**Numerical Optimisation codes**

**Anshika Singh**

**16010**

**Code 1**

import sympy as sp

x1,x2= sp.symbols('x1 x2')

function= input("Enter the function")

gradient= [sp.diff(function,x1), sp.diff(function,x2)]

hassien = [[sp.diff(gradient [0],x1), sp.diff(gradient [0],x2)],[sp.diff(gradient [0],x1), sp.diff(gradient [0],x2)]]

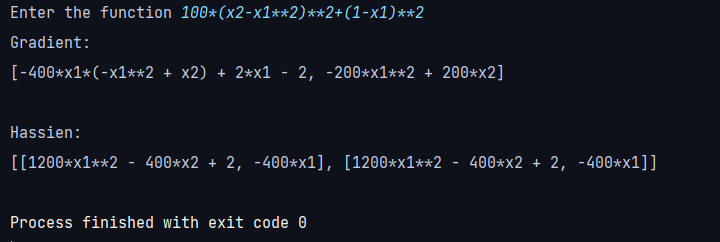
print("Gradient:")

print(gradient)

print ("\nHassien:")

print (hassien)

**Output:**

****

**Code 2**

import numpy as np

def objective\_function(x):

return x\*\*2 + 4\*x + 4

def gradient(x):

return 2\*x + 4

def line\_search(initial\_x, learning\_rate, epsilon):

x = initial\_x

iteration = 0

while True:

gradient\_x = gradient(x)

new\_x = x - learning\_rate \* gradient\_x

# Check for convergence

if abs(new\_x - x) < epsilon:

break

x = new\_x

iteration += 1

return x, objective\_function(x), iteration

# Initial parameters

initial\_x = 0.0

learning\_rate = 0.1

epsilon = 1e-6

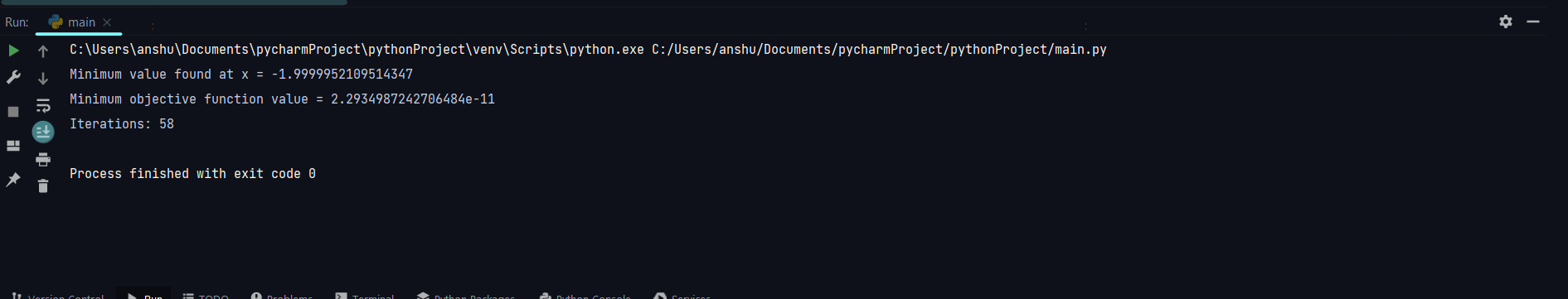
result\_x, result\_min, iterations = line\_search(initial\_x, learning\_rate, epsilon)

print(f"Minimum value found at x = {result\_x}")

print(f"Minimum objective function value = {result\_min}")

print(f"Iterations: {iterations}")

**Output:**

****

**Code 3:**

import matplotlib.pyplot as plt

# Define the constraints

# Coefficients of the form: Ax + By <= C

constraints = [

(2, 1, 20),

(4, -5, 10)

]

# Define the objective function coefficients

# Z = cx + dy

c = 3

d = 4

# Create a function to plot the constraints

def plot\_constraints():

plt.figure()

for constraint in constraints:

A, B, C = constraint

x = [0, C / A] # considering x = 0 and x = C/A to plot lines

y = [(C - A \* xi) / B for xi in x]

plt.plot(x, y, label=f"{A}x + {B}y <= {C}")

plt.xlabel('x')

plt.ylabel('y')

plt.axhline(0, color='black',linewidth=0.5)

plt.axvline(0, color='black',linewidth=0.5)

plt.legend()

plt.show()

# Function to calculate Z value

def calculate\_Z(x, y):

return c \* x + d \* y

# Finding corner points and calculating Z values

def find\_optimal\_solution():

corner\_points = []

for constraint in constraints:

A, B, C = constraint

x = 0 if A == 0 else C / A

y = 0 if B == 0 else C / B

corner\_points.append((x, y))

max\_Z = float('-inf')

optimal\_point = None

for point in corner\_points:

x, y = point

current\_Z = calculate\_Z(x, y)

if current\_Z > max\_Z:

max\_Z = current\_Z

optimal\_point = point

return optimal\_point, max\_Z

# Plot the constraints

plot\_constraints()

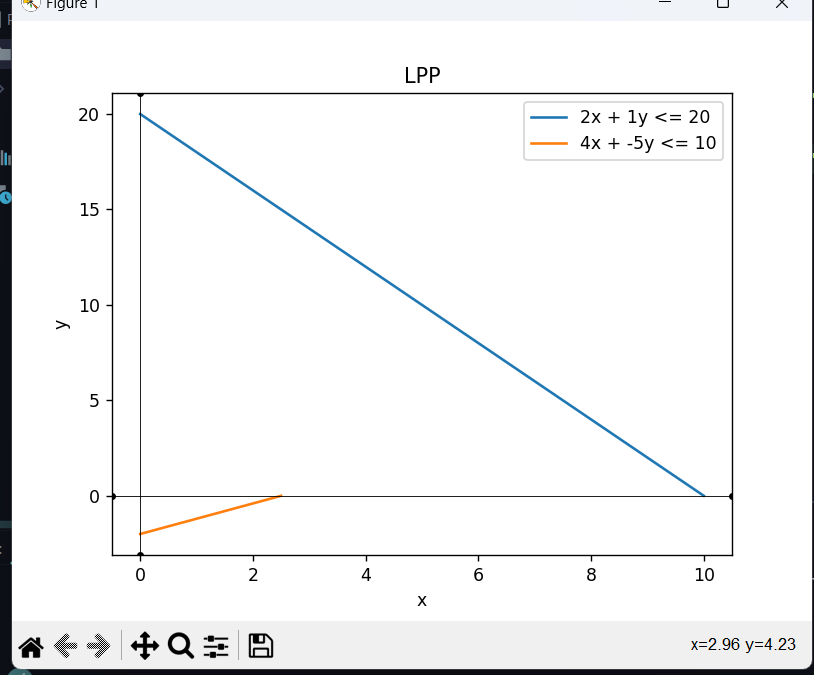
# Calculate the optimal solution

optimal\_solution, max\_Z\_value = find\_optimal\_solution()

# Display the optimal solution

print(f"The optimal solution is at point: {optimal\_solution} with Z value: {max\_Z\_value}")

**Output:**



**Code 4**

import numpy as np

import matplotlib.pyplot as plt

# Define the function f(x)

def objective\_function(x):

return -10 \* np.cos(np.pi \* x - 2.2) + (x + 1.5) \* x

# Generate x values

x = np.linspace(-5, 5, 20)

print(x)

y = objective\_function(x)

print(y)

plt.plot(x, y, label='f(x) = -10Cos(pi x - 2.2) + (x + 1.5) \* x')

plt.xlabel('x')

plt.ylabel('f(x)')

plt.title(' Function f(x)')

plt.grid(True)

min\_y = min(y)

min\_x = x[np.argmin(y)]

plt.scatter(min\_x, min\_y, color='blue', label=f'Minimum: ({min\_x}, {min\_y})')

plt.legend()

plt.show()

print("Global optimal solution is", min\_x)

print("Optimal function value is”, min\_y)

**Output:**

**Code 5:**

import numpy as np

from scipy.optimize import differential\_evolution

def objective\_function(x):

return -10 \* np.cos(np.pi \* x - 2.2) + (x + 1.5) \* x

bounds = [(-10, 10)]

result = differential\_evolution(objective\_function, bounds)

min\_x = result.x

global\_min\_val = result.fun

print("global min x: ",min\_x)

print("Global Optimal Solution:")

print(f"x = {min\_x[0]}")

print(f"f(x) = {global\_min\_val}")

**Output:**

**Code 6:**

from sympy import symbols, diff, solve, Matrix

x, y, l = symbols('x y lambda')

f = x\*\*2 + y\*\*2

g = x + y - 1

# Define the Lagrangian

L = f - l \* g

# Compute partial derivatives

partials = [diff(L, var) for var in (x, y, l)]

# Solve the system of equations

solution = solve(partials, (x, y, l), dict=True)[0]

# Extract the optimal values

optimal\_x = solution[x]

optimal\_y = solution[y]

# Compute the Hessian matrix

# Compute the Hessian matrix using a list of lists

hessian\_list = []

# Iterate over var2

for var2 in (x, y, l):

# Initialize a row for var2

row = []

# Iterate over var1

for var1 in (x, y, l):

# Calculate the second-order partial derivative and append to the row

row.append(diff(L.diff(var1), var2))

# Append the row to the Hessian list

hessian\_list.append(row)

# Create an instance of the Matrix class from the list of lists

hessian\_matrix = Matrix(hessian\_list)

# Display the Hessian matrix

print(hessian\_matrix)

hessian\_determinant = hessian\_matrix.det()

if hessian\_determinant > 0:

print("Stationary point is a local minimum.")

elif hessian\_determinant < 0:

print("Stationary point is a local maximum.")

else:6

print("Second-order test inconclusive (saddle point or test fails).")

# Display the result

print("Optimal solution:")

print(f"x: {optimal\_x}")

print(f"y: {optimal\_y}")

**Output:**