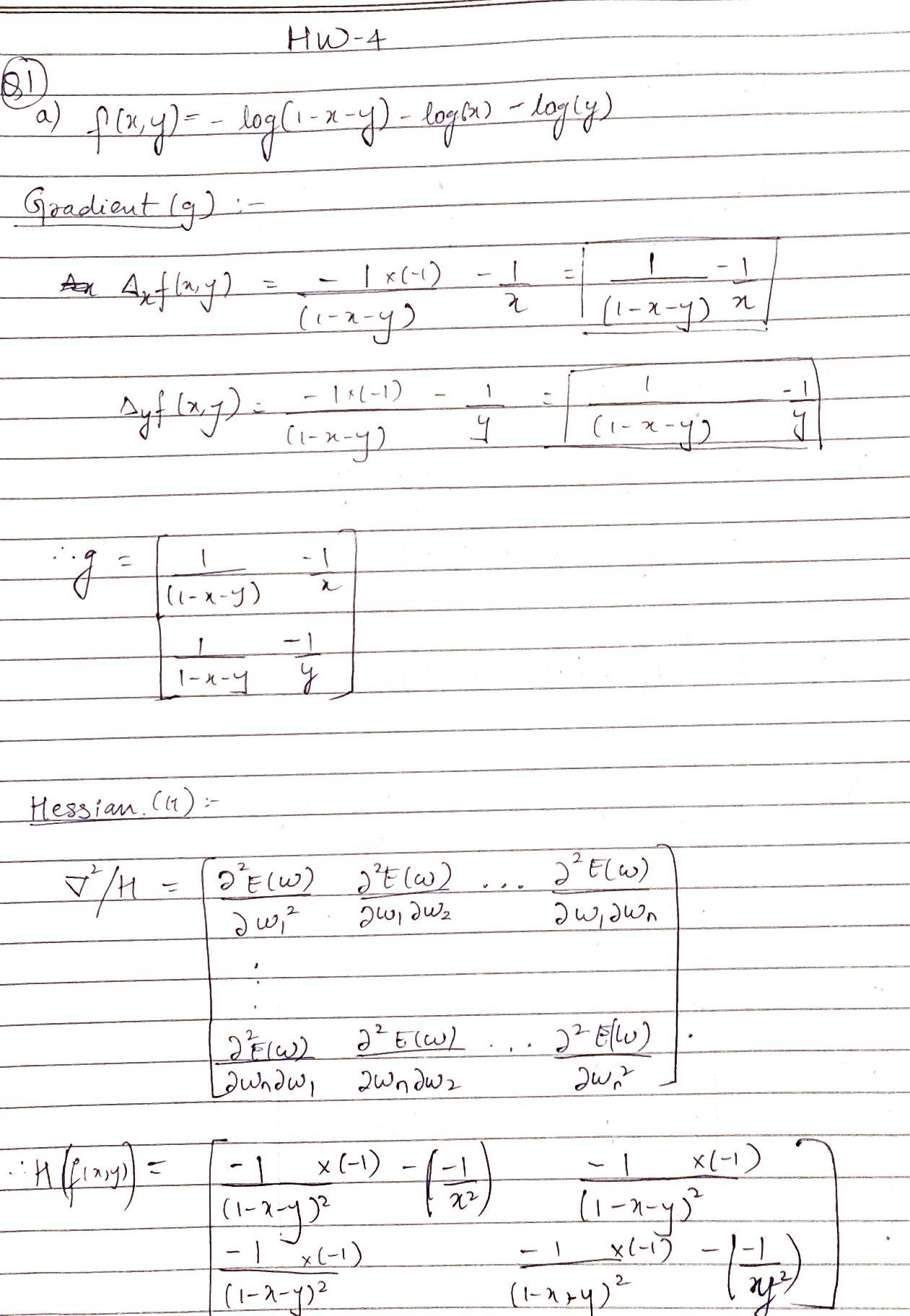
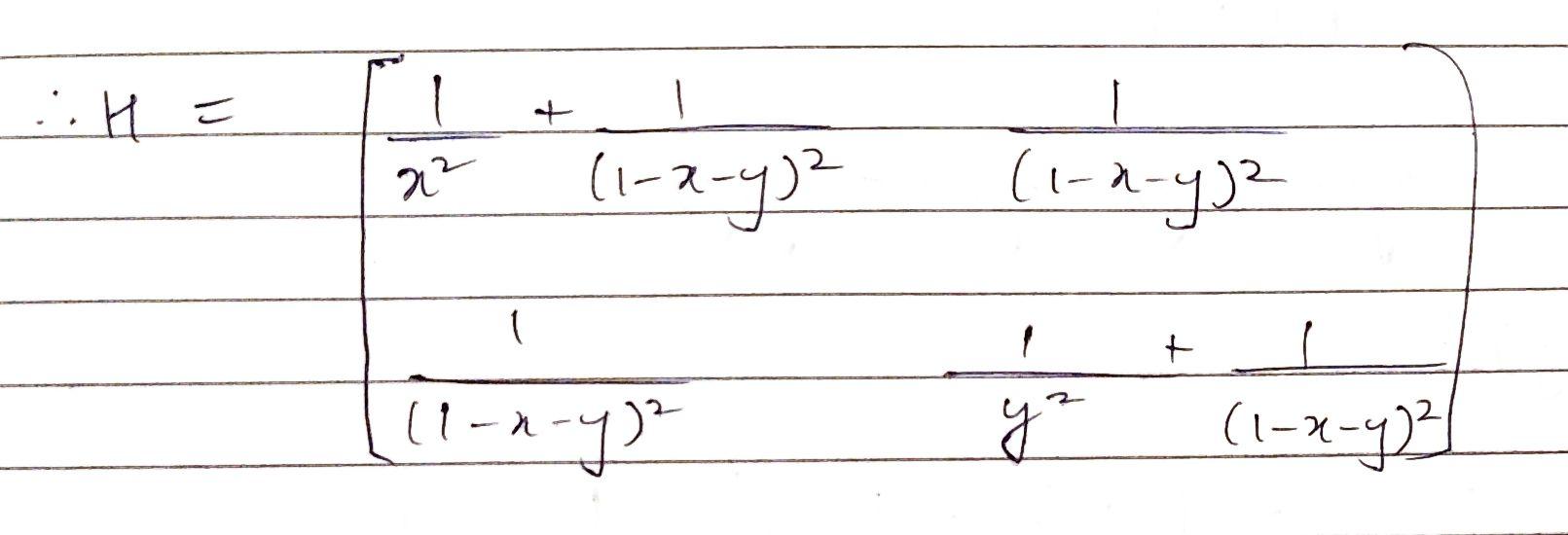
**CS559 - Neural Networks**

HW3 - Final Report

Q1)



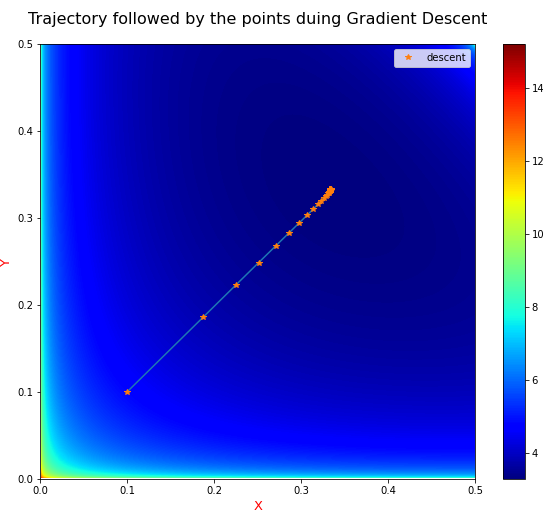


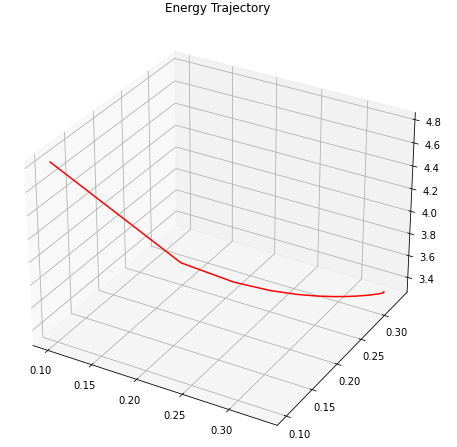
b)

I chose the initial point w0 as [0.1,0.1]. After starting with eta=1, it had to be changed several times so as the points fit the domain. The eta that I have settled on for my G.D is 0.01.

The local minimum occurs at: [0.3333428647524593, 0.33332426221451894]

Number of iterations taken: 67



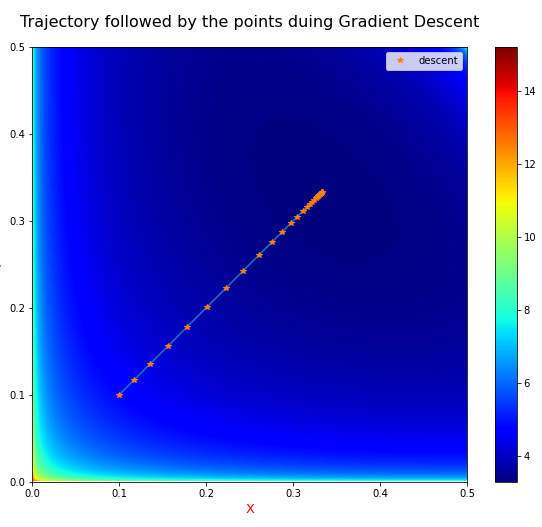


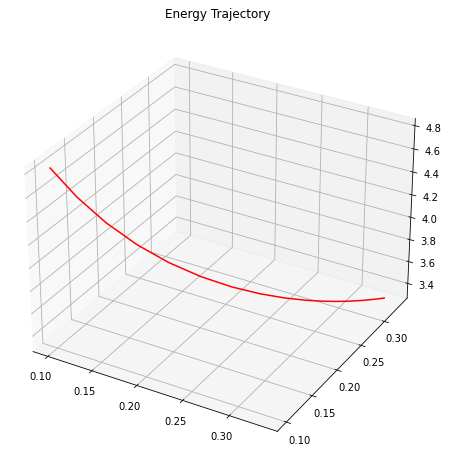
c)

For G.D using Newton’s method, my w0 is the same as before, [0.1,0.1]. Although the algorithm worked with initial eta = 1, it terminated only after 6 iterations. To get more distinguished descents, I have chosen eta as 0.2 for this part.

The local minimum occurs at: [0.33330098249289375, 0.33330098249289375]

Number of iterations taken: 42





d) For the above defined set of initial hyperparameters, the Gradient Descent algorithm was able to find the global minimum after 67 iterations whereas Newton's method implementation was able to find the same (almost if the end digits are ignored) after 42 iterations. I tried several iterations with different combinations of eta and w0, and the number of iterations for Newton’s method were comparatively lower in every iteration. This means that the gradient descent using Newton’s method converges faster.

However, I had to use a higher value of eta (learning rate) for Newton’s method to get the global minimum. For smaller values of eta, the number of descents were very high (in 3 digits). So to further clarify, gradient descent via Newton’s method converges faster if the approximation of hyperparameters are accurate enough and the resulting step size is small enough.

Code for Q1:

import numpy as np

import matplotlib.pyplot as plt

from numpy.linalg import inv,pinv

#function for learning using Gradient Descent

def gradientDescent(w0,eta,max\_iters,dfx,dfy,fxy,precision):

iters = 0

previous\_step\_size = 1

x,y = w0

W = [[x,y]]

E = [0]

while previous\_step\_size > precision and iters < max\_iters:

curr\_value\_x = x

curr\_value\_y = y

x = x - eta\*dfx(x,y)

y = y - eta\*dfy(x,y)

W = np.vstack((W,[x,y]))

previous\_step\_size = abs(x - curr\_value\_x)

E = np.vstack((E,fxy(x,y)))

iters += 1

print('W0:',w0)

print('\nLearning Rate',eta)

print('\nThe local minimum occurs at:', list([x,y]))

print('\nNumber of iterations taken:',iters)

return E,W

#function for learning using Gradient descent via Newton's method

def minimize\_NewtonsMethod(w0,eta,max\_iters,H,g,precision):

iters = 0

previous\_step\_size = 1

x,y = w0

W = [[x,y]]

w\_curr = w0

w\_next = w0

while previous\_step\_size > precision and iters < max\_iters:

delta = eta\*np.transpose(np.dot(pinv(H(w\_curr[0],w\_curr[1])),g(w\_curr[0],w\_curr[1])))

w\_next[0] = w\_curr[0] - delta[0][0]

w\_next[1] = w\_curr[1] - delta[0][1]

W = np.vstack((W,w\_next))

previous\_step\_size = abs(delta[0][0])

w\_curr = w\_next

iters += 1

print('W0:',w0)

print('\nLearning Rate',eta)

print('\nThe local minimum occurs at:', w\_curr)

print('\nNumber of iterations taken:',iters)

return W

#Function to plot the descent trajectory

def plot\_gradient\_trajectory(W):

x = np.linspace(0,0.5,1000)

y = np.linspace(0,0.5,1000)

X,Y = np.meshgrid(x,y)

Z = fxy(X,Y)

X\_trajectory = W[:,0]

Y\_trajectory = W[:,1]

figure = plt.figure(figsize = (10,8))

plt.imshow(Z,extent=[0,0.5,0,0.5],origin='lower',cmap = 'jet')

plt.title("Trajectory followed by the points duing Gradient Descent",fontsize=16,pad=20)

plt.plot(X\_trajectory,Y\_trajectory)

plt.plot(X\_trajectory,Y\_trajectory,'\*',label="descent")

plt.xlabel('X',fontsize=13,color ='red')

plt.ylabel('Y',fontsize=13,color ='red')

plt.legend(loc="upper right")

plt.colorbar()

plt.show()

#Function to plot the energies at each descent point

def plot\_Energy\_vs\_xy(W,fxy):

x = W[:,0]

y = W[:,1]

E = fxy(x,y)

figure = plt.figure(figsize = (10,8))

ax = plt.axes(projection = '3d')

ax.plot3D(x,y,E,'red')

ax.set\_title('Energy Trajectory');

#Gradient Descent

#initializing the learning parameters and starting point of the descent

w0 = [0.1,0.1]

eta = 0.01

max\_iters = 1000

precision = 0.000001

#defining the functions and gradients

dfx = lambda x,y: (1/(1-x-y) - 1/x)

dfy = lambda x,y: (1/(1-x-y) - 1/y)

fxy = lambda x,y : -np.log(1 - x - y) - np.log(x) - np.log(y)

E,W = gradientDescent(w0,eta,max\_iters,dfx,dfy,fxy,precision)

np.seterr(divide = 'ignore')

plot\_gradient\_trajectory(W)

plot\_Energy\_vs\_xy(W,fxy)

#Gradient Descent using Newton's Method

w0 = [0.1,0.2]

eta = 0.2

max\_iters = 1000

precision = 0.00001

H = lambda x,y : [[1/(x\*\*2) + 1/((1-x-y)\*\*2), 1/((1-x-y)\*\*2)],[1/((1-x-y)\*\*2),1/(y\*\*2) + 1/((1-x-y)\*\*2)]]

g = lambda x,y : [[(1/(1-x-y) - 1/x)],[(1/(1-x-y) - 1/y)]]

W = minimize\_NewtonsMethod(w0,eta,max\_iters,H,g,precision)

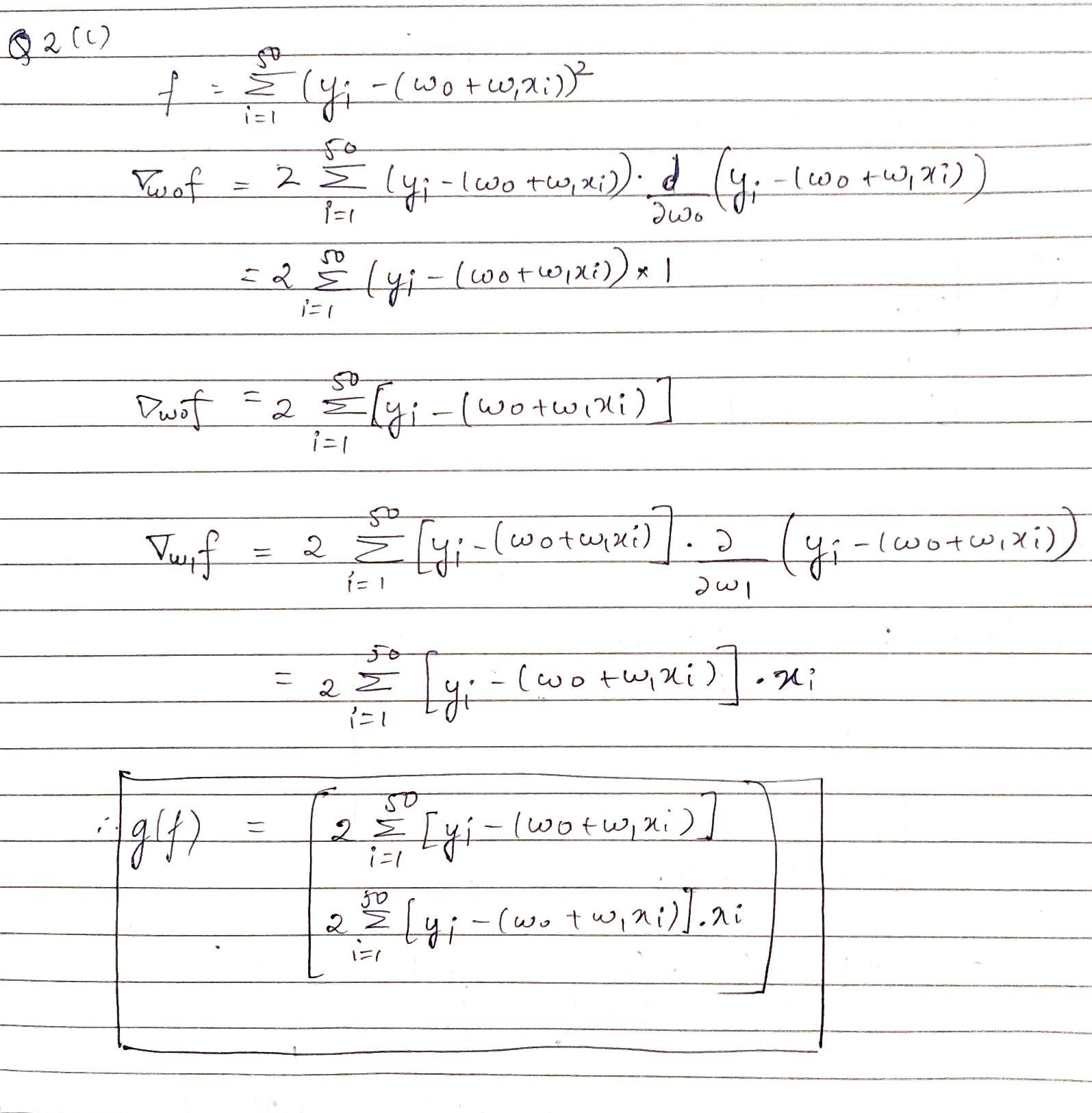
plot\_gradient\_trajectory(W)

plot\_Energy\_vs\_xy(W,fxy)

Q2)

a,b,c,d,f → code provided at the end of the document.

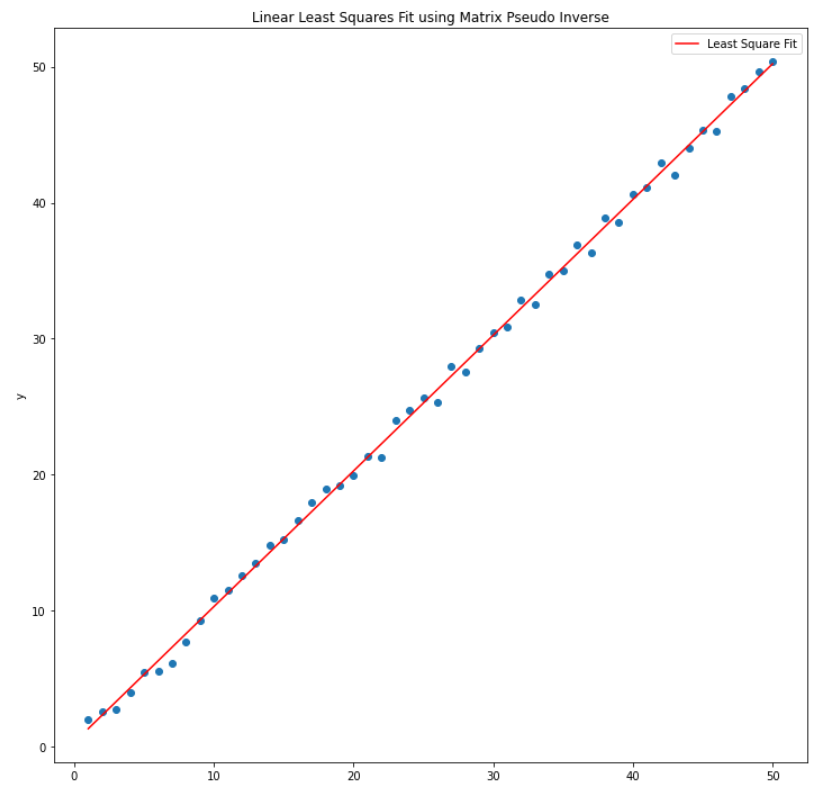
e)



Results:

Using Linear Least Mean Square Fit:

Final Weights: [0.31219901 0.99803169]



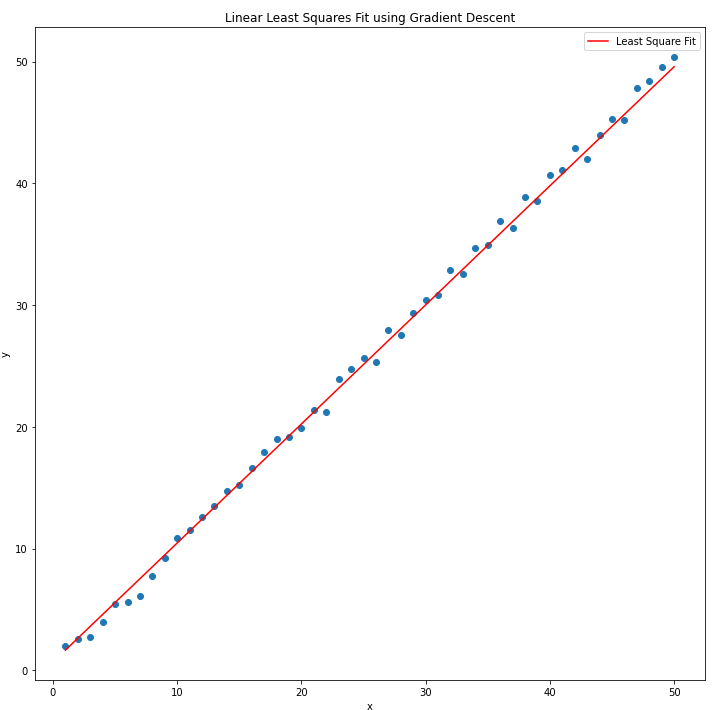
Using Gradient Descent:

eta = 0.00001

epsilon = 0.1

W0: [-0.67967812 0.58026508] -> Generated Randomly

Final Weights after GD: [-0.55470562 1.02378861]



2(f): Using the Gradient Descent method with an eta of 0.00001, the Weights take longer to converge. Also, the final values may be the global minimum because of which the number of iterations was limited to 500. The final weights are different with the Matrix Pseudo Inverse method and Gradient method, depending on the initial weight parameters.

Code for Q2:

import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline

def plot\_Graph(X,Y,y,type='pseudo\_inverse'):

fig,ax = plt.subplots(figsize=(12,12))

plt.scatter(X,Y)

plt.plot(X,y,color = 'red',label = 'Least Square Fit')

if type == 'gradient':

plt.title('Linear Least Squares fit using Gradient Descent')

else:

plt.title('Linear Least Squares fit using W = Y X+')

plt.xlabel('x')

plt.ylabel('y')

ax.legend()

plt.show()

return

def LeastSquareFit():

X = np.array([i for i in range(1,51)])

Y = np.array([i+np.random.uniform(low=-1,high=1) for i in X])

W = np.matmul(Y,np.linalg.pinv(np.array((np.ones\_like(X),X))))

y\_fit = [W[0] + W[1] \* x for x in X]

plot\_Graph(X,Y,y\_fit)

return X,Y,W

X,Y,W = LeastSquareFit()

print('Final Weights: {}'.format(W))

def getGradient(X,Y,W):

dW0 = 0

dW1 = 0

for i in range(len(X)):

dW0 += (-2) \* (Y[i]-(W[0]+W[1]\*X[i]))

dW1 += (-2) \* (X[i]) \* (Y[i]-(W[0]+W[1]\*X[i]))

return np.array([dW0,dW1])

def W\_update(X,Y,W,eta):

W = W - (eta\*getGradient(X,Y,W))

return W

def Error(X,Y,W):

E = 0

for i in range(len(X)):

E+= (Y[i]-(W[0]-W[1]\*X[i]))\*\*2

return E

def gradient\_descent(X,Y,W,eta,epsilon,max\_epochs):

errors = []

error\_rate = 1

epoch = 1

error = Error(X,Y,W)

errors.append(error)

while error\_rate >= epsilon and epoch <= max\_epochs:

error = Error(X,Y,W)

W = W\_update(X,Y,W,eta)

updated\_error = Error(X,Y,W)

errors.append(updated\_error)

error\_rate = abs(updated\_error-error)

epoch+=1

return W

eta = 0.00001

epsilon = 0.1

W = np.array([np.random.uniform(-1,1),np.random.uniform(-1,1)])

W\_GD = gradient\_descent(X,Y,W,eta,epsilon,500)

print('\n\nEta: {}'.format(eta))

print('W0: {}'.format(W))

print('Final Weights after GD: {}'.format(W\_GD))

y = generate\_line(X,W\_final)

plot(X,Y,y,'gradient')