HACETTEPE UNIVERSITY
ENGINEERING FACULTY
ELECTRICAL AND ELECTRONICS ENGINEERING PROGRAM

2023-2024
SPRING SEMESTER

ELE785
NEURAL NETWORKS

HW1

N23239410 – Ali Bölücü

Table of Content

1) Q1. Minimum Norm

Find least square estimator θls for m<n (i.e. there are more variables than equations since m is the number of data pairs recorded and n is the number of parameters) under the minimum norm consideration by stating a minimization problem as follows:

minimize $|\theta|$

subject to: $A.\theta = y \ (with\ variable\ \theta \in R^n)$

1.a) Method solution

The A. $\theta =$ y is

M equation and N variables with M<N

$$f_1(u_1)\theta_1 + \ f_2(u_1)\theta_1 + \cdots + \ f_n(u_1)\theta_n = y_1$$
$$f_1(u_2)\theta_1 + \ f_2(u_2)\theta_1 + \cdots + \ f_n(u_2)\theta_n = y_2$$
$$\ldots$$
$$f_1(u_m)\theta_1 + \ f_2(u_m)\theta_1 + \cdots + \ f_n(u_m)\theta_n = y_m$$

The exact solution may not be possible. To overcome this problem, the add error e to equation as;

$$A.\theta + e = y$$

Now the goal is to minimize the error to find solution in θ.

$$minimize\ \left||y - A.\theta|\right|^2$$

The 2-norm or Euclidean norm of this equation is

$$\sum_{i=1}^{i=m}(y_i - a_i.\theta)^2 = (y - A.\theta)^T.(y - A.\theta)$$

$$\boldsymbol{\theta = (A^T.A)^{-1}A^T.y}$$

2) Q2. RLS Study

a) Find the least square estimator θ =[θ0 θ1 θ2] for the model y(t)= θ0 + θ1*x1+ θ2*x2+ e'(t) proposed for the data set given in the Table.1 below,

b) Use LMS algorithm to find θ=[θ0 θ1 θ2] for the same model. Choose the learning rate carefully for the convergent iteration. Plot each variable θi during the adaptation. Compare your results with the Wiener's Optimal Solution (θ*=R-1 p).

X1	1	2	2	2	3	3	4	5	5	5	6	7	8	8	9
X2	2	5	3	2	4	5	6	5	6	7	8	6	4	9	8
Y	2	1	2	2	1	3	2	3	4	3	4	2	4	3	4

Table 1

2.a) Using Least Square Estimator

The equation $\boldsymbol{\theta} = (\boldsymbol{A^T}.\boldsymbol{A})^{-1}\boldsymbol{A^T}.\boldsymbol{y}$ is derived from the least squares method, which minimizes the sum of the squared differences between the observed and predicted values.

The A matrix defined as

A = transpose([1 1 1 1 1 1 1 1 1 1 1 1 1 1 1;
1 2 2 2 3 3 4 5 5 5 6 7 8 8 9;
2 5 3 2 4 5 6 5 6 7 8 6 4 9 8];);

$y = \text{transpose}([\ [2\ 1\ 2\ 2\ 1\ 3\ 2\ 3\ 4\ 3\ 4\ 2\ 4\ 3\ 4];);$

$\boldsymbol{\theta} = \text{transpose}(A) * A)^\wedge - 1 * \text{transpose}(A) * y$

After applying the equation above. The result of $\boldsymbol{\theta'}$s are

```
a) Find the least square estimator

answer =

    1.3534
    0.2862
   -0.0042
```

Figure 1. Result of $\boldsymbol{\theta}$

Then we can calculate the resulting regression line as,

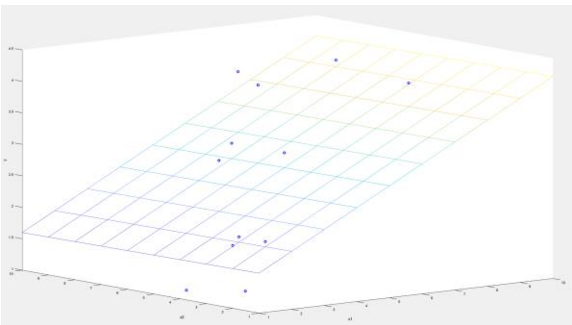


Figure 2. Data points and regression line

2.a.i) MATLAB Code

Editor - D:\Ders\M.Sc\M.Sc\ELE785 - Neural Networks\HW1\Q2_RLS_Study_part1.m

```
function Q2_RLS_Study
clear
clc
%y = teta_0 + teta_.x1 + teta_2.x;

table = [1 2 2 2 3 3 4 5 5 5 6 7 8 8 9;
         2 5 3 2 4 5 6 5 6 7 8 6 4 9 8;
         2 1 2 2 1 3 2 3 4 3 4 2 4 3 4];

table_x1 = [1 2 2 2 3 3 4 5 5 5 6 7 8 8 9];
table_x2 = [2 5 3 2 4 5 6 5 6 7 8 6 4 9 8];
table_y  = [2 1 2 2 1 3 2 3 4 3 4 2 4 3 4];


disp("a) Find the least square estimator");

% A. teta = y

A = [1 1 2;
     1 2 5;
     1 2 3;
     1 2 2;
     1 3 4;
     1 3 5;
     1 4 6;
     1 5 5;
     1 5 6;
     1 5 7;
     1 6 8;
     1 7 6;
     1 8 4;
     1 8 9;
     1 9 8];

y = transpose(table_y);
answer = (transpose(A)*A)^-1* transpose(A)*y

answer_line = answer(1) + answer(2)*table_x1 + answer(3)*table_x2;

figure (1);
plot3(table_x1,table_x2,y,'bo','linewidth',2);
hold on;

n=length(table_x1);
A=[ones(n,1) table_x1' table_x2'];
c=pinv(A)*y;
x1=linspace(1,10,10);
x2=linspace(1,10,10);
[x1,x2]=meshgrid(x1,x2);
y=c(1)+c(2)*x1+c(3)*x2;
mesh(x1,x2,y)

xlabel('x1');
ylabel('x2');
zlabel('y');
hold off;

end
```

2.b) Using LMS algorithm

The LMS algorithm is given in the book as

Input signal vector : x(n)

Desired response : d(n)

User-selected parameter: η

Initialization. Set w^(0) = 0

Computation For n = 1, 2, ..., compute

$$e(n) = d(n) - w^T(n).x(n)$$
$$w(n+1) = w(n) + \eta.\mathrm{x}(\mathrm{n}).\mathrm{e}(\mathrm{n})$$

The Wiener’s optimal solution is similar to least square estimator, the answer we get is same with least square estimation.

```
Wiener's Optimal Solution:
    1.3534
    0.2862
   -0.0042
```

Figure 3 Weiner's Optimal Solution

The LMS algorithm’s result will converge these values but the picking the right step-size is important.

When the step-size parameter selected as 0.001

$\boldsymbol{\theta}$ values results as

$\theta_0 = 1.3963$
$\theta_1 = 0.2794$
$\theta_2 = -0.0073$

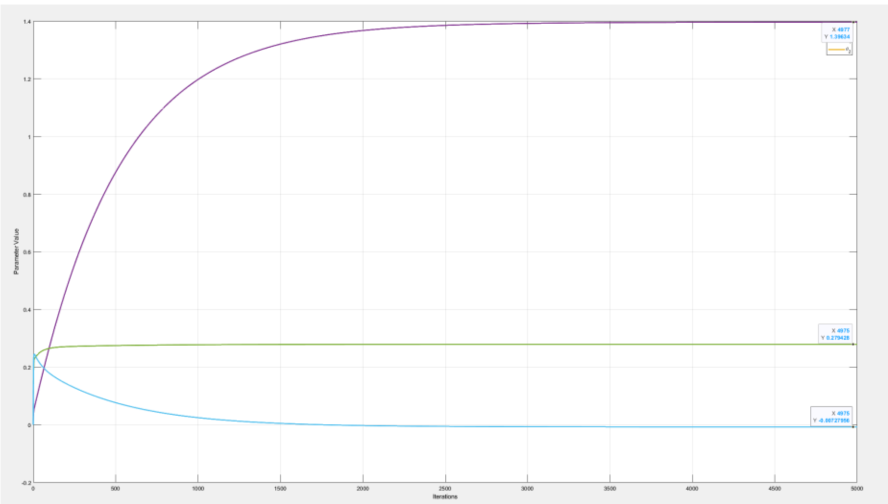


Figure 4 Result of ϑ at 5000 iteration

When the step-size parameter selected as 0.001

$\boldsymbol{\theta}$ values results as

$\theta_0 = 1.3531$
$\theta_1 = 0.2853$
$\theta_2 = -0.0036$

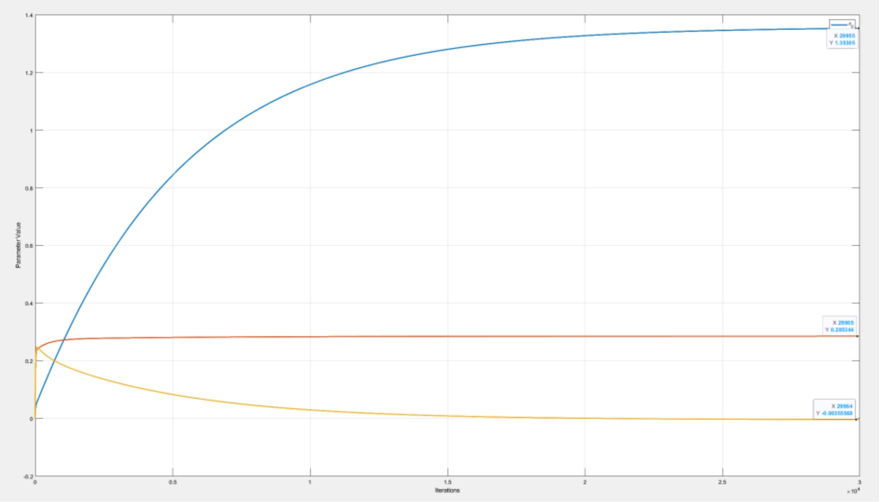


Figure 5 Result of ϑ at 30000 iteration

The result of Wiener's Optimal Solution is better than LMS in this experiment. The LMS solution converges the better parameters when a very low step-size is selected.

2.b.i) MATLAB Code

Editor - D:\Ders\M.Sc\M.Sc\ELE785 - Neural Networks\HW1\Q2_RLS_Study_part2.m

```matlab
% Data
data = [1 2 2 2 3 3 4 5 5 5 6 7 8 8 9;
        2 5 3 2 4 5 6 5 6 7 8 6 4 9 8;
        2 1 2 2 1 3 2 3 4 3 4 2 4 3 4];

% Add bias term to the inputs
X = [ones(1, size(data, 2)); data(1:2,:)];
y = data(3,:);

% LMS algorithm
learning_rate = 0.0001;
epochs = 30000;
weight = zeros(size(X, 1), 1);
weight_history = zeros(size(X, 1), epochs+1);
weight_history(:,1) = weight;

for epoch = 1:epochs
    for i = 1:size(X, 2)
        % Predicted output
        y_pred = X(:,i)' * weight;
        % Error
        error = y(i) - y_pred;
        % Update parameters
        weight = weight + learning_rate * error * X(:,i);
        % Store parameter values for plotting
        weight_history(:,epoch+1) = weight;
    end
end
%weight
% Plotting
figure (2);
iterations = 0:epochs;
plot(iterations, weight_history(1,:), 'LineWidth', 2);
hold on;
plot(iterations, weight_history(2,:), 'LineWidth', 2);
plot(iterations, weight_history(3,:), 'LineWidth', 2);
xlabel('Iterations');
ylabel('Parameter Value');
legend('\theta_0', '\theta_1', '\theta_2');
grid on;


```

Editor - D:\Ders\M.Sc\M.Sc\ELE785 - Neural Networks\HW1\Q2_RLS_Study_part3.m

```matlab
% Data
data = [1 2 2 2 3 3 4 5 5 5 6 7 8 8 9;
        2 5 3 2 4 5 6 5 6 7 8 6 4 9 8;
        2 1 2 2 1 3 2 3 4 3 4 2 4 3 4];

% Input data
X = [ones(1, size(data, 2)); data(1:2,:)];
% Output data
y = data(3,:);

% Autocorrelation matrix
R = X * X' / size(X, 2)
eig(R)

% Cross-correlation vector
p = X * y' / size(X, 2);

% Wiener's optimal solution
theta_optimal = inv(R) * p;
disp('Wiener''s Optimal Solution:');
disp(theta_optimal);

```

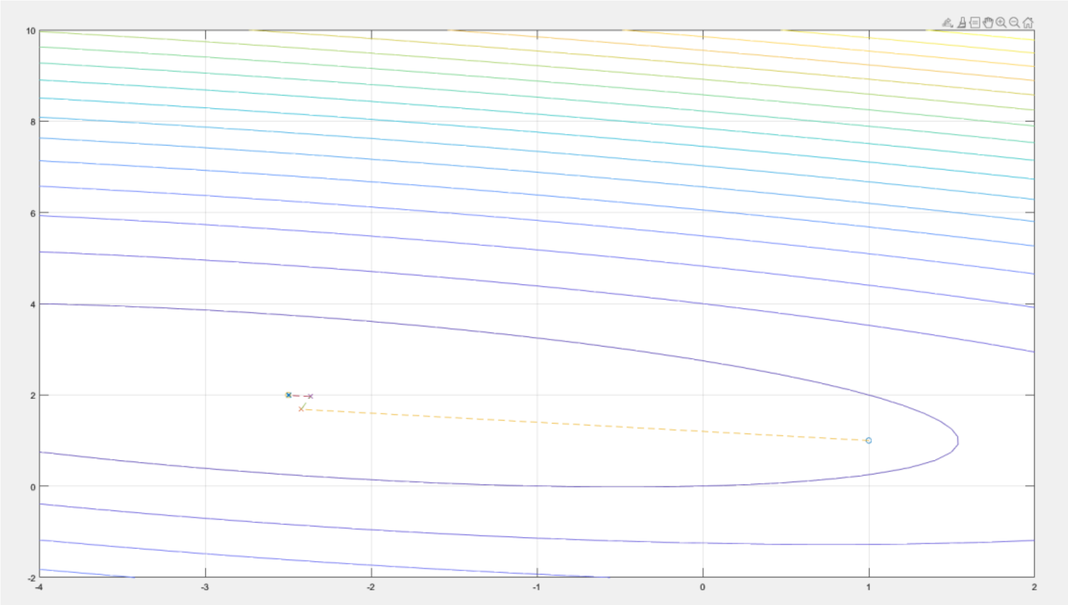
3) Q3. Derivative Based Optimization

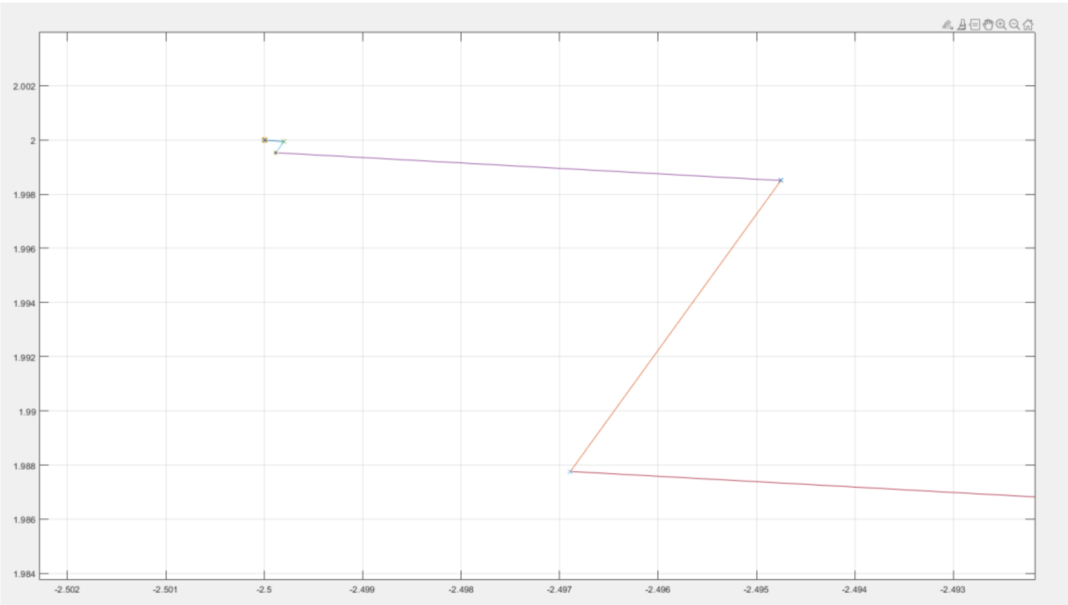
3.a) Apply the Four Descent Methods

3.a.i) The Steepest Descent Method

This method uses the first derivative to find its direction. İt converges to optimal solution slowly and the learning-rate parameter has influence in its converge behevior.

The algorithm converged to [-2.5, 2] point at **seventh** iteration.





3.a.i.1 MATLAB Code

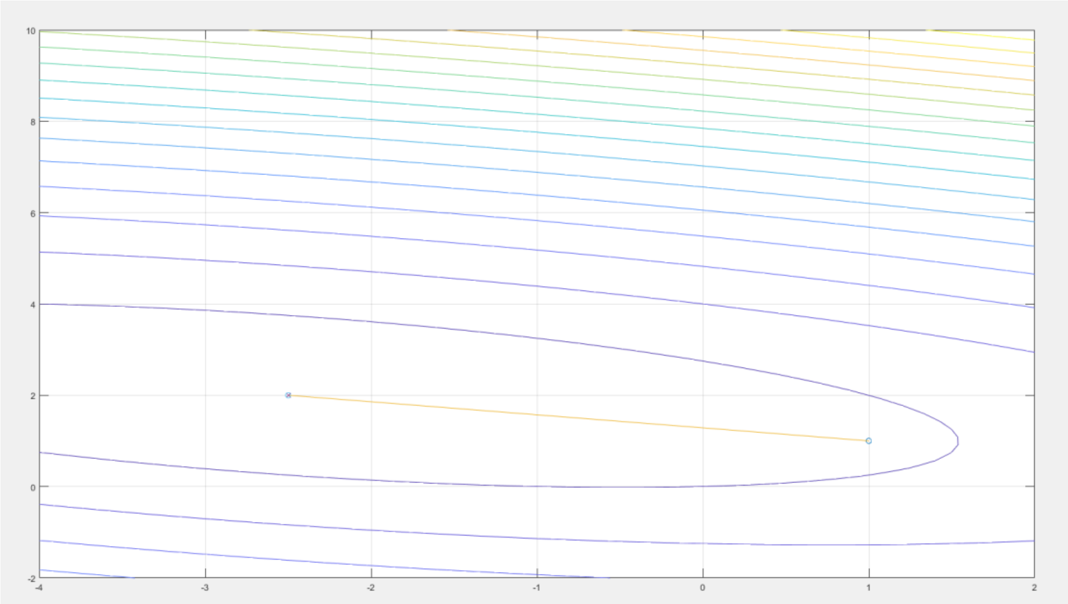
Editor - D:\Ders\M.Sc\M.Sc\ELE785 - Neural Networks\HW1\Q3_Derivative_Based_Optimization_Steepest_Descent.m

```
function Q3_Derivative_Based_Optimization_Steepest_Descent
tolerance = 1e-5;

% Initial conditions
q1_0 = 1;
q2_0 = 1;

q1_new = q1_0;
q2_new = q2_0;


E = @(q1,q2) q1.^2 + 4*q2.^2 + 2*q1.*q2 + q1 - 11*q2;

gradf = @(x) [2*x(1) + 2*x(2) + 1; 2*x(1) + 8*x(2) - 11];
hessf = @(x) [2, 2; 2,8];

dEdq1 = @(q1,q2) 2*q1 + 2*q2 + 1;
dEdq2 = @(q1,q2) 2*q1 + 8*q2 - 11;

figure (3);

fcontour(E,[-4 2 -2 10],'LevelStep',20);
grid;
hold on;
plot(q1_0,q2_0,'o');

s1 = -dEdq1(q1_new,q2_new);
s2 = -dEdq2(q1_new,q2_new);

    k = 0;
while norm([s1,s2]) > tolerance && k<10
    k = k+1;
    % Search Direction
    s1 = -dEdq1(q1_new,q2_new);
    s2 = -dEdq2(q1_new,q2_new);

    q1_d = @(d) q1_new+d*s1;
    q2_d = @(d) q2_new+d*s2;
    sE = @(d) E(q1_d(d),q2_d(d)); % bulunan yönde ilerliyecek
    step_size = 0.5

    q1_old = q1_new;
    q2_old = q2_new;

    q1_new = q1_d(step_size)
    q2_new = q2_d(step_size)


    plot(q1_new,q2_new,'x');
    plot([q1_old, q1_new], [q2_old, q2_new], '-');


end
   k
 plot(q1_new,q2_new,'o');

end
```

3.a.ii) Newton's method

This algorithm uses second order derivatives to find the right direction but there should be second order derivative and taking inverse of Hessian of w can be complex.

This algorithm converged to [-2.5, 2] point at **first** iteration.



3.a.ii.1 MATLAB Code

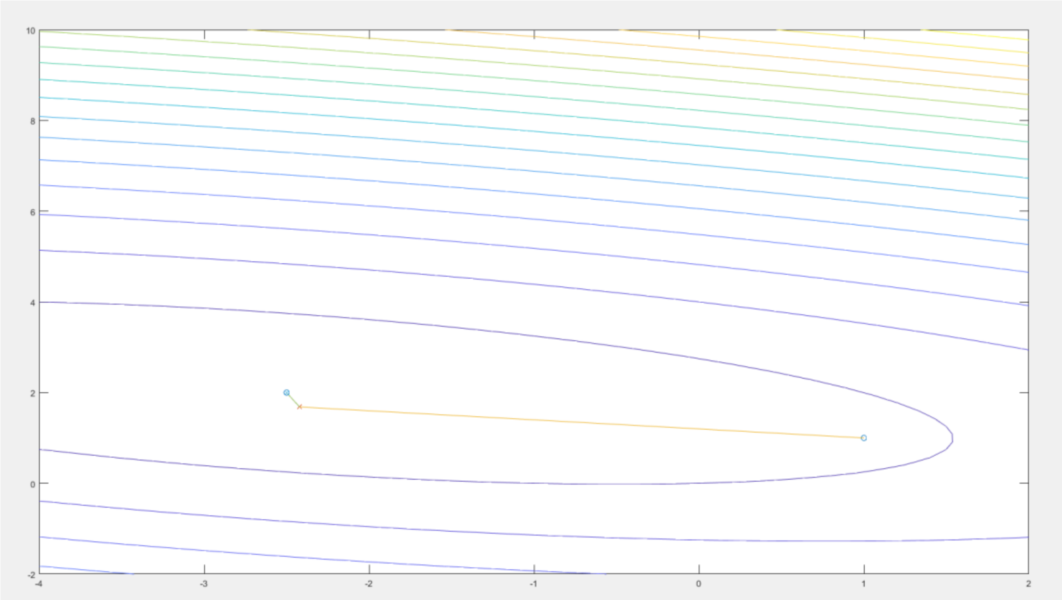
Editor - D:\Ders\M.Sc\M.Sc\ELE785 - Neural Networks\HW1\Q3_Derivative_Based_Optimization_Newtons_method.m

```
function Q3_Derivative_Based_Optimization_Newtons_method

tolerance = 1e-6;

% Initial conditions
q1_0 = 5;
q2_0 = 3.2;

q1_new = q1_0;
q2_new = q2_0;


E = @(q1,q2) q1.^2 + 4*q2.^2 + 2*q1.*q2 + q1 - 11*q2;

gradf = @(x) [2*x(1) + 2*x(2) + 1; 2*x(1) + 8*x(2) - 11];
hessf = @(x) [2, 2; 2,8];

dEdq1 = @(q1,q2) 2*q1 + 2*q2 + 1;
dEdq2 = @(q1,q2) 2*q1 + 8*q2 - 11;

figure (3);

fcontour(E,[-4 2 -2 10],'LevelStep',20);
grid;
hold on;
plot(q1_0,q2_0,'o');

s1 = -dEdq1(q1_new,q2_new);
s2 = -dEdq2(q1_new,q2_new);


while norm([s1,s2]) > tolerance

    % Search Direction
    s = hessf([q1_new,q2_new])\gradf([q1_new,q2_new]);
    s1 = -s(1)
    s2 = -s(2)

    q1_d = @(d) q1_new+s1;
    q2_d = @(d) q2_new+s2;
    sE = @(d) E(q1_d(d),q2_d(d)); % bulunan yönde ilerliyecek


    q1_old = q1_new;
    q2_old = q2_new;

    % bu method da burası gereksiz, sil
    q1_new = q1_d(step_size)
    q2_new = q2_d(step_size)

    plot(q1_new,q2_new,'x');
    plot([q1_old, q1_new], [q2_old, q2_new], '-');


end

 plot(q1_new,q2_new,'o');

end
```

3.a.iii) DFP Quasi-Newton method

This method is alternative to Newton method. It can be used when can be used if the Jacobian or Hessian is unavailable or is too expensive to compute at every iteration.

This algorithm converged to [-2.5, 2] point at **third** iteration.



3.a.iii.1 MATLAB Code

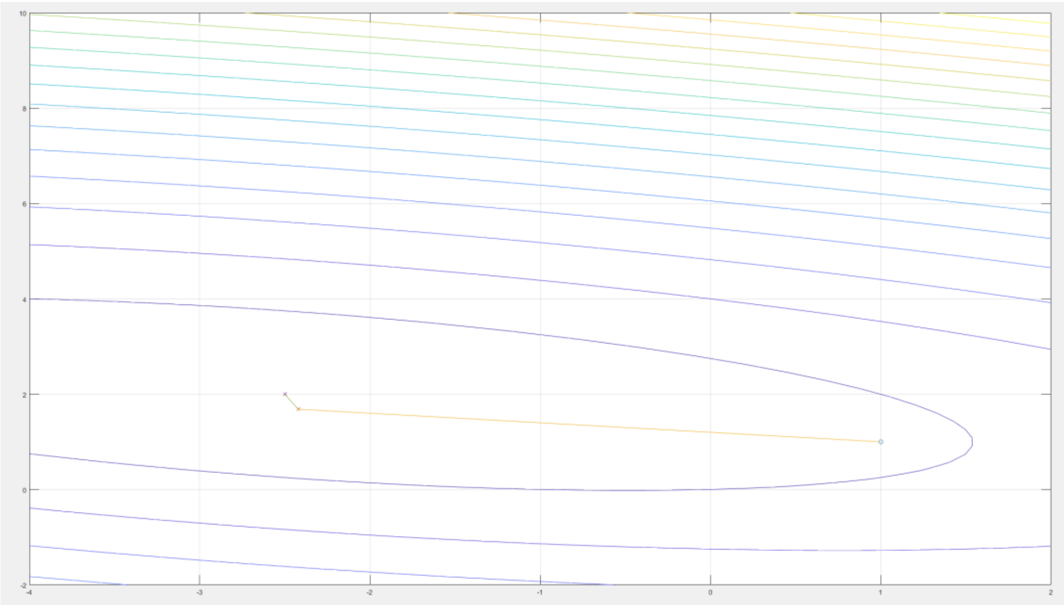
Editor - D:\Ders\M.Sc\M.Sc\ELE785 - Neural Networks\HW1\Q3_Derivative_Based_Optimization_DFP_Quasi_Newton_method.m

```
function Q3_Derivative_Based_Optimization_DFP_Quasi_Newton_method
    tolerance = 1e-6;


    q0 = [1; 1];

    E = @(q) q(1)^2 + 4*q(2)^2 + 2*q(1)*q(2) + q(1) - 11*q(2);
    gradE = @(q) [2*q(1) + 2*q(2) + 1; 2*q(1) + 8*q(2) - 11];

    H_inv = eye(2);

    figure(3);
    EE = @(a1,a2) a1.^2 + 4*a2.^2 + 2*a1.*a2 + a1 - 11*a2;
    fcontour(EE, [-4 2 -2 10], 'LevelStep', 20);
    hold on;

    plot(q0(1), q0(2), 'o');

    % Main optimization loop
    while true
        % Calculate search direction using inverse Hessian approximation
        s = -H_inv * gradE(q0);

        % Line search along the search direction
        sE = @(alpha) E(q0 + alpha * s);
        step_size = fminsearch(sE, 0);

        % Update the current point
        q_old = q0;
        q0 = q0 + step_size * s;

        % Update the inverse Hessian approximation using DFP formula
        y = gradE(q0) - gradE(q_old);
        H_inv = H_inv + ((step_size * s) * (step_size * s)') / (step_size * s' * s) - (H_inv * y * y' * H_inv) / (y' * H_inv * y);

        % Plot the current point and the line segment
        plot(q0(1), q0(2), 'x');
        plot([q_old(1), q0(1)], [q_old(2), q0(2)], '-');

        if norm(gradE(q0)) < tolerance
            break;
        end


    end

    plot(q0(1), q0(2), 'o');
end

```

3.a.iv) Fletcher-Reeves's conjugate gradient method

This method uses the conjugate directions with line search algorithms to converge optimal solution.

This algorithm converged to [-2.5, 2] point at **Second** iteration.



3.a.iv.1 MATLAB Code

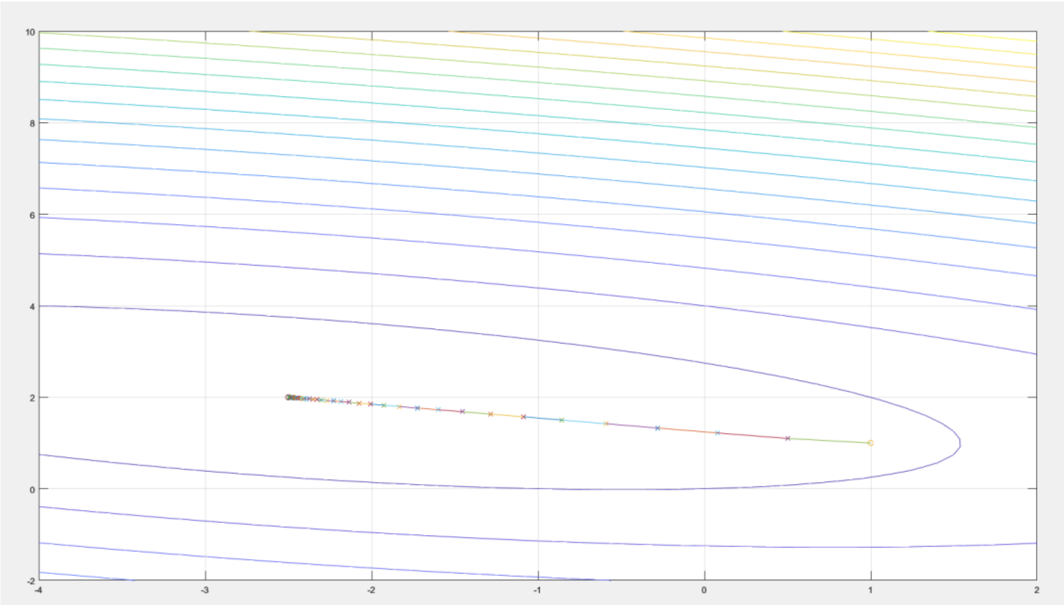
Editor - D:\Ders\M.Sc\M.Sc\ELE785 - Neural Networks\HW1\Q3_Derivative_Based_Optimization_FletcherReeves.m

```matlab
function Q3_Derivative_Based_Optimization_FletcherReeves

    f = @(x) x(1).^2 + 4*x(2).^2 + 2*x(1)*x(2) + x(1) - 11*x(2);
    gradf = @(x) [2*x(1) + 2*x(2) + 1; 2*x(1) + 8*x(2) - 11];
    hessf = @(x) [2, 2; 2,8];
    ms = {'Fletcher-Reeves', 'Polak-Ribiere'};
    x0 = [1;1];
    tol = 1e-6;
    maxits = 30;

    E = @(q1,q2) q1.^2 + 4*q2.^2 + 2*q1.*q2 + q1 - 11*q2;
    fcontour(E,[-4 2 -2 10],'LevelStep',20);
    grid;
    hold on;
    plot(x0(1),x0(2),'o');

    k = 0;
    x = x0;
    g = gradf(x);
    s = -g;

    while norm(s) > tol && k < maxits
        k = k+1;
        x_old =x;
        [x, alpha] = LineSearch(f, x, s);
        g_new = gradf(x);

        beta = (g_new'*g_new)/(g'*g);

        s = -g_new + beta*s;
        g = g_new;

        plot(x(1),x(2),'x');
        plot([x_old(1), x(1)], [x_old(2),  x(2)], '-');

    end

    hold off;
end

function [x_new, alpha] = LineSearch(f, x, s)
    % min_alpha f(x+alpha*s)
    k = 0;
    maxits = 10;
    f0 = phi(0, f, x, s);
    alpha = 1;
    f1 = phi(alpha, f, x, s);

    while f1 > f0 && k < maxits
        k = k+1;
        alpha = alpha/2;
        f1 = phi(alpha, f, x, s);
    end

    options = optimset('TolX', 1e-6, 'MaxIter', 50);
    alpha = fminbnd(@phi, 0, 2*alpha, options, f, x, s);
    x_new = x + alpha*s;
end

function [val] = phi(alpha, fun, x, s)
    val = feval(fun, x + alpha*s);
end
```

3.b) Use fixed η values for the steepest descent method.

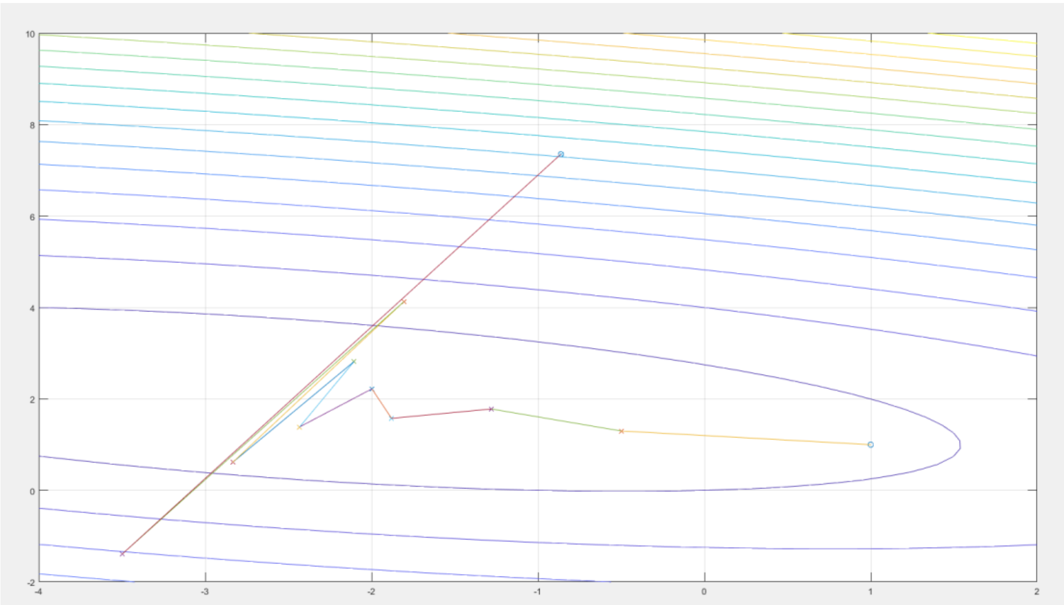
3.b.i) η=0.1

At the 90^{th} iteration, it converge the right result



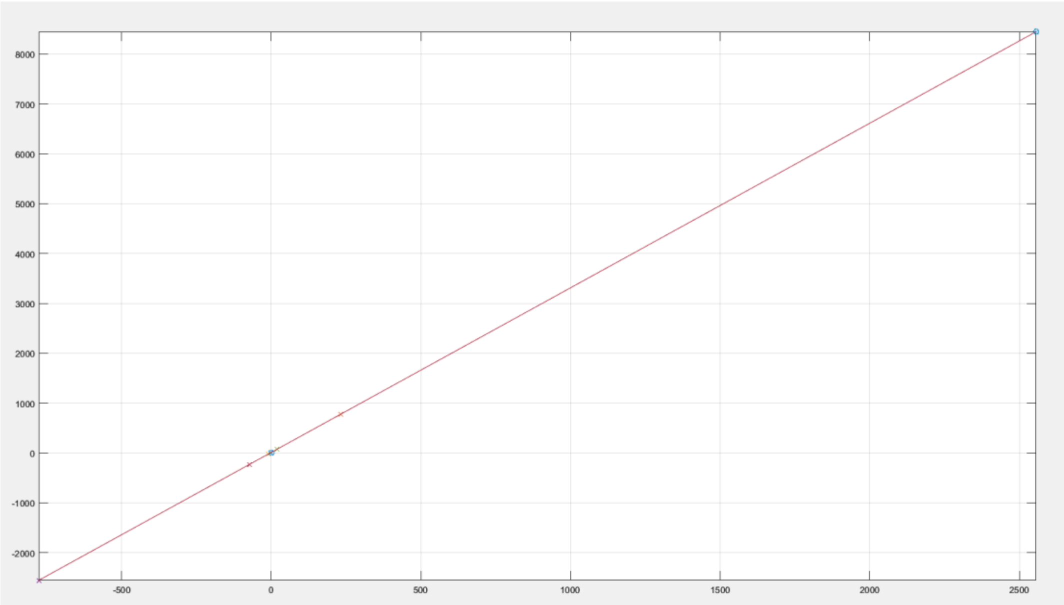
3.b.ii) η=0.3

After the 10 iteration it can be seen that fixed step-size with higher than maximum step-size value can diverge the answer.



3.b.i) η=0.5

When the step-size increased even more, it quickly diverges.



The choice of step size (η) significantly impacts the convergence behavior of the algorithm. In the case where η=0.3, after 10 iterations, it diverges, indicating that a step size higher than the maximum allowable value can lead to divergence. This highlights the importance of selecting an appropriate step size to ensure convergence.

4) Resources

[1] https://www.youtube.com/watch?v=V1cZXDW8nDo

[2] https://www.youtube.com/watch?v=xnnvgFaJCdo

[3] https://www.ee.hacettepe.edu.tr/~usezen/ele604/optimization2-2p.pdf

[4] https://en.wikipedia.org/wiki/Quasi-Newton_method

[5] Scientific Computing: An Introductory Survey, Revised Second Edition, Michael.T.Heath, SIAM, 2018