**NUMERICAL ANALYSIS AND PROGRAMMING**

b.

1. 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')

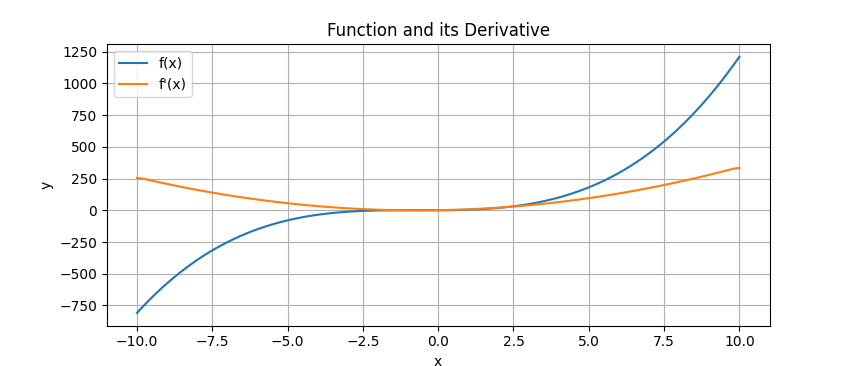
plt.ylabel('y')

plt.title('Function and its Derivative')

plt.grid(True)

plt.show()

output:



1. Numerical integration
2. from scipy.integrate import quad
3. # Define the function to integrate
4. def integrand(x):
5. return x\*\*3 + 2\*x\*\*2 + x + 1
6. # Integrate from a to b
7. a, b = 0, 10
8. integral, error = quad(integrand, a, b)
9. 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

1. 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')

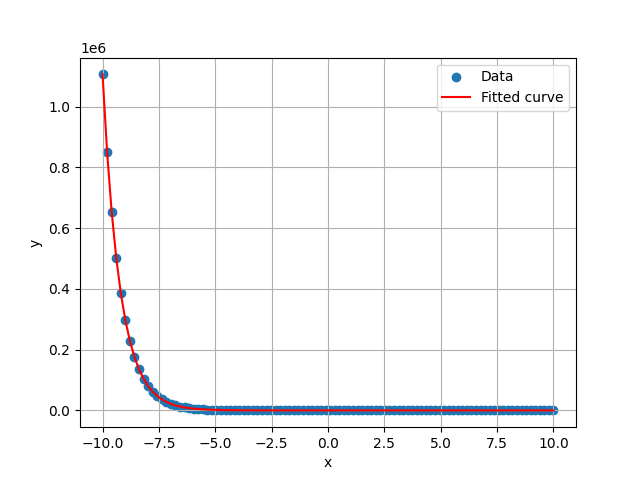
plt.ylabel('y')

plt.legend()

plt.grid(True)

plt.show()

Output:



1. Linear Regression

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')

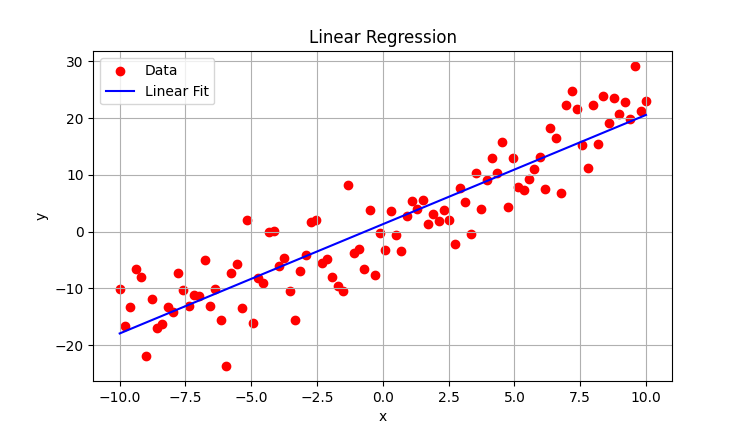
plt.title('Linear Regression')

plt.grid(True)

plt.show()

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

Output:



1. 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')

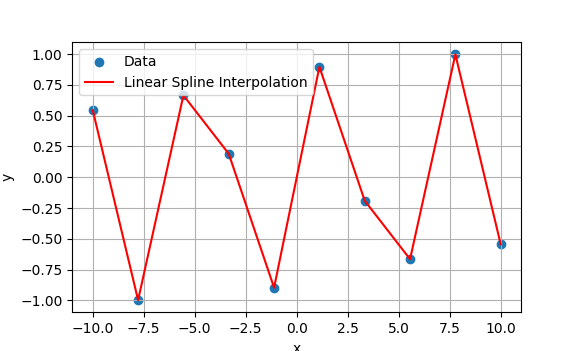
plt.ylabel('y')

plt.legend()

plt.grid(True)

plt.show()

Output:



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)')

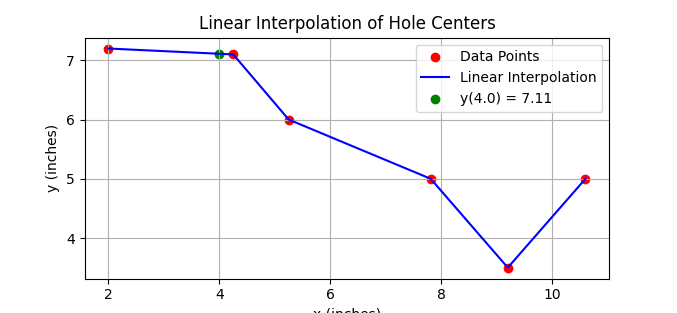
plt.ylabel('y (inches)')

plt.title('Linear Interpolation of Hole Centers')

plt.grid(True)

plt.show()

Output:



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

# 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)')

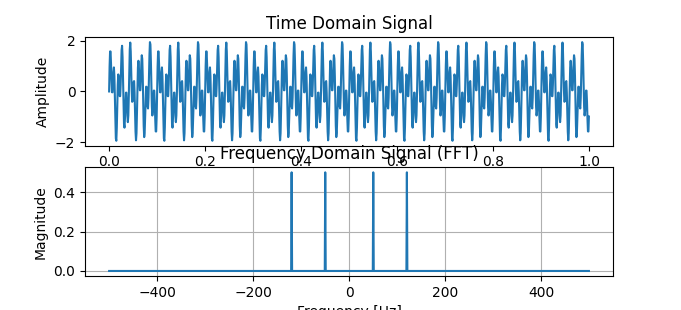
plt.xlabel('Frequency [Hz]')

plt.ylabel('Magnitude')

plt.grid()

plt.show()

Output:



g.

import numpy as np

import matplotlib.pyplot as plt

# 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')

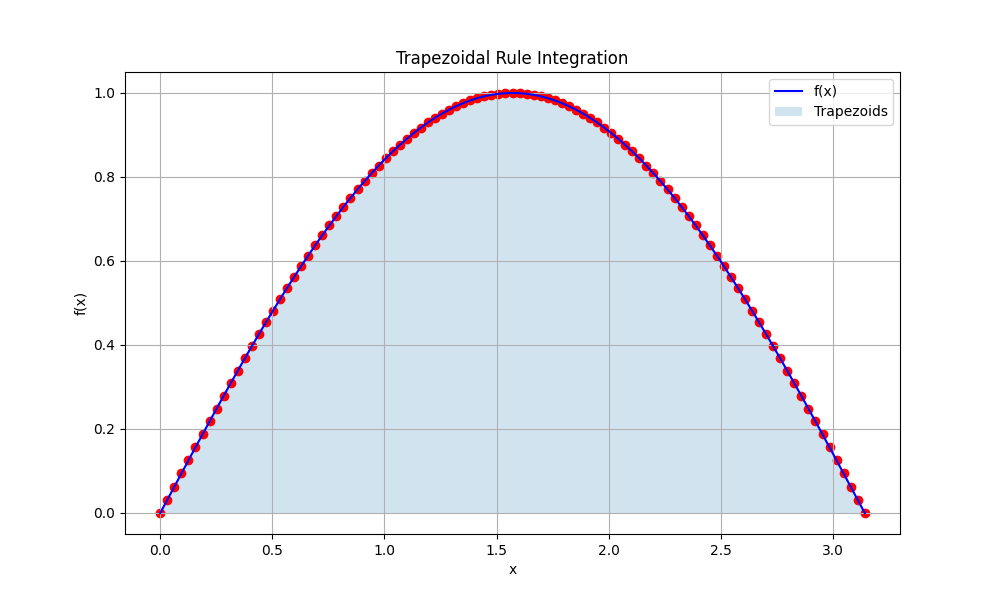
plt.ylabel('f(x)')

plt.legend()

plt.grid(True)

plt.show()

Output:



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')

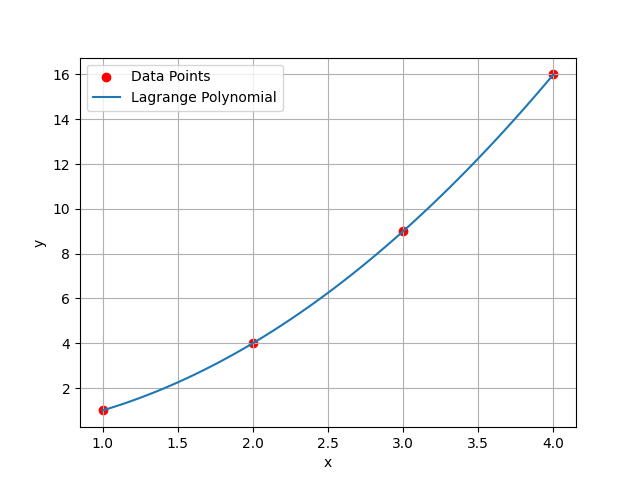
plt.ylabel('y')

plt.legend()

plt.grid(True)

plt.show()

Output:



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]

# 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')

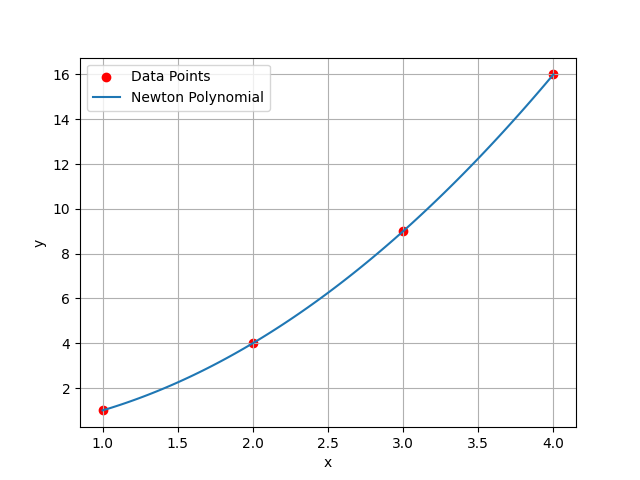
plt.ylabel('y')

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)

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]]

K.

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=334915894115877853799870391639656415346859263484153859492072466102194142925529924239360.000000,

Y=1418726494219980270055622135149610722930159885322838598381451923571026937103847534886912.000000