule of integration

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

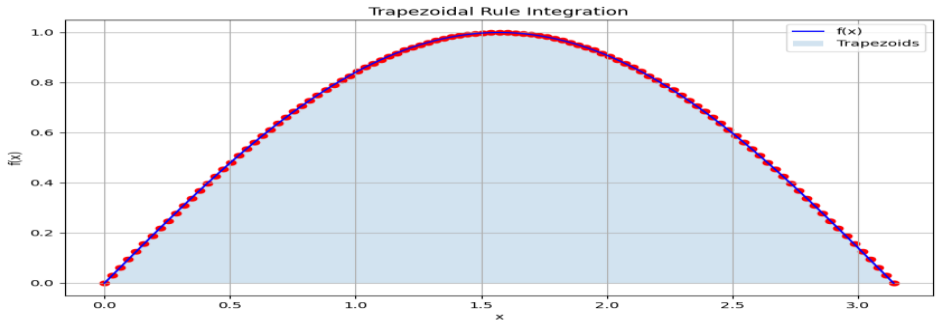
LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2

```
# Define the function to integrate
def f(x):
    return np.sin(x)

# Implement the trapezoidal rule
def trapezoidal_rule(func, a, b, n):
    x = np.linspace(a, b, n+1)
    y = func(x)
    h = (b - a) / n
    integral = (h / 2) * (y[0] + 2 * np.sum(y[1:n]) + y[n])
    return integral

# Parameters for the integration
a = 0          # Lower limit
b = np.pi      # Upper limit
n = 100        # Number of sub-intervals

# Compute the integral using the trapezoidal rule
integral = trapezoidal_rule(f, a, b, n)
print(f'The approximate value of the integral is {integral:.6f}')

# Plot the function and the trapezoids
x = np.linspace(a, b, 1000)
y = f(x)
x_trap = np.linspace(a, b, n+1)
y_trap = f(x_trap)

plt.figure(figsize=(10, 6))
plt.plot(x, y, 'b', label='f(x)')
plt.fill_between(x_trap, y_trap, alpha=0.2, label='Trapezoids')
plt.scatter(x_trap, y_trap, color='red')
plt.title('Trapezoidal Rule Integration')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()
```

Output:

```
The approximate value of the integral is 1.999836
```

LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2



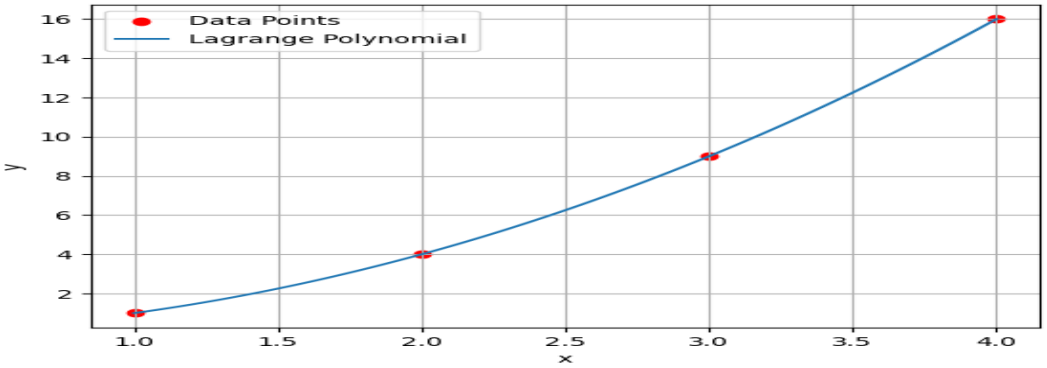
i.

I. Lagrange polynomial method

```
import numpy as np
import matplotlib.pyplot as plt
# Define the data points
data_points = np.array([(1, 1), (2, 4), (3, 9), (4, 16)])
x_points, y_points = data_points[:, 0], data_points[:, 1]
# Define the Lagrange polynomial interpolation function
def lagrange_interpolation(x, x_points, y_points):
    total = 0
    n = len(x_points)
    for i in range(n):
        xi, yi = x_points[i], y_points[i]
        term = yi
        for j in range(n):
            if i != j:
                xj = x_points[j]
                term *= (x - xj) / (xi - xj)
        total += term
    return total
# Generate data points for plotting the Lagrange polynomial
x = np.linspace(1, 4, 100)
y = lagrange_interpolation(x, x_points, y_points)
# Plot the data points and the Lagrange polynomial
plt.scatter(x_points, y_points, label='Data Points', color='red')
plt.plot(x, y, label='Lagrange Polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

Output:

LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2



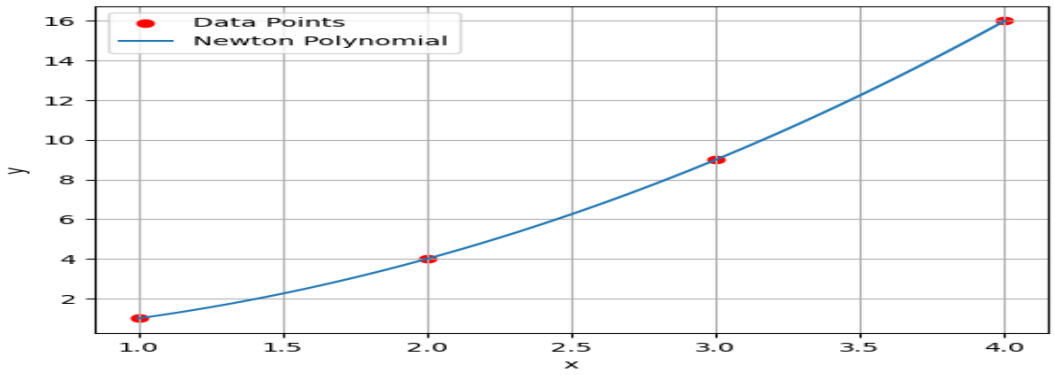
II. Newton's divided difference method

```
import numpy as np
import matplotlib.pyplot as plt
# Define the data points
data_points = np.array([(1, 1), (2, 4), (3, 9), (4, 16)])
x_points, y_points = data_points[:, 0], data_points[:, 1]
# Function to compute the coefficients of the Newton's divided difference
polynomial
def newton_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, :]
# Function to evaluate the Newton's divided difference polynomial at a given
value x
def newton_polynomial(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
# Compute the coefficients of the Newton's divided difference polynomial
coef = newton_divided_diff(x_points, y_points)
# Generate data points for plotting the Newton polynomial
x = np.linspace(1, 4, 100)
y = [newton_polynomial(coef, x_points, xi) for xi in x]
```

LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2

```
# Plot the data points and the Newton polynomial
plt.scatter(x_points, y_points, label='Data Points', color='red')
plt.plot(x, y, label='Newton Polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

Output:



J. I) Power Iteration method

```
import numpy as np
# Define the matrix A
A = np.array([[4, 1, 1], [1, 3, -1], [1, -1, 2]])
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
    return b_k1_norm, b_k
# Compute the dominant eigenvalue and eigenvector using power iteration
eigenvalue, eigenvector = power_iteration(A, 1000)
# Print the results
print("Dominant eigenvalue:", eigenvalue)
print("Corresponding eigenvector:", eigenvector)
```

LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2

Output:

```
Dominant eigenvalue: 4.675130870566647
Corresponding eigenvector: [0.88765034 0.42713229 0.17214786]
```

ii. QR Algorithm

```
import numpy as np
# Define the matrix A
A = np.array([[4, 1, 1], [1, 3, -1], [1, -1, 2]])

def qr_algorithm(A, num_iterations: int):
    n = A.shape[0]
    Q = np.eye(n)
    R = A.copy()

    for _ in range(num_iterations):
        Q_iter, R_iter = np.linalg.qr(R @ Q)
        Q = Q @ Q_iter
        R = R_iter @ R
        # Normalize Q and R to prevent overflow
        norm_factor = np.linalg.norm(R)
        R /= norm_factor
        Q /= norm_factor
    eigenvalues = np.diag(R)
    eigenvectors = Q
    return eigenvalues, eigenvectors
# Compute the eigenvalues and eigenvectors using QR algorithm
eigenvalues_qr, eigenvectors_qr = qr_algorithm(A, 1000)
# Print the results
print("Eigenvalues (QR Algorithm):", eigenvalues_qr)
print("Eigenvectors (QR Algorithm):")
print(eigenvectors_qr)
```

Output:

```
Eigenvalues (QR Algorithm): [0.88765034 0.         0.        ]
Eigenvectors (QR Algorithm):
[[ 0.88765034  0.23319198 -0.39711255]
 [ 0.42713229 -0.73923874  0.52065737]
 [ 0.17214786  0.63178128  0.75578934]]
```

LILIAN IRUSA
INTE/MG/2618/09/22
INTE 316 CAT 2
K. Gradient Descent method

```
import numpy as np

def gradient_descent(f, grad_f, x0, y0, learning_rate=0.1, max_iter=1000, tol=1e-
6):
    x, y = x0, y0
    for _ in range(max_iter):
        grad_x, grad_y = grad_f(x, y)
        x_new = x - learning_rate * grad_x
        y_new = y - learning_rate * grad_y
        if np.sqrt((x_new - x)**2 + (y_new - y)**2) < tol:
            break
        x, y = x_new, y_new
    return x, y

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

# Initial guess
x0, y0 = 0, 0

# Perform gradient descent
min_x, min_y = gradient_descent(f, grad_f, x0, y0)
print(f"Minimum value of f(x,y) is at x = {min_x:.6f}, y = {min_y:.6f}")
```

Output:

```
Minimum value of f(x,y) is at
X=33491589411587785379987039163965641534685926348415385949207246610219414292552992
4239360.000000,
Y=14187264942199802700556221351496107229301598853228385983814519235710269371038475
34886912.000000
```