[30/07, 11:47] Godfather: import numpy as np

import matplotlib.pyplot as plt

from scipy.optimize import curve\_fit

# Define the model function

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

return a \* x\*\*2 + b \* x + c

# Sample data

x\_data = np.array([1, 2, 3, 4, 5])

y\_data = np.array([1, 4, 9, 16, 25])

# Fit the model to the data

params, covariance = curve\_fit(model, x\_data, y\_data)

# Plot the data and the fit

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}")

[30/07, 11:47] Godfather: import sympy as sp

# Define the symbol

x = sp.symbols('x')

# Define the function

f = x\*\*2 - x - 2

# Differentiate the function

f\_prime = sp.diff(f, x)

print(f"The derivative of {f} is {f\_prime}")

[30/07, 11:47] Godfather: import numpy as np

import matplotlib.pyplot as plt

from scipy.interpolate import splrep, splev

# Sample data

x = np.array([1, 2, 3, 4, 5])

y = np.array([1, 4, 9, 16, 25])

# Perform spline interpolation

spline = splrep(x, y)

x\_new = np.linspace(1, 5, 100)

y\_new = splev(x\_new, spline)

# Plot the data and the spline interpolation

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

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

plt.legend()

plt.show()

[30/07, 11:47] Godfather: import scipy.integrate as spi

import numpy as np

# Define the function

def f(x):

return x\*\*2 - x - 2

# Integrate the function from 1 to 3

integral, error = spi.quad(f, 1, 3)

print(f"The integral of the function from 1 to 3 is {integral} with an error of {error}")

[30/07, 11:47] Godfather: import numpy as np

import matplotlib.pyplot as plt

# Signal parameters

f1 = 50 # Frequency of first sine wave

f2 = 120 # Frequency of second sine wave

fs = 1000 # Sampling frequency

T = 1 # Duration in seconds

# Time vector

t = np.linspace(0, T, fs\*T, endpoint=False)

# Signal definition

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

# Compute FFT

S = np.fft.fft(s)

# Compute frequencies

frequencies = np.fft.fftfreq(len(S), 1/fs)

# Only take the positive frequencies

positive\_frequencies = frequencies[:len(frequencies)//2]

positive\_S = np.abs(S[:len(S)//2]) # Magnitude of FFT

# Plot the signal

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

# Plot time domain signal

plt.subplot(2, 1, 1)

plt.plot(t, s)

plt.title('Time Domain Signal')

plt.xlabel('Time [s]')

plt.ylabel('Amplitude')

# Plot frequency domain signal

plt.subplot(2, 1, 2)

plt.plot(positive\_frequencies, positive\_S)

plt.title('Frequency Domain Signal')

plt.xlabel('Frequency [Hz]')

plt.ylabel('Magnitude')

plt.grid()

plt.tight\_layout()

plt.show()

[30/07, 11:47] Godfather: import numpy as np

def gradient\_descent(grad\_f, initial\_guess, learning\_rate, num\_iterations):

x, y = initial\_guess

for \_ in range(num\_iterations):

grad\_x, grad\_y = grad\_f(x, y)

x -= learning\_rate \* grad\_x

y -= learning\_rate \* grad\_y

return x, y

def grad\_f(x, y):

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

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

return np.array([df\_dx, df\_dy])

# Initial guess

initial\_guess = (0, 0)

# Learning rate

learning\_rate = 0.1

# Number of iterations

num\_iterations = 100

# Perform gradient descent

min\_x, min\_y = gradient\_descent(grad\_f, initial\_guess, learning\_rate, num\_iterations)

print(f"Minimum value found at x = {min\_x}, y = {min\_y}")

[30/07, 11:47] Godfather: import numpy as np

import matplotlib.pyplot as plt

# Define the data points

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

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

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

"""

Perform Lagrange interpolation for the given data points.

Parameters:

x\_points : array-like

The x coordinates of the data points.

y\_points : array-like

The y coordinates of the data points.

x : float

The x value at which to evaluate the interpolating polynomial.

Returns:

float

The interpolated value at x.

"""

n = len(x\_points)

total = 0

for i in range(n):

term = y\_points[i]

for j in range(n):

if i != j:

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

total += term

return total

# Define the x values for plotting the polynomial

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

y\_values = [lagrange\_interpolation(x\_points, y\_points, x) for x in x\_values]

# Plot the data points and the interpolating polynomial

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

plt.plot(x\_values, y\_values, label='Lagrange polynomial')

plt.xlabel('x')

plt.ylabel('y')

plt.title('Lagrange Polynomial Interpolation')

plt.legend()

plt.grid(True)

plt.show()