*Theoretical Model in Computing Lab*

**LAB 6: OPTIMIZATION**

**Instructor: Dr. Phuong Vo**

Do

**Q1.**a) b)

**A notebook with writing on it

Description automatically generatedA notebook with writing on it

Description automatically generated**

**A notebook with a graph on it

Description automatically generated**

**A notebook with a pen and a marker

Description automatically generated**

**c)**

#Code by Nguyen Minh Duc\_ITITIU21045\_Lab06TMC\_EX1c

import cvxpy as cp

# Define the variables

x = cp.Variable()

y = cp.Variable()

# Define the objective function to maximize

objective = cp.Maximize(6\*x + 8\*y)

# Define the constraints

constraints = [

    5\*x + 2\*y <= 40,

    6\*x + 6\*y <= 60,

    2\*x + 4\*y <= 32,

    x >= 0,

    y >= 0

]

# Formulate the problem

problem = cp.Problem(objective, constraints)

# Solve the problem

problem.solve()

# Display the results

print("Optimal value:", problem.value)

print("Optimal x:", x.value)

print("Optimal y:", y.value)

A number on a black background

Description automatically generated

**Q2**a)

#Code by Nguyen Minh Duc\_ITITIU21045\_Lab06TMC\_EX2a

def f(x, y):

    return 3.5\*x + 2\*y + x\*\*2 - x\*\*4 + 2\*x\*y - y\*\*2

x = 0

y = 0

h = 0.1

for i in range(3):

    df\_dx = (f(x + h, y) - f(x - h, y)) / (2 \* h)

    df\_dy = (f(x, y + h) - f(x, y - h)) / (2 \* h)

    x = x + h \* df\_dx

    y = y + h \* df\_dy

    value = f(x, y)

    print(f"Iteration {i + 1}: x = {x:.3f}, y = {y:.3f}, f(x, y) = {value:.3f}")

A group of white letters and numbers

Description automatically generated

b)

#Code by Nguyen Minh Duc\_ITITIU21045\_Lab06TMC\_EX2b

import numpy as np

import matplotlib.pyplot as plt

def f(x, y):

    return 3.5\*x + 2\*y + x\*\*2 - x\*\*4 + 2\*x\*y - y\*\*2

def gradient(x, y, h):

    df\_dx = (f(x + h, y) - f(x - h, y)) / (2 \* h)

    df\_dy = (f(x, y + h) - f(x, y - h)) / (2 \* h)

    return df\_dx, df\_dy

def gradient\_ascent(init\_x, init\_y, step\_size, num\_iter):

    x = init\_x

    y = init\_y

    values = []

    for i in range(num\_iter):

        df\_dx, df\_dy = gradient(x, y, h)

        x = x + step\_size \* df\_dx

        y = y + step\_size \* df\_dy

        value = f(x, y)

        values.append(value)

    return values

init\_x = 0

init\_y = 0

step\_sizes = [0.01, 0.1, 0.5, 1]

num\_iter = 100

for h in step\_sizes:

    values = gradient\_ascent(init\_x, init\_y, h, num\_iter)

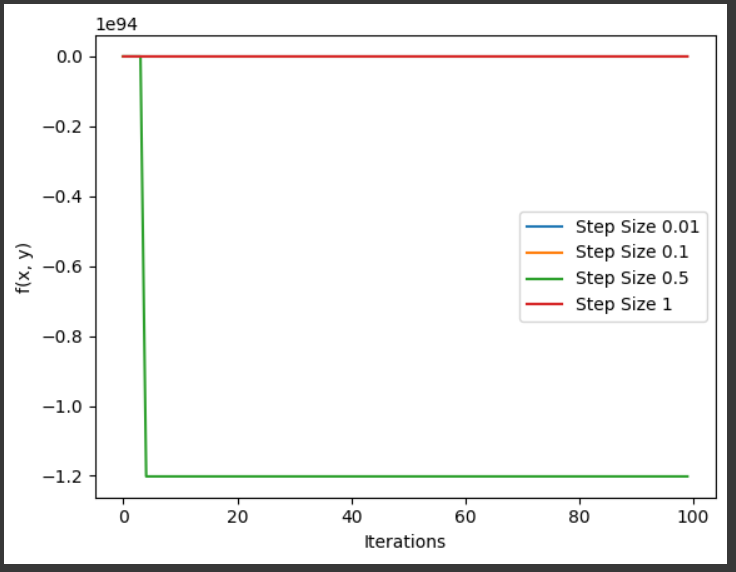
    plt.plot(range(num\_iter), values, label=f'Step Size {h}')

plt.xlabel('Iterations')

plt.ylabel('f(x, y)')

plt.legend()

plt.show()



c)

#Code by Nguyen Minh Duc\_ITITIU21045\_Lab06TMC\_EX2c

import numpy as np

def f(x, y):

    return 3.5\*x + 2\*y + x\*\*2 - x\*\*2 + 2\*x\*y - y\*\*2

def gradient(x, y):

    df\_dx = 3.5 + 2\*x - 1 - y

    df\_dy = 2 - x - 2\*y

    return np.array([df\_dx, df\_dy])

def hessian(x, y):

    hessian\_matrix = np.array([[2, -1],

                               [-1, -2]])

    return hessian\_matrix

x = 0

y = 0

tolerance = 1e-6

max\_iterations = 100

for i in range(max\_iterations):

    grad = gradient(x, y)

    hess = hessian(x, y)

    step = np.linalg.solve(hess, -grad)

    x += step[0]

    y += step[1]

    delta\_f = f(x, y) - f(x - step[0], y - step[1])

    if abs(delta\_f) < tolerance:

        break

print(f"Optimal solution: x = {x}, y = {y}, f(x, y) = {f(x, y)}")



d)

#Code by Nguyen Minh Duc\_ITITIU21045\_Lab06TMC\_EX2d

from scipy.optimize import minimize

import numpy as np

# Define the objective function

def objective\_function(parameters):

    x, y = parameters

    return -(3.5\*x + 2\*y + x\*\*2 - x\*\*4 + 2\*x\*y - y\*\*2)

# Define the gradient of the objective function

def gradient(parameters):

    x, y = parameters

    df\_dx = -(-4\*x\*\*3 + 2\*y + 3.5)

    df\_dy = -(2\*x - 2\*y + 2)

    return np.array([df\_dx, df\_dy])

# Define the Hessian matrix of the objective function

def hessian(parameters):

    x, y = parameters

    d2f\_dx2 = 12\*x\*\*2 - 3.5

    d2f\_dy2 = 2

    d2f\_dxdy = 2

    return np.array([[d2f\_dx2, d2f\_dxdy], [d2f\_dxdy, d2f\_dy2]])

# Initial guess

initial\_guess = [0, 0]

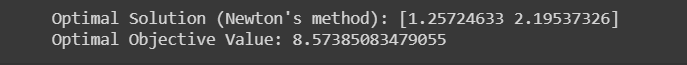
# Minimize the negative of the objective function (maximize the original function) using Newton's method

result = minimize(objective\_function, initial\_guess, method='Newton-CG', jac=gradient, hess=hessian)

# Display the result

print("Optimal Solution (Newton's method):", result.x)

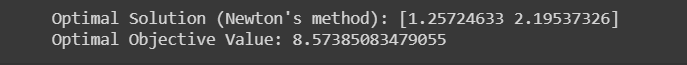
print("Optimal Objective Value:", -result.fun)  # Negate the value to get the original objective function value



Comparing 4 results:

1. A group of white letters and numbers

   Description automatically generated--- 0.0025413036346435547 seconds ---
2. A graph with numbers and a line

   Description automatically generated with medium confidence--- 0.17054200172424316 seconds ---
3. --- 0.0010807514190673828 seconds ---
4. --- 0.005110263824462891 seconds ---

So the most optimized way is the steepest ascent method

**Q3**

#Code by Nguyen Minh Duc\_ITITIU21045\_Lab06TMC\_EX3

import numpy as np

def f(x, y):

    return (x - 3)\*\*2 + (y - 2)\*\*2

def gradient(x, y):

    df\_dx = 2 \* (x - 3)

    df\_dy = 2 \* (y - 2)

    return np.array([df\_dx, df\_dy])

def steepest\_descent(x0, y0, epsilon\_s):

    x = x0

    y = y0

    iteration = 0

    while True:

        grad = gradient(x, y)

        magnitude\_grad = np.linalg.norm(grad)

        print(f"Iteration {iteration}: x = {x:.4f}, y = {y:.4f}, f(x, y) = {f(x, y):.4f}, ||grad|| = {magnitude\_grad:.4f}")

        if magnitude\_grad < epsilon\_s:

            break

        # Update x and y in the opposite direction of the gradient

        step\_size = 0.1  # You may need to experiment with different step sizes

        x -= step\_size \* grad[0]

        y -= step\_size \* grad[1]

        iteration += 1

    print(f"\nMinimum found at x = {x:.4f}, y = {y:.4f}, f(x, y) = {f(x, y):.4f}")

# Initial guess and stopping criterion

x0, y0 = 1, 1

epsilon\_s = 0.01  # 1%

# Perform steepest descent

steepest\_descent(x0, y0, epsilon\_s)

