UNIVERSITY OF TEHRAN

COLLEGE OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

INTELLIGENT SYSTEM

ASSIGNMENT#1

MOHAMMAD HEYDARI

810197494

UNDER SUPERVISION OF:

DR. RESHAD HOSSINI

ASSISTANT PROFESSOR

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

UNIVERSITY OF TEHRAN

Oct. 2021

1 CONTENTS

2 QUESTION #1

2.1 ANALYSIS OF STATIONARY POINT

In this part we intend to find whole stationary points of a convex function which mentioned below:

$$f(\mathrm{x}_1, \mathrm{x}_2) = 3\mathrm{x}_1{}^2 + 12\mathrm{x}_1 + 8\mathrm{x}_2{}^2 + 8\mathrm{x}_2 + 6\mathrm{x}_1\mathrm{x}_2;$$

First order derivative:

$$fx_1 = \frac{\partial f(x)}{\partial x_1} = 6x_1 + 6x_2 + 12$$

$$\xrightarrow{solve\ equations} \quad \begin{cases} x_1 = -2.4 \\ x_2 = +0.4 \end{cases}$$

$$fx_2 = \frac{\partial f(x)}{\partial x_2} = 6x_1 + 16x_2 + 8$$

Second order derivative:

$$fx_1x_1 = \frac{\partial \partial f(x)}{\partial x_1 \partial x_1} = \frac{\partial(6x_1 + 6x_2 + 12)}{\partial x_1} = 6$$

$$fx_2x_2 = \frac{\partial \partial f(x)}{\partial x_2 \partial x_2} = \frac{\partial(6x_1 + 16x_2 + 8)}{\partial x_2} = 16$$

$$fx_2x_1 = \frac{\partial \partial f(x)}{\partial x_2 \partial x_1} = \partial(6x_1 + 16x_2 + 8)\partial x1 = 6$$

Then, we have only one stationary point in this specific question.

In the next step we are going to determine type of our stationary point which can be minimum, maximum or a saddle one.

$$D(\mathrm{x}_1, \mathrm{x}_2) = fx_1x_1(\mathrm{x}_1, \mathrm{x}_2)\ .\ fx_2x_2(\mathrm{x}_1, \mathrm{x}_2) - [fx_1x_1(\mathrm{x}_1, \mathrm{x}_2)]^2$$

$$\xrightarrow{yields} \begin{cases} D(x_1, x_2) = (6)(16) - 36 = 96 - 36 = 60 \ > \ 0 \\ f_{x_1 x_1} > 0 \end{cases} \xrightarrow{yields}$$ (-2.4,0.4) is a **minimum**.

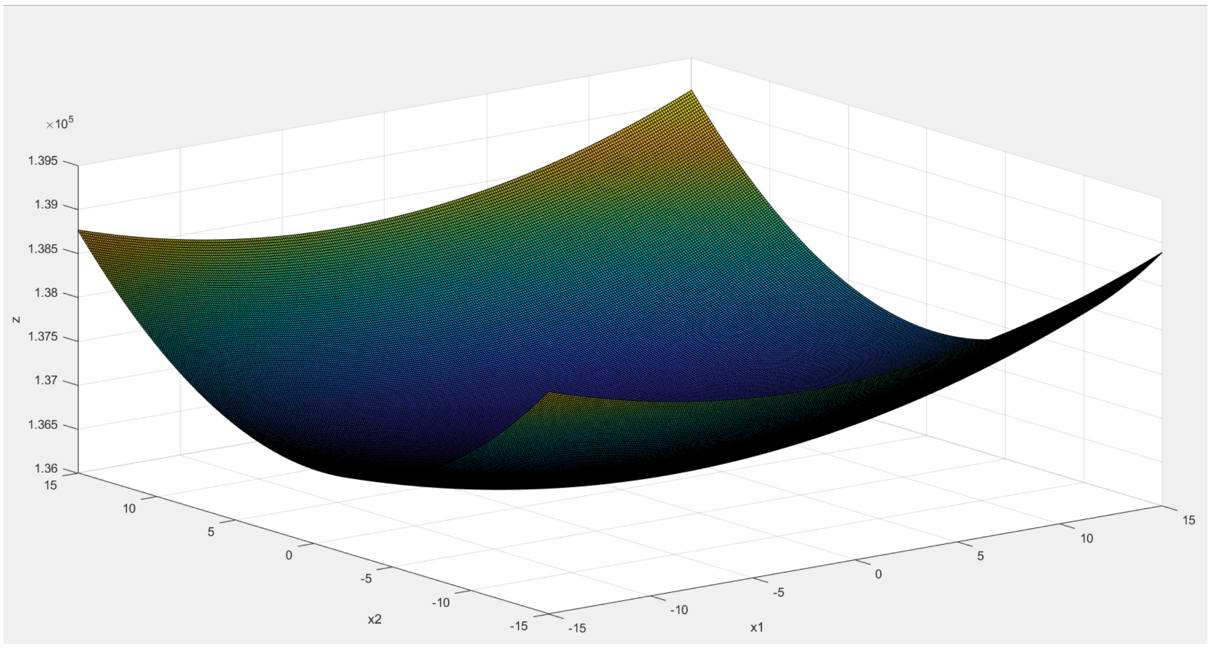


Figure 1: 3D representation of the Function

above picture represents 3D figure of our convex function in order to have a better process in both theoretical and graphical approaches.

As we can see this function have an absolute minimum point which is same as local minimum so in this case (-2.4,0.4) can be consider as both local or absolute minimum!

2.2 GRADIENT DESCENT METHOD

In this part we intend to use gradient descent method for finding step size First we start with exact Line search method to obtain the step size based on a minimum approach for an in gradient descent equations using differentiation operator.

Afterward, we implement gradient descent method in MATLAB using specific error function to give us the iteration numbers, final value, optimal x-values.

Note that we use gradient norm as our error function which guaranties algorithm convergence, further we will provide more detailed information about the inputs and outputs of our function.

- Part A: Calculating **direction** of gradient descent with starting from (1,1):

$$\boldsymbol{d_k} = -\nabla f\left(x_1^{k_1}, x_2^{k_2}\right) = \begin{pmatrix} -6x_1 - 6x_2 - 12 \\ -6x_1 - 16x_2 - 8 \end{pmatrix} = \begin{pmatrix} -24 \\ -30 \end{pmatrix}$$

- Part B: Exact Line search method for calculating the step size: (theoretical analyse)

Based on exact line search method we intend to minimise step-size so we substitute($x^k + a^k d^k$) term in our function then obtain minimum of $h(a)$ using differentiation operator:

Solve:

Step1:

$min_a \; h(a) := f\left(x^k + ad^k\right)$

$h(a) = f\left(x^k + ad^k\right) = 3(x_1^k + ad_1^k)^2 + 12(x_1^k + ad_1^k) + 8(x_2^k + ad_2^k)^2 + 8\left(x_2^k + ad_2^k\right) + 6\left(x_1^k + ad_1^k\right)\left(x_2^k + ad_2^k\right).$

Then we derive based on $\boldsymbol{a}$ from equation and then assign to zero:

$$\frac{\partial h(a)}{\partial a} = 6d_1^k(x_1^k + ad_1^k) + 12d_1^k + 16d_2^k(x_2^k + ad_2^k) + 8d_2^k + 6x_1^k d_2^k + 6d_1^k x_2^k + 12ad_1^k d_2^k = 0$$

Then:

$$a = \frac{(-6d_1^k x_1^k - 12d_1^k - 16d_2^k x_2^k - 8d_2^k - 6x_1^k d_2^k - 6d_1^k x_2^k)}{6d_1^k d_1^k + 16d_2^k d_2^k + 12d_1^k d_2^k}$$

$$\begin{cases} x_1^0 = 1\,, x_2^0 = 1 \\ d_1^0 = -24\,, d_2^0 = -30 \end{cases} \xrightarrow{yields} a = \frac{[-6(-24)(1)-12(-24)-16(-30)(1)-8(-30)-6(1)(-30)-6(-24)(1)]}{6(24)(24)+16(30)(30)+12(24)(30)} = \mathbf{0.0557}$$

Now we obtain updated x-value in steepest descent algorithm:

$$x_1^1 \leftarrow x_1^0 + a \times d_1^0 \qquad \xrightarrow{yields} \qquad x_1^1 = 1 + (0.0557)(-24) = -0.3368$$

$$x_2^1 \leftarrow x_2^0 + a \times d_2^0 \qquad \xrightarrow{yields} \qquad x_2^1 = 1 + (0.0557)(-30) = -0.671$$

Step2:

$min_a\, h(a) := f\left(x^k + ad^k\right)$

$h(a) = f\left(x^k + ad^k\right) = 3(x_1^k + ad_1^k)^2 + 12(x_1^k + ad_1^k) + 8(x_2^k + ad_2^k)^2 + 8\left(x_2^k + ad_2^k\right) +$
$6\left(x_1^k + ad_1^k\right)\left(x_2^k + ad_2^k\right).$

Then we derive based on $\boldsymbol{a}$ from equation and then assign to zero:

$$\frac{\partial h(a)}{\partial a} = 6d_1^k(x_1^k + ad_1^k) + 12d_1^k + 16d_2^k(x_2^k + ad_2^k) + 8d_2^k + 6x_1^k d_2^k + 6d_1^k x_2^k + 12ad_1^k d_2^k = 0$$

Then:

$$a = \frac{(-6d_1^k x_1^k - 12d_1^k - 16d_2^k x_2^k - 8d_2^k - 6x_1^k d_2^k - 6d_1^k x_2^k)}{6d_1^k d_1^k + 16d_2^k d_2^k + 12d_1^k d_2^k}$$

$$\begin{cases} x_1^1 = -0.3368\,, x_2^1 = -0.671 \\ d_1^1 = -24\,, d_2^1 = -30 \end{cases} \xrightarrow{yields} a = \frac{(-6d_1^1 x_1^1 - 12d_1^1 - 16d_2^1 x_2^1 - 8d_2^1 - 6x_1^1 d_2^1 - 6d_1^1 x_2^1)}{6d_1^1 d_1^1 + 16d_2^1 d_2^1 + 12d_1^1 d_2^1}$$

$$\xrightarrow{yields} a = +0.247$$

Now we obtain updated x-value in steepest descent algorithm:

$$x_1^2 \leftarrow x_1^1 + a \times d_1^1 \quad \xrightarrow{yields} \quad x_1^2 = -0.3368 + (0.247)(-5.9532) = -1.8072$$

$$x_2^2 \leftarrow x_2^1 + a \times d_2^1 \quad \xrightarrow{yields} \quad x_2^2 = -0.671 + (0.247)(4.7568) \quad = \quad 0.5039$$

- Part C: Now, we implement gradient descent using MATLAB:

All codes and simulation results have been attached in related directory.

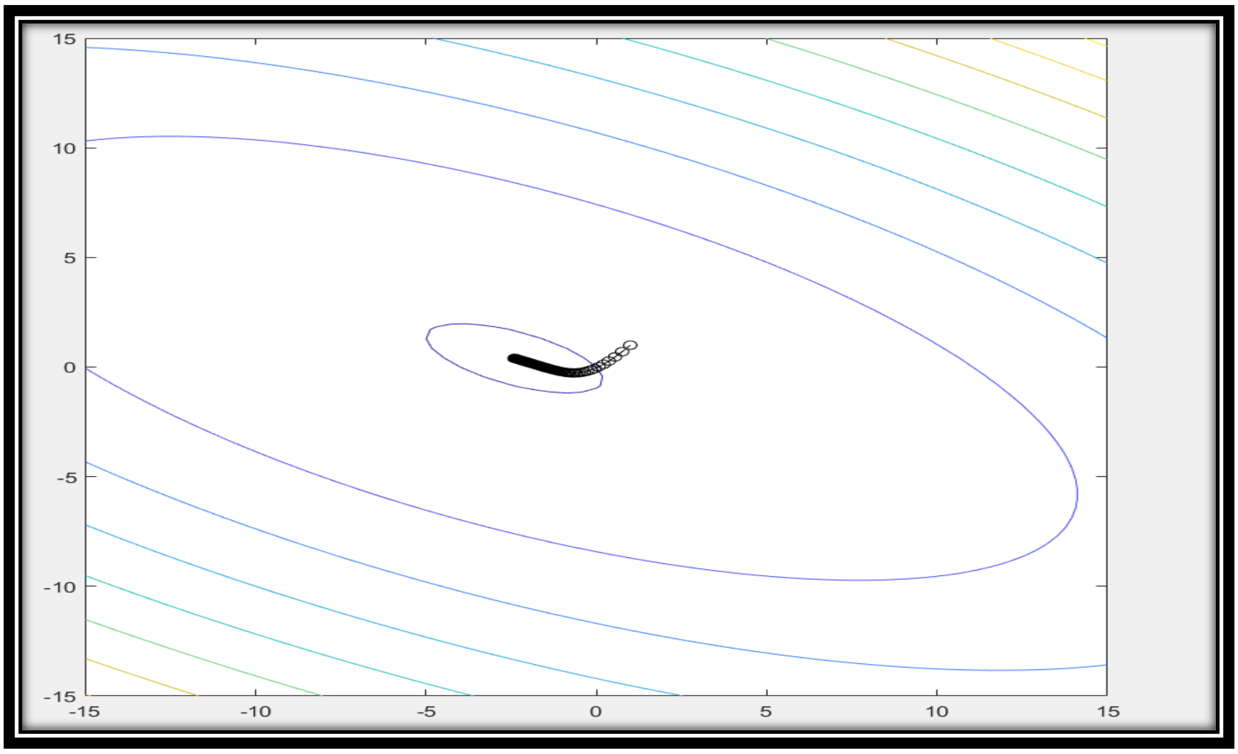
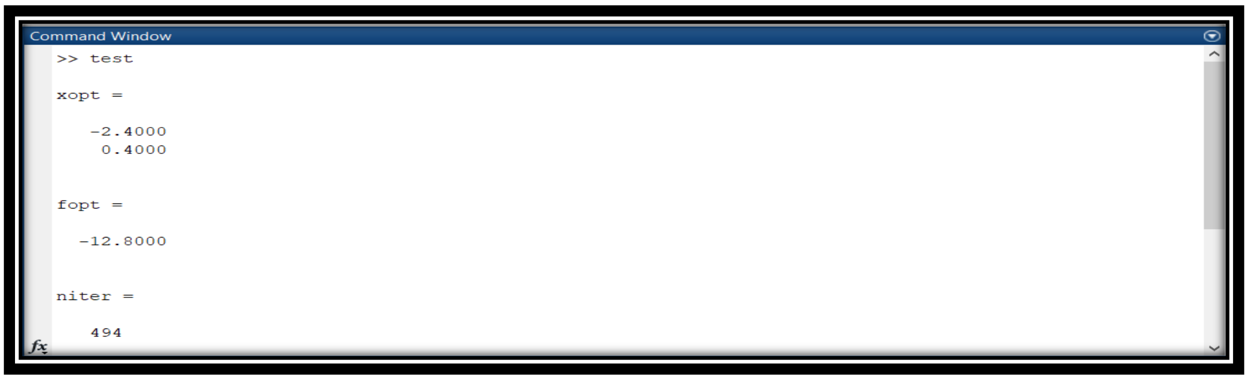
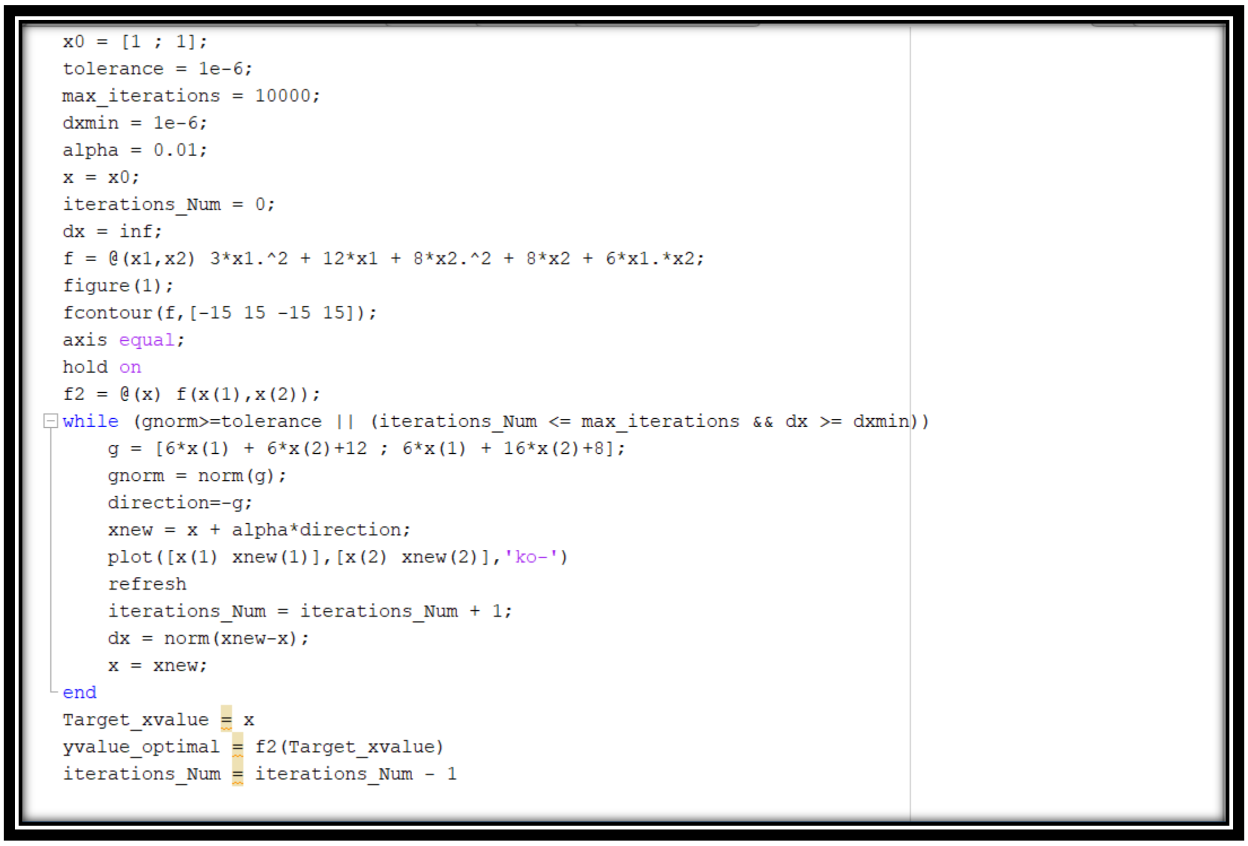


Figure 2: 3D representation of the contour and points



```
>> test

xopt =

   -2.4000
    0.4000


fopt =

  -12.8000


niter =

   494
```

Figure 3: final result



```matlab
x0 = [1 ; 1];
tolerance = 1e-6;
max_iterations = 10000;
dxmin = 1e-6;
alpha = 0.01;
x = x0;
iterations_Num = 0;
dx = inf;
f = @(x1,x2) 3*x1.^2 + 12*x1 + 8*x2.^2 + 8*x2 + 6*x1.*x2;
figure(1);
fcontour(f,[-15 15 -15 15]);
axis equal;
hold on
f2 = @(x) f(x(1),x(2));
while (gnorm>=tolerance || (iterations_Num <= max_iterations && dx >= dxmin))
    g = [6*x(1) + 6*x(2)+12 ; 6*x(1) + 16*x(2)+8];
    gnorm = norm(g);
    direction=-g;
    xnew = x + alpha*direction;
    plot([x(1) xnew(1)],[x(2) xnew(2)],'ko-')
    refresh
    iterations_Num = iterations_Num + 1;
    dx = norm(xnew-x);
    x = xnew;
end
Target_xvalue = x
yvalue_optimal = f2(Target_xvalue)
iterations_Num = iterations_Num - 1
```

Gradient descent implementation with MATLAB

In this section we implement gradient descent algorithm based on equations and martials which discussed in class session.

I think the most important factor in this implementation is about finding suitable function error and also determine an adequate and absolute threshold for hitting suitable convergence rate.

Note that in this case we use 10^-6 as of our threshold and also use from norm command in MATLAB as of our function error!

<u>**For running this part please check related directory and there you can easily find all of you need!**</u>

2.3 NEWTON'S METHOD

in this part we use Newton’s Method to find the minimum of previous function and we expect that the result matches as well as we supposed.

$$X_{k+1} \leftarrow X_k - H^{-1}g \quad \xrightarrow{yields} \quad X_1 = X_0 - H^{-1}g$$

where:

$$g_k = \nabla f\left(x_1^{k_1}, x_2^{k_2}\right) = \begin{pmatrix} +6x_1+6x_2+12 \\ +6x_1+16x_2+8 \end{pmatrix} \ @\, X_0 \xrightarrow{yields} \quad g = \begin{pmatrix} 24 \\ 30 \end{pmatrix}$$

$$fx_1x_1 = \frac{\partial\partial f(x)}{\partial x_1 \partial x_1} = \frac{\partial(6x_1 + 6x_2 + 12)}{\partial x_1} = 6$$

$$fx_2x_2 = \frac{\partial\partial f(x)}{\partial x_2 \partial x_2} = \frac{\partial(6x_1 + 16x_2 + 8)}{\partial x_2} = 16 \qquad \xrightarrow{yields} \qquad H = \nabla^2 f\left(x_1^{k_1}, x_2^{k_2}\right) = \begin{pmatrix} 6 & 6 \\ 6 & 16 \end{pmatrix}$$

$$fx_2x_1 = \frac{\partial\partial f(x)}{\partial x_2 \partial x_1} = \partial(6x_1 + 16x_2 + 8)\partial x1 = 6$$

$$\xrightarrow{yields} \ H^{-1} = \frac{1}{60}\begin{pmatrix} 16 & -6 \\ -6 & 60 \end{pmatrix} = \begin{pmatrix} 0.27 & -0.1 \\ -0.1 & 0.1 \end{pmatrix}$$

Then:

$$X_1 \leftarrow X_0 - H^{-1}g \quad \xrightarrow{yields} \quad X_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 0.27 & -0.1 \\ -0.1 & 0.1 \end{pmatrix} g = \begin{pmatrix} 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 3.4 \\ 0.6 \end{pmatrix} = \begin{pmatrix} -2.4 \\ 0.4 \end{pmatrix}$$

2.4 SECOND ORDER FUNCTION

In this part we intend to analyse a second-order representation of function which mentioned below in a matrix form:

$$f(x_{1,}x_2) = \frac{1}{2}\,(x_1\ \ x_2)\begin{pmatrix} 6 & 20 \\ -8 & 16 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + (12\ \ 8)\begin{pmatrix} x_1 \\ x_2 \end{pmatrix};$$

In the first step we must calculate stationary points of function in a matrix form:

We know that for a second-order function with following matrix form:

$$G = \frac{1}{2}X^T HX - B^T X \quad \xrightarrow{yields} \; gradient - matrix = HX - B \quad where\;\; H\; is\; Hessian\; matrix$$

Then:

$$H = \begin{pmatrix} 6 & 20 \\ -8 & 16 \end{pmatrix}, B = \begin{pmatrix} 12 \\ 8 \end{pmatrix} \xrightarrow{yields} gradient = \begin{pmatrix} 6 & 20 \\ -8 & 16 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 12 \\ 8 \end{pmatrix} = \begin{pmatrix} 6x_1 + 20x_2 + 12 \\ -8x_1 + 16x_2 + 8 \end{pmatrix}$$

As you can see the result is completely different with previous process in part1 then let's calculate the stationary point in this case:

$$\xrightarrow{solve\ equations} \quad \begin{cases} x_1 = -0.125 \\ x_2 = -0.56 \end{cases}$$

<u>Symmetric Hessian Matrix Vs Asymmetrical Hessian matrix:</u>

$$(1)\ f(x_{1,}x_2) = \frac{1}{2}\,(x_1\ \ x_2)\begin{pmatrix} 6 & 20 \\ -8 & 16 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + (12\ \ 8)\begin{pmatrix} x_1 \\ x_2 \end{pmatrix};$$

$$(2)\ f(x_{1,}x_2) = \frac{1}{2}\,(x_1\ \ x_2)\begin{pmatrix} 6 & 6 \\ 6 & 16 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + (12\ \ 8)\begin{pmatrix} x_1 \\ x_2 \end{pmatrix};$$

Hussein Matrix is Second Derivatives of function, moreover symmetry indicates that the second-order derivatives are continuous. in our case hessian matrix is not symmetric, so its derivatives are not continuous and there is no Taylor expansion at all in this regard.

In additional, it is not possible to determine the behaviour of the stationary points based on our previous knowledge and I think that's it.

3 QUESTION #2

3.1 STEEPEST DESCENT

In this part we intend to find whole stationary points of a concave function which mentioned below:

x1_optimal =9.7692 determine by MATLAB and gives us F_optimal = **-75.6**

x2_optimal = 4.9996

$f(x_1, x_2) = x_1{}^2 - 10x_2 \cos(0.2\pi x_1) + x_2{}^2 - 15x_1 \cos(0.4\pi x_2)$;

In first stage of our process we calculate the gradient matrix of above function:

$$g = \nabla f\left(x_1^{k_1}, x_2^{k_2}\right) = \begin{pmatrix} 2x_1 + 2\pi x_2 \sin(0.2\pi x_1) - 15\cos(0.4\pi x_2) \\ 2x_2 - 10\cos(0.2\pi x_1) + 6\pi x_1 \sin(0.4\pi x_2) \end{pmatrix} = \begin{pmatrix} -15 \\ -10 \end{pmatrix}$$

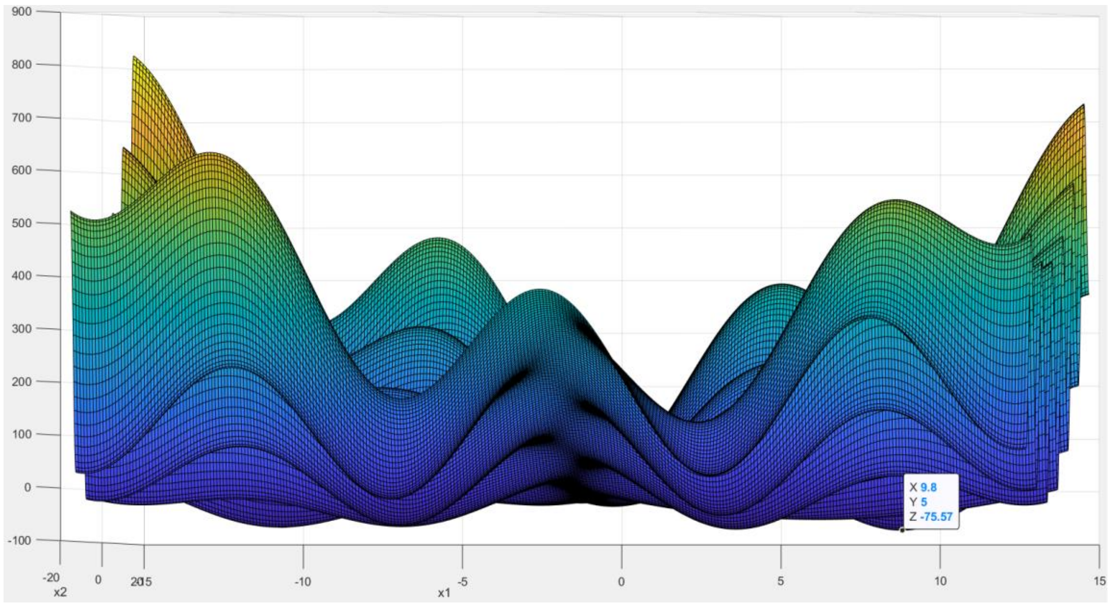
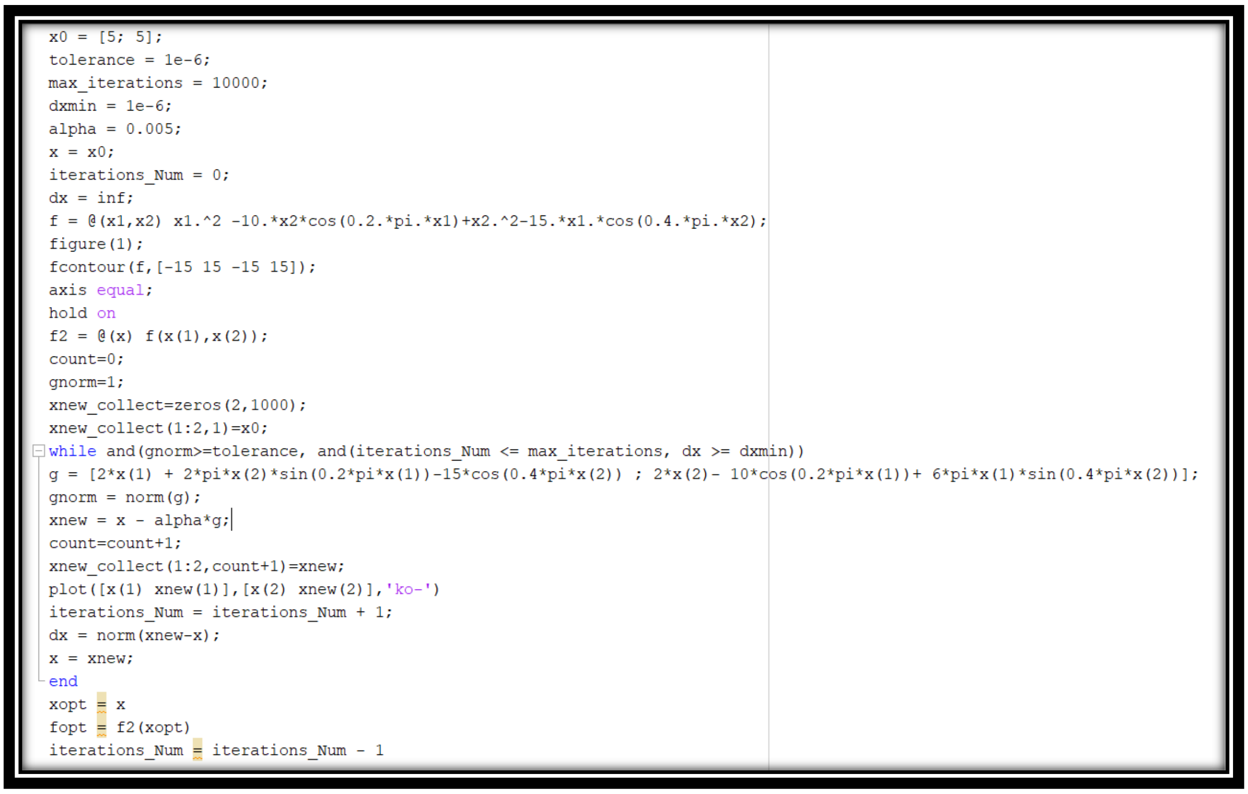


Figure 4: 3D representation of the Function

I found that the maximum of alpha we can use to don't miss any detailed information and reach to global minimum is **0.015**.
this threshold has been obtained by MATLAB and we can interpret that if we choose the step-size greater than mentioned threshold we can't reach to global minimum because step size in each stage helps us to check more points and investigating more point helps us to go ahead and don't miss any certain point and generally increasing the step size just causes that we analyse less than points and miss more useful information!



```
x0 = [5; 5];
tolerance = 1e-6;
max_iterations = 10000;
dxmin = 1e-6;
alpha = 0.005;
x = x0;
iterations_Num = 0;
dx = inf;
f = @(x1,x2) x1.^2 -10.*x2*cos(0.2.*pi.*x1)+x2.^2-15.*x1.*cos(0.4.*pi.*x2);
figure(1);
fcontour(f,[-15 15 -15 15]);
axis equal;
hold on
f2 = @(x) f(x(1),x(2));
count=0;
gnorm=1;
xnew_collect=zeros(2,1000);
xnew_collect(1:2,1)=x0;
while and(gnorm>=tolerance, and(iterations_Num <= max_iterations, dx >= dxmin))
g = [2*x(1) + 2*pi*x(2)*sin(0.2*pi*x(1))-15*cos(0.4*pi*x(2)) ; 2*x(2)- 10*cos(0.2*pi*x(1))+ 6*pi*x(1)*sin(0.4*pi*x(2))];
gnorm = norm(g);
xnew = x - alpha*g;
count=count+1;
xnew_collect(1:2,count+1)=xnew;
plot([x(1) xnew(1)],[x(2) xnew(2)],'ko-')
iterations_Num = iterations_Num + 1;
dx = norm(xnew-x);
x = xnew;
end
xopt = x
fopt = f2(xopt)
iterations_Num = iterations_Num - 1
```

Gradient descent implementation with MATLAB

In this section, we implement a gradient descent algorithm based on equations and materials which discussed in the class session.

I think the most important factor in this implementation is about finding suitable function error and also determine an adequate and absolute threshold for hitting suitable convergence rate.

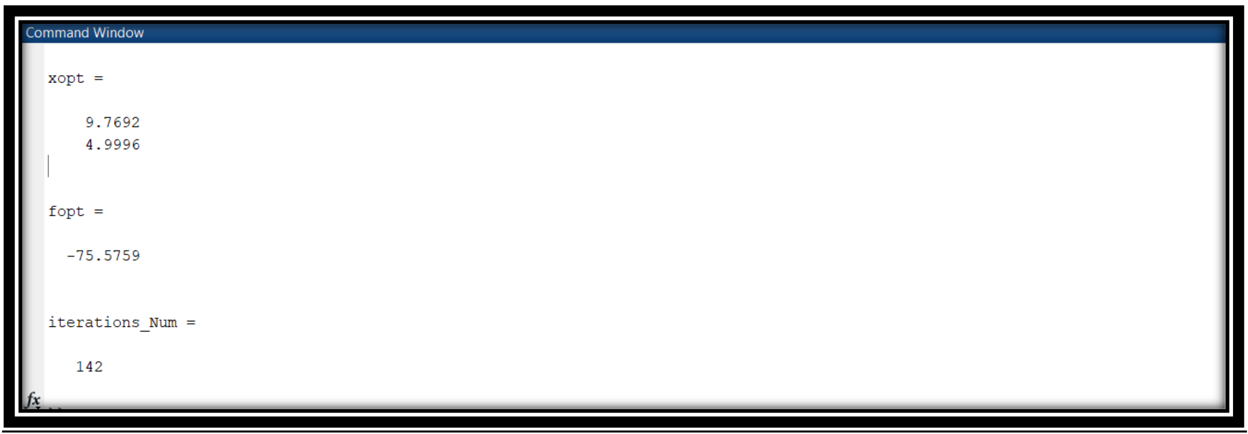
Note that in this regard we face with a concave function as you can see 3D plot in Figure 4, so after a few attempts, I found that we must drive the algorithm with an initial point that is near the ideal point (10,5) so with investigating 3D figure I use point (5,5) to reach desired convergence rate as well as we supposed and also in this way, we dominate on the local minimum that algorithm would be getting in trouble with these kinds of stuffs and couldn't get the global minimum as of our target!

I also use a reasonable threshold value and also a suitable initial alpha in my implementation and I had reached to the expected point (9.7,4.9) with a corresponding f-value which is (-75.57).

In addition, a considerable note is about the number of iterations and convergence rate which I met as well as I supposed with 142 Number of iterations!

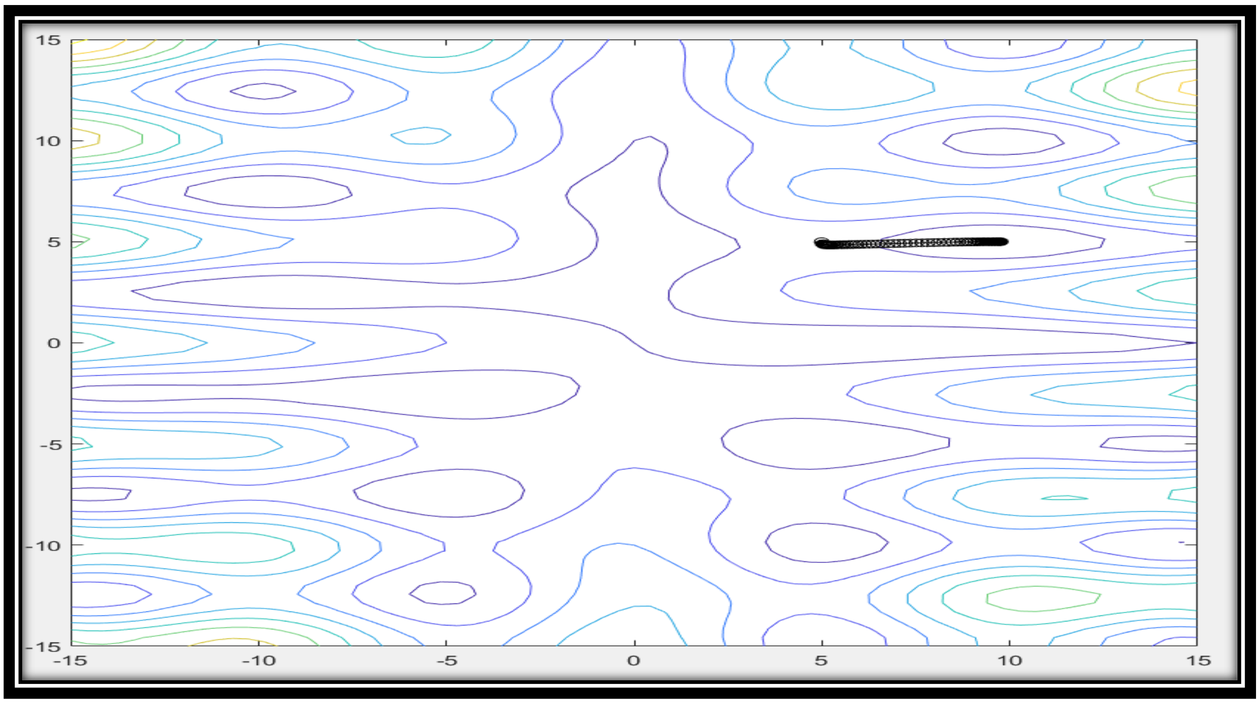
For running this part please have an accurate look on related directory and There you can easily find all of you need and also I would appreciate it if you would consider them ☺

Some of results have been provided below:



```
xopt =

    9.7692
    4.9996


fopt =

  -75.5759


iterations_Num =

   142
```

Global minimum with Corresponding F-value and Number of iterations



representation of the contour and points

3.2 Step length in Gradient Descent

- Exact Line search method for calculating the step size: (theoretical analyse)

$$g = \nabla f\left(x_1^{k_1}, x_2^{k_2}\right) = \begin{pmatrix} 2x_1 + 2\pi x_2 \sin(0.2\pi x_1) - 15\cos(0.4\pi x_2) \\ 2x_2 - 10\cos(0.2\pi x_1) + 6\pi x_1 \sin(0.4\pi x_2) \end{pmatrix} = \begin{pmatrix} -15 \\ -10 \end{pmatrix}$$

Based on exact line search method we intend to minimise step-size so we substitute($x^k + a^k d^k$) term in our function then obtain minimum of $h(a)$ using differentiation operator:

<u>**Start with point (0,0)**</u>

Solve:

Step1:

$min_a\, h(a) := f\left(x^k + ad^k\right)$

$h(a) = f\left(x^k + ad^k\right) = (x_1^k + ad_1^k)^2 - 10\left(x_2^k + ad_2^k\right)\cos(0.2\pi(x_1^k + ad_1^k)) + (x_2^k + ad_2^k)^2 - 15\left(x_1^k + ad_1^k\right)\cos(0.4\pi(x_2^k + ad_2^k))$

Then we derive based on ***a*** from equation and then assigning to zero:

$\frac{\partial h(a)}{\partial a} = 2d_1^k(x_1^k + ad_1^k) - 10d_2^k\cos(0.2\pi(x_1^k + ad_1^k)) + 2\pi d_1^k(x_2^k + ad_2^k)\sin(0.2\pi(x_1^k + ad_1^k)) + 2d_2^k(x_2^k + ad_2^k) - 15d_1^k\cos(0.4\pi(x_2^k + ad_2^k)) + 6\pi d_2^k(x_1^k + ad_1^k))\sin(0.4\pi(x_2^k + ad_2^k)) = 0$

Then using **MATLAB**:

We know that alpha must be a positive value so despite global minimum occurs in negative values we ignore that.

We have several roots so first of all we must investigate each of them to find best alpha value!

$$\begin{cases} x_1^0 = 0\,, x_2^0 = 0 \\ d_1^0 = 15\,, d_2^0 = 10 \end{cases} \xrightarrow{yields} a = \begin{cases} 0.0681\ corresponding\ fvalue \rightarrow -13.99 \\ 0.5208\ corresponding\ fvalue \rightarrow -35.19 \\ -0.276\ corresponding\ fvalue \rightarrow -57.72 \end{cases}$$

According to above values we choose $\mathbf{a = 0.5208}$ which is the best local minimum in comparison with others.

Now we obtain updated x-value in steepest descent algorithm:

$$x_1^1 \leftarrow x_1^0 + a \times d_1^0 \quad \xrightarrow{yields} \quad x_1^1 = 0 + (+0.5208)(15) = 7.8114$$

$$x_2^1 \leftarrow x_2^0 + a \times d_2^0 \quad \xrightarrow{yields} \quad x_2^1 = 0 + (+0.5208)(10) = 5.2076$$

Step2:

$min_a\ h(a) := f(x^k + ad^k)$

$h(a) = f(x^k + ad^k) = (x_1^k + ad_1^k)^2 - 10(x_2^k + ad_2^k)\cos(0.2\pi(x_1^k + ad_1^k)) + (x_2^k + ad_2^k)^2 - 15(x_1^k + ad_1^k)\cos(0.4\pi(x_2^k + ad_2^k))$

Then we derive based on $\boldsymbol{a}$ from equation and then assigning to zero:

$$\frac{\partial h(a)}{\partial a} = 2d_1^k(x_1^k + ad_1^k) - 10d_2^k\cos(0.2\pi(x_1^k + ad_1^k)) + 2\pi d_1^k(x_2^k + ad_2^k)\sin(0.2\pi(x_1^k + ad_1^k)) + 2d_2^k(x_2^k + ad_2^k) - 15d_1^k\cos(0.4\pi(x_2^k + ad_2^k)) + 6\pi d_2^k(x_1^k + ad_1^k))\sin(0.4\pi(x_2^k + ad_2^k)) = 0$$

Then using **MATLAB**:

We know that alpha must be a positive value so despite global minimum occurs in negative values we ignore that.

We have several roots so first of all we must investigate each of them to find best alpha value!

$$\begin{cases} x_1^1 = 7.8114\,, x_2^1 = 5.2076 \\ d_1^1 = 30.96\,, d_2^1 = 46.45 \end{cases} \xrightarrow{yields} a = \begin{cases} 0.0073\ \ corresponding\ fvalue \rightarrow -46.746 \\ 0.1111\ corresponding\ fvalue \rightarrow -42.209 \\ 0.219\ corresponding\ fvalue \rightarrow -29.19 \end{cases}$$

According to above values we choose $\mathbf{a = 0.0073}$ which is the best local minimum in comparison with others.

Now we obtain updated x-value in steepest descent algorithm:

$$x_1^2 \leftarrow x_1^1 + a \times d_1^1 \quad \xrightarrow{yields} \quad x_1^2 = 7.8114 + (+0.0073)(30.96) = 8.0371$$

$$x_2^2 \leftarrow x_2^1 + a \times d_2^1 \quad \xrightarrow{yields} \quad x_2^2 = 5.2076 + (+0.0073)(46.45) = 4.8690$$

Start with point (10,7)

Solve:

Step1:

$min_a\, h(a) := f\left(x^k + ad^k\right)$

$h(a) = f\left(x^k + ad^k\right) = (x_1^k + ad_1^k)^2 - 10\left(x_2^k + ad_2^k\right)\cos(0.2\pi(x_1^k + ad_1^k)) + (x_2^k + ad_2^k)^2 - 15\left(x_1^k + ad_1^k\right)\cos(0.4\pi(x_2^k + ad_2^k))$

Then we derive based on ***a*** from equation and then assigning to zero:

$\frac{\partial h(a)}{\partial a} = 2d_1^k(x_1^k + ad_1^k) - 10d_2^k\cos(0.2\pi(x_1^k + ad_1^k)) + 2\pi d_1^k(x_2^k + ad_2^k)\sin(0.2\pi(x_1^k + ad_1^k)) + 2d_2^k(x_2^k + ad_2^k) - 15d_1^k\cos(0.4\pi(x_2^k + ad_2^k)) + 6\pi d_2^k(x_1^k + ad_1^k))\sin(0.4\pi(x_2^k + ad_2^k)) = 0$

Then using **MATLAB**:

We know that alpha must be a positive value so despite global minimum occurs in negative values we ignore that.

We have several roots so first of all we must investigate each of them to find best alpha value!

$$\begin{cases} x_1^0 = 10\,, x_2^0 = 7 \\ d_1^0 = -32.13\,, d_2^0 = -114.79 \end{cases} \xrightarrow{yields} a = \begin{cases} 0.0174\ \ corresponding\ fvalue \rightarrow -74.42 \\ 0.1046\ corresponding\ fvalue \rightarrow -56.22 \\ 0.1475\ corresponding\ fvalue \rightarrow -50.29 \end{cases}$$

According to above values we choose $\mathbf{a = 0.0174}$ which is the best local minimum in comparison with others.

Now we obtain updated x-value in steepest descent algorithm:

$x_1^1 \leftarrow x_1^0 + a \times d_1^0 \quad \xrightarrow{yields} \quad x_1^1 = 10 + (+0.0174)(-32.13) = 9.4419$

$x_2^1 \leftarrow x_2^0 + a \times d_2^0 \quad \xrightarrow{yields} \quad x_2^1 = 7 + (+0.0174)(-114.79) = 5.0063$

Step2:

$min_a \; h(a) := f\left(x^k + ad^k\right)$

$h(a) = f\left(x^k + ad^k\right) = (x_1^k + ad_1^k)^2 - 10\left(x_2^k + ad_2^k\right)\cos(0.2\pi(x_1^k + ad_1^k)) + (x_2^k + ad_2^k)^2 - 15\left(x_1^k + ad_1^k\right)\cos(0.4\pi(x_2^k + ad_2^k))$

Then we derive based on $\boldsymbol{a}$ from equation and then assigning to zero:

$\frac{\partial h(a)}{\partial a} = 2d_1^k(x_1^k + ad_1^k) - 10d_2^k\cos(0.2\pi(x_1^k + ad_1^k)) + 2\pi d_1^k(x_2^k + ad_2^k)\sin(0.2\pi(x_1^k + ad_1^k)) + 2d_2^k(x_2^k + ad_2^k) - 15d_1^k\cos(0.4\pi(x_2^k + ad_2^k)) + 6\pi d_2^k(x_1^k + ad_1^k))\sin(0.4\pi(x_2^k + ad_2^k)) = 0$

Then using **MATLAB**:

We know that alpha must be a positive value so despite global minimum occurs in negative values we ignore that.

We have several roots so first of all we must investigate each of them to find best alpha value!

$$\begin{cases} x_1^1 = 9.4419\,, x_2^1 = 5.0063 \\ d_1^1 = 6.9214\,, d_2^1 = -2.0301 \end{cases} \xrightarrow{yields} a = \begin{cases} 0.0260 \;\; corresponding\, fvalue \to -46.746 \\ 2.3721 \; corresponding\, fvalue \to 293.5 \end{cases}$$

According to above values we choose $\mathbf{a = 0.0260}$ which is the best local minimum in comparison with others.

Now we obtain updated x-value in steepest descent algorithm:

$$x_1^2 \leftarrow x_1^1 + a \times d_1^1 \qquad \xrightarrow{yields} \qquad x_1^2 = 9.4419 + (+0.0260)(6.9214) = 9.6220$$

$$x_2^2 \leftarrow x_2^1 + a \times d_2^1 \qquad \xrightarrow{yields} \qquad x_2^2 = 5.0063 + (+0.0260)(-2.0301) = 4.9535$$

- inexact Line search method **(Armijo Rule)** for calculating the step size: (theoretical analyse):

$$g = \nabla f\left(x_1^{k_1}, x_2^{k_2}\right) = \begin{pmatrix} 2x_1 + 2\pi x_2 \sin(0.2\pi x_1) - 15\cos(0.4\pi x_2) \\ 2x_2 - 10\cos(0.2\pi x_1) + 6\pi x_1 \sin(0.4\pi x_2) \end{pmatrix} = \begin{pmatrix} -15 \\ -10 \end{pmatrix}$$

Based on inexact line search method we intend to check below inequality in each cycle to reach absolute and expected value for alpha by multiplying in a constant coefficient in each step of loop.

Note that Armijo rule provides conditions for reaching inexact alpha which used at the beginning of each stage.

Checking below inequality in each cycle:

$if \quad f(x_k + ad_k) \le f(x_k) + a\epsilon\Delta f(x_k)'d_k \quad then\ Stop\ and\ return\ alpha\ as\ of\ our\ result$

$Else\ multiply\ beta\ in\ previous\ alpha\ and\ update\ value.$

$x^{k+1} \leftarrow x^k + a \times d^k$

Start with point (0,0)
Step1:

Solve:
Suppose that the initial value of the alpha is $\boldsymbol{a_0}$ then we start the Armijo algorithm to find best alpha value.

$$a_0 = \frac{2(f_k - f_{k-1})}{\Delta f(x_{k-1})' p_{k-1}} \quad \xrightarrow{yields} \quad a_0 = \frac{2[f(a_0 d_1^0, a_0 d_2^0) - f(0,0)]}{-d_1^0 d_1^0 - d_2^0 d_2^0}$$

$$\begin{cases} a_0\left(-d_1^0 d_1^0 - d_2^0 d_2^0\right) = 2\left[f\left(a_0 d_1^0, a_0 d_2^0\right) - f(0,0)\right] \\ f(x_1, x_2) = x_1^2 - 10x_2\cos(0.2\pi x_1) + x_2^2 - 15x_1\cos(0.4\pi x_2) \\ d_1^0 = 15 \quad , \quad d_2^0 = 10 \end{cases} \xrightarrow{yields} \boldsymbol{a_0 = 0.0815}$$

Note that above result obtained using MATLAB for solving the initial value of alpha which mentioned in class as of question description!

$$x_1^2 \leftarrow x_1^1 + a \times d_1^1 \quad \xrightarrow{yields} \quad x_1^2 = 0 + (+0.0815)(15) = 1.2220$$

$$x_2^2 \leftarrow x_2^1 + a \times d_2^1 \quad \xrightarrow{yields} \quad x_2^2 = 0 + (+0.0815)(10) = 0.8147$$

<u>Checking below inequality in each cycle:</u>

$if \quad f(x_k + ad_k) \le f(x_k) + a\epsilon\Delta f(x_k)'d_k$

Left hand side: $f(x_k + ad_k) =$ -13.24 **so the alpha=0.8147 is our final result!**

Right hand side: $f(x_k) + a\epsilon\Delta f(x_k)'d_k =$ -0.0026

Step2:

Solve:

Suppose that the initial value of the alpha is $\boldsymbol{a_0}$ then we start the Armijo algorithm to find best alpha value.

$$a_0 = \frac{2(f_k - f_{k-1})}{\Delta f(x_{k-1})' p_{k-1}} \xrightarrow{yields} a_0 = \frac{2[f(a_0 d_1^1, a_0 d_2^1) - f(1.22{,}0.81)]}{-d_1^1 d_1^1 - d_2^1 d_2^1}$$

$$\begin{cases} a_0\left(-d_1^1 d_1^1 - d_2^1 d_2^1\right) = 2\left[f\left(a_0 d_1^1, a_0 d_2^1\right) - f(1.22{,}0.81)\right] \\ f(\mathrm{x}_1, \mathrm{x}_2) = \mathrm{x}_1{}^2 - 10\mathrm{x}_2\cos(0.2\pi\mathrm{x}_1) + \mathrm{x}_2{}^2 - 15\mathrm{x}_1\cos(0.4\pi\mathrm{x}_2) \\ d_1^1 = 1.8027 \quad , \quad d_2^1 = -14.1081 \end{cases} \xrightarrow{yields} \boldsymbol{a_0 = 0.0516}$$

Note that above result obtained using MATLAB for solving the initial value of alpha which mentioned in class as of question description!

$$x_1^2 \leftarrow x_1^1 + a \times d_1^1 \xrightarrow{yields} x_1^2 = 1.2220 + (+0.0516)(1.8027) = 1.3150$$

$$x_2^2 \leftarrow x_2^1 + a \times d_2^1 \xrightarrow{yields} x_2^2 = 0.8147 + (+0.0516)(-14.1081) = 0.0865$$

<u>Checking below inequality in each cycle:</u>

$if \quad f(x_k + ad_k) \le f(x_k) + a\epsilon\Delta f(x_k)'d_k$

Left hand side: $f(x_k + ad_k) =$ -18.4591 **so the alpha=0.0516 is our final result!**

Right hand side: $f(x_k) + a\epsilon\Delta f(x_k)'d_k =$ -13.2394

3.3 SIMULATED-ANNEALING

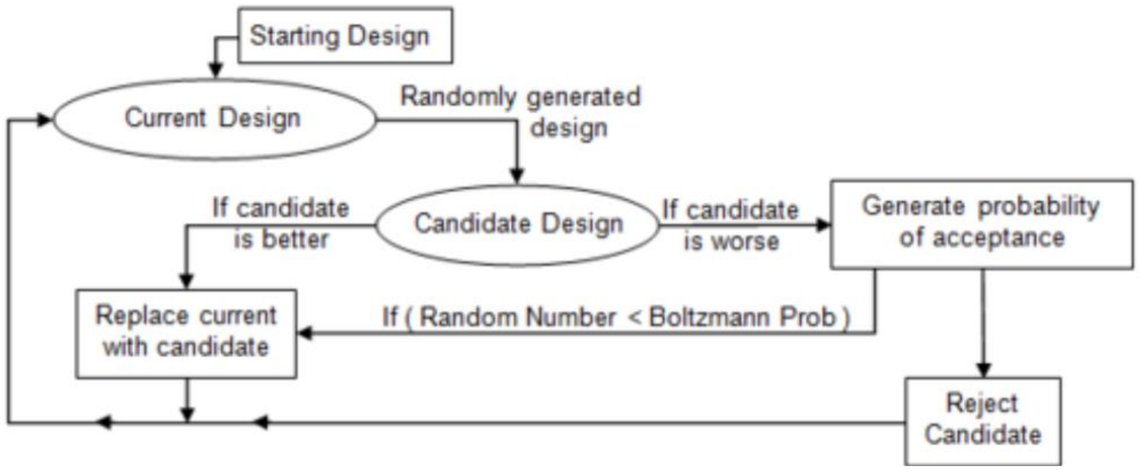
Theory behind Simulated-Annealing:

Simulated annealing copies a phenomenon in nature-the annealing of solids to optimize a complex system. Annealing refers to heating a solid and then cooling it slowly. Atoms then assume a nearly globally minimum energy state. In 1953 Metropolis created an algorithm to simulate the annealing process. The algorithm simulates a small random displacement of an atom that results in a change in energy. If the change in energy is negative, the energy state of the new configuration is lower and the new configuration is accepted. If the change in energy is positive, the new configuration has a higher energy state; however, it may still be accepted according to the Boltzmann probability factor:

$$P = exp(-\frac{\Delta E}{k_b T})$$

where k_b is the Boltzmann constant and T is the current temperature. By examining this equation, we should note two things: the probability is proportional to temperature as the solid cools, the probability gets smaller; and inversely proportional to as the change in energy is larger the probability of accepting the change gets smaller.

When applied to engineering design, an analogy is made between energy and the objective function. The design is started at a high “temperature”, where it has a high objective (we assume we are minimizing). Random perturbations are then made to the design. If the objective is lower, the new design is made the current design; if it is higher, it may still be accepted according the probability given by the Boltzmann factor. The Boltzmann probability is compared to a random number drawn from a uniform distribution between 0 and 1; if the random number is smaller than the Boltzmann probability, the configuration is accepted. This allows the algorithm to escape local minima.


Starting Design
Current Design
Randomly generated design
If candidate is better
Candidate Design
If candidate is worse
Generate probability of acceptance
Replace current with candidate
If (Random Number < Boltzmann Prob)
Reject Candidate

flowchart of simulated annealing algorithm

As the temperature is gradually lowered, the probability that a worse design is accepted becomes smaller. Typically, at high temperatures the gross structure of the design emerges which is then refined at lower temperatures.

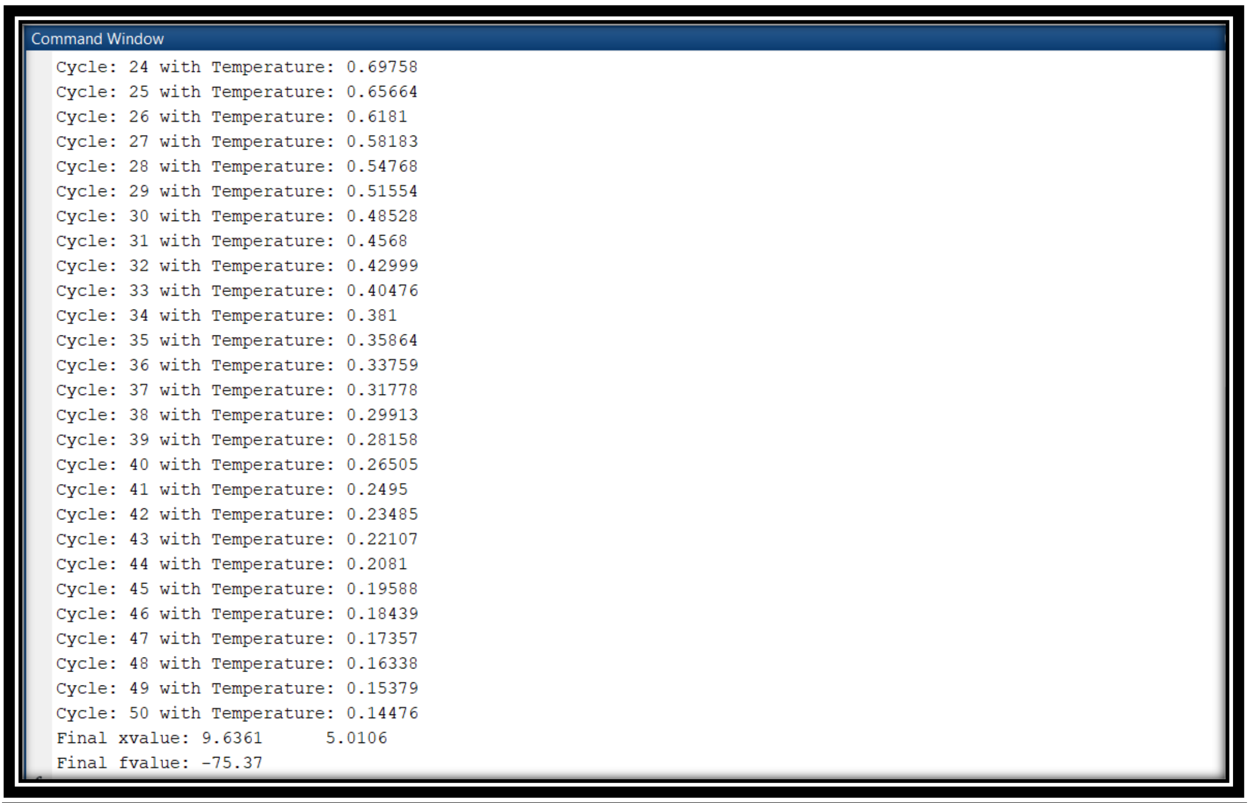
Although it can be used for continuous problems, simulated annealing is especially effective when applied to combinatorial or discrete problems. Although the algorithm is not guaranteed to find the best optimum, it will often find near optimum designs with many fewer design evaluations than other algorithms. (It can still be computationally expensive, however.) It is also an easy algorithm to implement.

Implementation in MATLAB:

In this section we intend to have an approximately implementation of simulated-annealing for finding global minimum of following function:

(based on the theory which mentioned in previous part.)

results have been provided below:

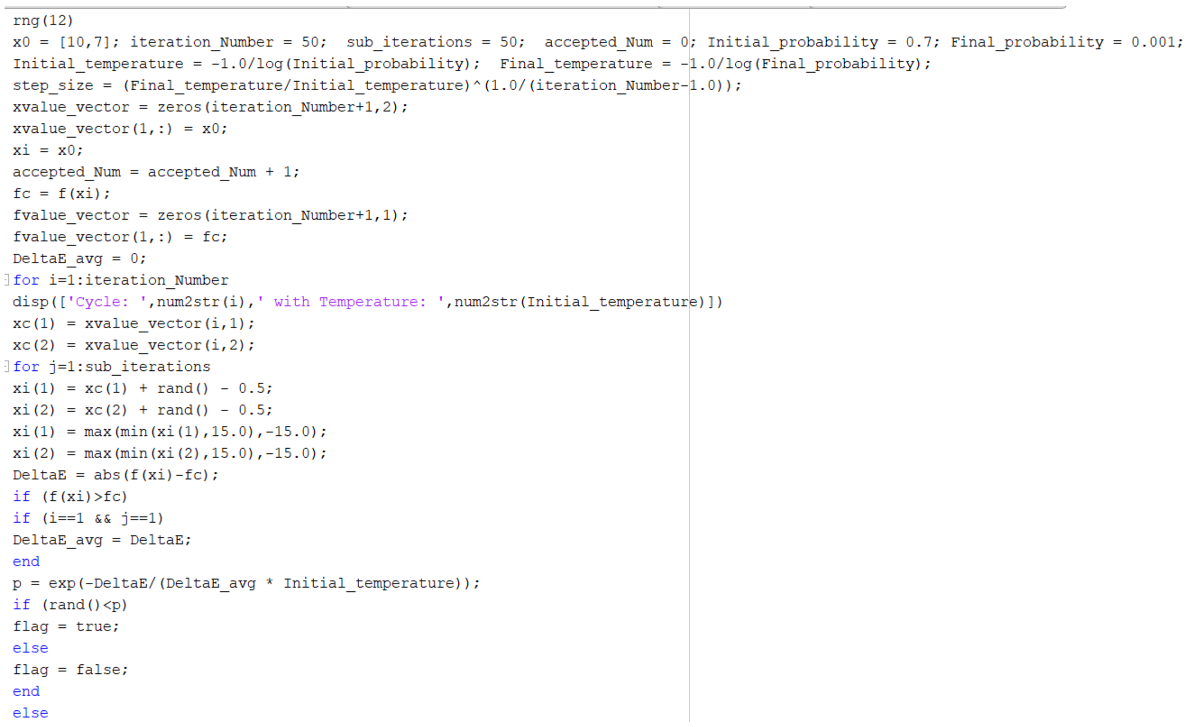


```
Cycle: 24 with Temperature: 0.69758
Cycle: 25 with Temperature: 0.65664
Cycle: 26 with Temperature: 0.6181
Cycle: 27 with Temperature: 0.58183
Cycle: 28 with Temperature: 0.54768
Cycle: 29 with Temperature: 0.51554
Cycle: 30 with Temperature: 0.48528
Cycle: 31 with Temperature: 0.4568
Cycle: 32 with Temperature: 0.42999
Cycle: 33 with Temperature: 0.40476
Cycle: 34 with Temperature: 0.381
Cycle: 35 with Temperature: 0.35864
Cycle: 36 with Temperature: 0.33759
Cycle: 37 with Temperature: 0.31778
Cycle: 38 with Temperature: 0.29913
Cycle: 39 with Temperature: 0.28158
Cycle: 40 with Temperature: 0.26505
Cycle: 41 with Temperature: 0.2495
Cycle: 42 with Temperature: 0.23485
Cycle: 43 with Temperature: 0.22107
Cycle: 44 with Temperature: 0.2081
Cycle: 45 with Temperature: 0.19588
Cycle: 46 with Temperature: 0.18439
Cycle: 47 with Temperature: 0.17357
Cycle: 48 with Temperature: 0.16338
Cycle: 49 with Temperature: 0.15379
Cycle: 50 with Temperature: 0.14476
Final xvalue: 9.6361      5.0106
Final fvalue: -75.37
```

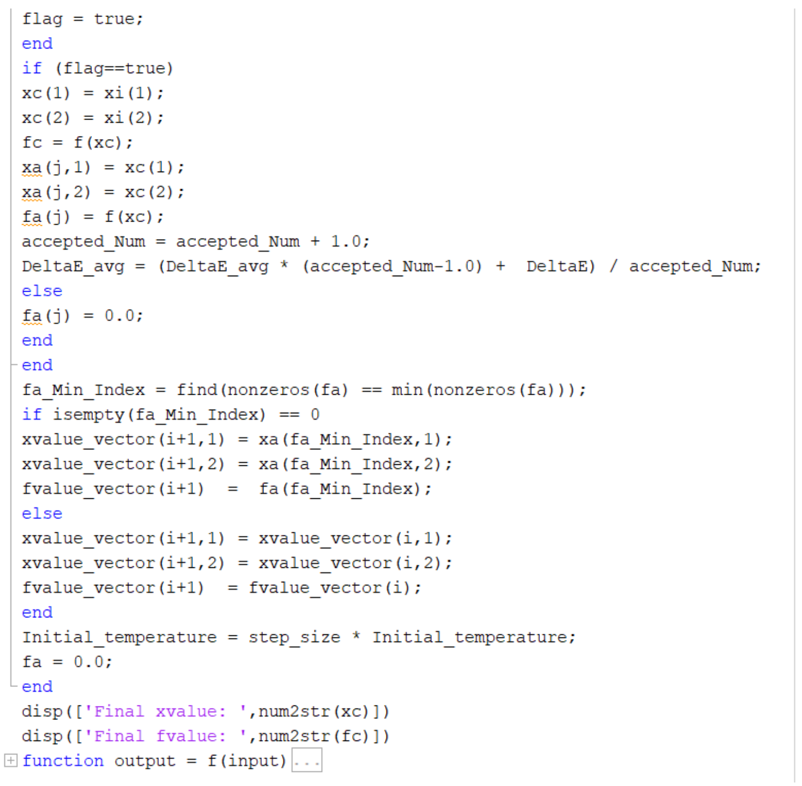
Results of Simulated annealing algorithm for minimizing our concave function

As you can see the results match as well as we expected with our previous analyses.

Here you can find all of my code implementation using MATLAB if there is any issue with that please let me know to provide more detailed information. (be in touch **here!**)



```
rng(12)
x0 = [10,7]; iteration_Number = 50;  sub_iterations = 50;  accepted_Num = 0; Initial_probability = 0.7; Final_probability = 0.001;
Initial_temperature = -1.0/log(Initial_probability);  Final_temperature = -1.0/log(Final_probability);
step_size = (Final_temperature/Initial_temperature)^(1.0/(iteration_Number-1.0));
xvalue_vector = zeros(iteration_Number+1,2);
xvalue_vector(1,:) = x0;
xi = x0;
accepted_Num = accepted_Num + 1;
fc = f(xi);
fvalue_vector = zeros(iteration_Number+1,1);
fvalue_vector(1,:) = fc;
DeltaE_avg = 0;
for i=1:iteration_Number
disp(['Cycle: ',num2str(i),' with Temperature: ',num2str(Initial_temperature)])
xc(1) = xvalue_vector(i,1);
xc(2) = xvalue_vector(i,2);
for j=1:sub_iterations
xi(1) = xc(1) + rand() - 0.5;
xi(2) = xc(2) + rand() - 0.5;
xi(1) = max(min(xi(1),15.0),-15.0);
xi(2) = max(min(xi(2),15.0),-15.0);
DeltaE = abs(f(xi)-fc);
if (f(xi)>fc)
if (i==1 && j==1)
DeltaE_avg = DeltaE;
end
p = exp(-DeltaE/(DeltaE_avg * Initial_temperature));
if (rand()<p)
flag = true;
else
flag = false;
end
else
```



```
flag = true;
end
if (flag==true)
xc(1) = xi(1);
xc(2) = xi(2);
fc = f(xc);
xa(j,1) = xc(1);
xa(j,2) = xc(2);
fa(j) = f(xc);
accepted_Num = accepted_Num + 1.0;
DeltaE_avg = (DeltaE_avg * (accepted_Num-1.0) +  DeltaE) / accepted_Num;
else
fa(j) = 0.0;
end
end
fa_Min_Index = find(nonzeros(fa) == min(nonzeros(fa)));
if isempty(fa_Min_Index) == 0
xvalue_vector(i+1,1) = xa(fa_Min_Index,1);
xvalue_vector(i+1,2) = xa(fa_Min_Index,2);
fvalue_vector(i+1)   =  fa(fa_Min_Index);
else
xvalue_vector(i+1,1) = xvalue_vector(i,1);
xvalue_vector(i+1,2) = xvalue_vector(i,2);
fvalue_vector(i+1)  = fvalue_vector(i);
end
Initial_temperature = step_size * Initial_temperature;
fa = 0.0;
end
disp(['Final xvalue: ',num2str(xc)])
disp(['Final fvalue: ',num2str(fc)])
function output = f(input)...
```

<u>Some explanation about parameters which have a key role in above implementation:</u>

1) initial value (10,7): because this point gives us a reasonable convergence rate!

2) iteration Number (50): I think it's enough to show the algorithm functionality!

3) Initial-probability (0.7): it represents Probability of accepting worse solution at the start

4) Final-probability (0.001): it represents Probability of accepting worse solution at the end.

5) Initial temperature: we obtain this parameter based on the formula discussed before.

6) Final temperature: we obtain this parameter based on the formula discussed before.

$$P = exp(-\frac{\Delta E}{k_b T})$$

where k_b is the Boltzmann constant and T is the current temperature!

7) Step-size: I found that this parameter acts just like a geometric sequences ratio and we use that to decrease the initial temperature in each step.

4 QUESTION #3

4.1 SVM THEORETICAL ANALYSE

In this part we obtain the support vector machine equations for below graph:

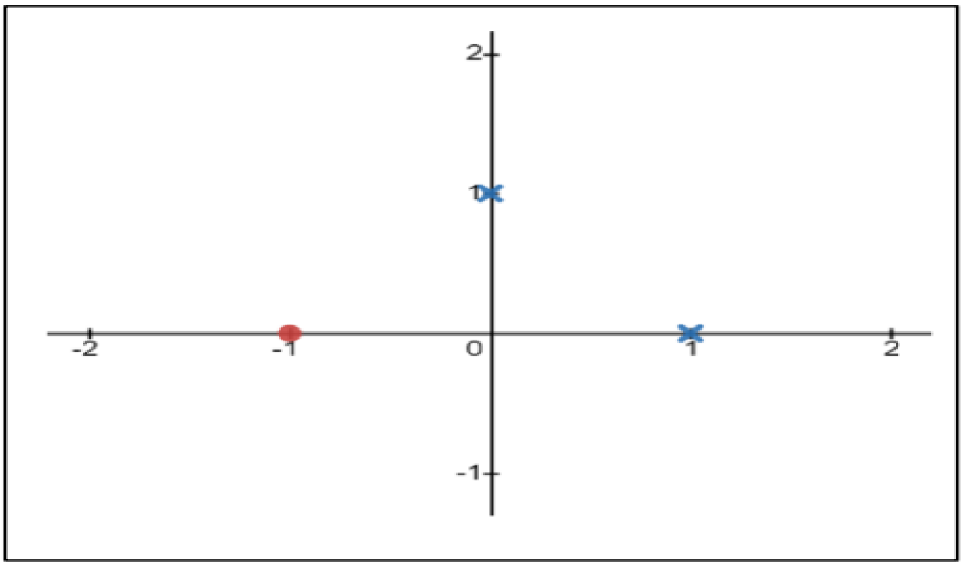


Figure 5: Data points consist of two separated group such as Blue & Red

Support vector machine algorithms are used in classification.

Classification can be viewed as the task of separating classes in feature space.

Solve:

$S_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ $\quad S_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ $\quad S_3 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$

Here we will use vectors augmented with a 1 as a bias input, and for clarity we will differentiate these with an over-tilde.

$\widetilde{S_1} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$ $\quad \widetilde{S_2} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$ $\quad \widetilde{S_3} = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}$

Then:

$$\begin{cases} a_1\widetilde{S_1}.\widetilde{S_1} + a_2\widetilde{S_2}.\widetilde{S_1} + a_3\widetilde{S_3}.\widetilde{S_1} = -1 \\ a_1\widetilde{S_1}.\widetilde{S_2} + a_2\widetilde{S_2}.\widetilde{S_2} + a_3\widetilde{S_3}.\widetilde{S_2} = -1 \\ a_1\widetilde{S_1}.\widetilde{S_3} + a_2\widetilde{S_2}.\widetilde{S_3} + a_3\widetilde{S_3}.\widetilde{S_3} = +1 \end{cases} \qquad \xrightarrow{yields}$$

$$a_1\begin{pmatrix}1\\0\\1\end{pmatrix}\cdot\begin{pmatrix}1\\0\\1\end{pmatrix} + a_2\begin{pmatrix}0\\1\\1\end{pmatrix}\cdot\begin{pmatrix}1\\0\\1\end{pmatrix} + a_3\begin{pmatrix}-1\\0\\1\end{pmatrix}\cdot\begin{pmatrix}1\\0\\1\end{pmatrix} = -1$$

$$a_1\begin{pmatrix}1\\0\\1\end{pmatrix}\cdot\begin{pmatrix}0\\1\\1\end{pmatrix} + a_2\begin{pmatrix}0\\1\\1\end{pmatrix}\cdot\begin{pmatrix}0\\1\\1\end{pmatrix} + a_3\begin{pmatrix}-1\\0\\1\end{pmatrix}\cdot\begin{pmatrix}0\\1\\1\end{pmatrix} = -1$$

$$a_1\begin{pmatrix}1\\0\\1\end{pmatrix}\cdot\begin{pmatrix}-1\\0\\1\end{pmatrix} + a_2\begin{pmatrix}0\\1\\1\end{pmatrix}\cdot\begin{pmatrix}-1\\0\\1\end{pmatrix} + a_3\begin{pmatrix}-1\\0\\1\end{pmatrix}\cdot\begin{pmatrix}-1\\0\\1\end{pmatrix} = +1 \qquad \xrightarrow{yields}$$

$$\begin{cases} 2a_1 + a_2 = -1 \\ a_1 + 2a_2 + a_3 = -1 \\ a_2 + 2a_3 = +1 \end{cases} \qquad \xrightarrow{yields} \qquad \begin{cases} a_1 = 0 \\ a_2 = -1 \\ a_3 = 1 \end{cases}$$

$$\xrightarrow{yields} \quad \widetilde{w} = \sum_{i=1}^{3} a_i\widetilde{S_3} = 0\begin{pmatrix}1\\0\\1\end{pmatrix} + (-1)\begin{pmatrix}0\\1\\1\end{pmatrix}. + (1)\begin{pmatrix}-1\\0\\1\end{pmatrix} = \begin{pmatrix}-1\\-1\\0\end{pmatrix}$$

Our vectors are augmented with a bias.

Hence we can equate the entry in $\widetilde{\boldsymbol{W}}$ as the hyper plane with an offset b.

Therefore the separating hyper plane equation $y = wx + b$ with $\mathrm{w} = \begin{pmatrix}-1\\-1\end{pmatrix}$ and offset $\mathrm{b} = 0$

Then : $-x - y + 0 = 0 \quad \xrightarrow{yields} \quad \boldsymbol{y = -x}$

4.2 IMPLEMENTING WITH SOFTWARE:

4.2.1 Implementing with Libraries!

In this part we implement Stochastic Gradient Descent(SGD) classifier using One Vs All algorithm and obtain all of below values according to question description:

1) Confusion Matrix

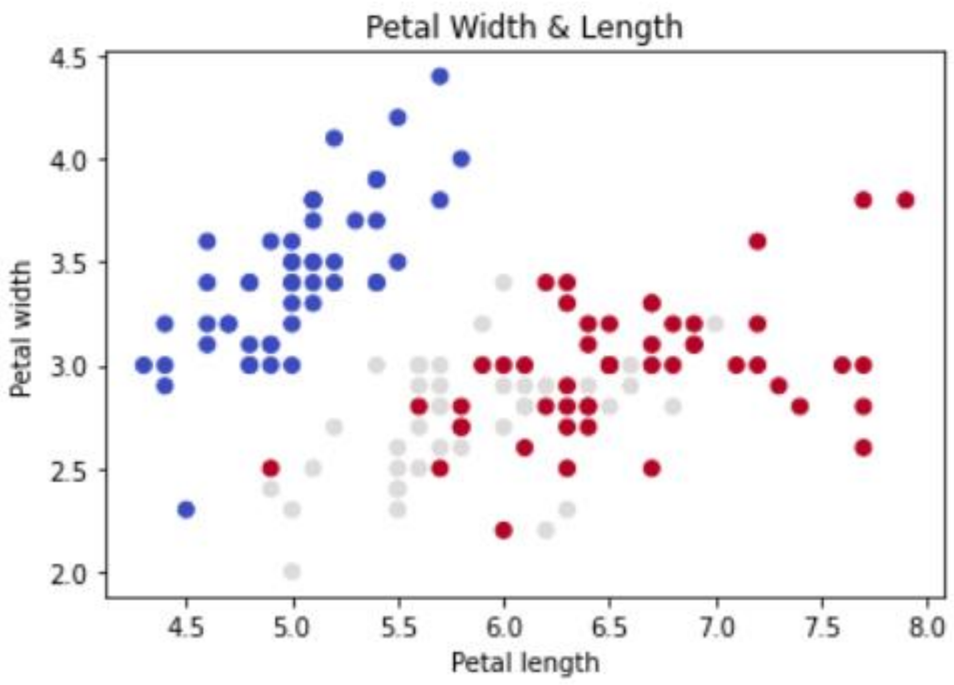
2) Confidence Matrix

3) Training data accuracy

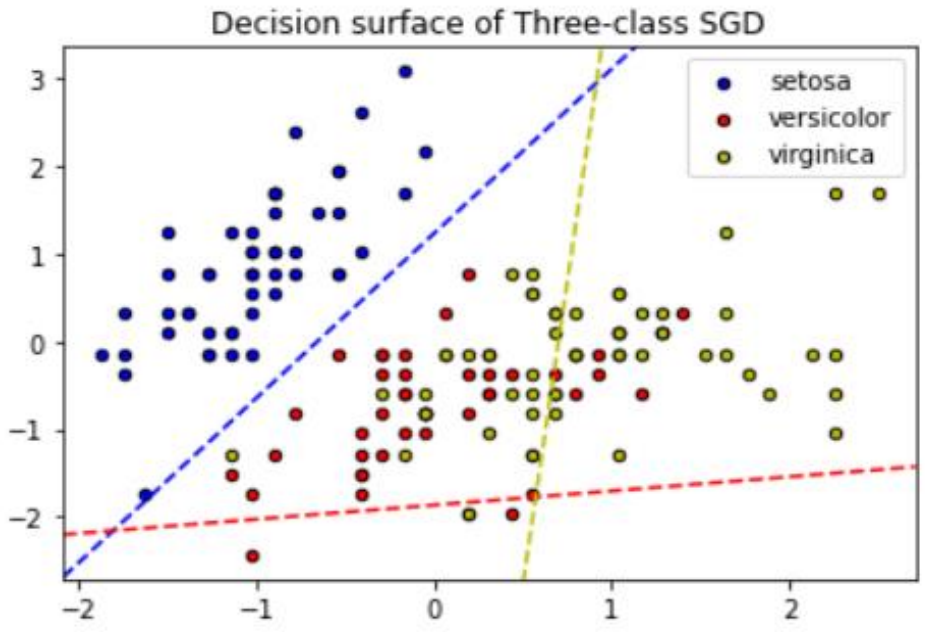
4) Determine and graphing the area of different classifications using One Vs All.

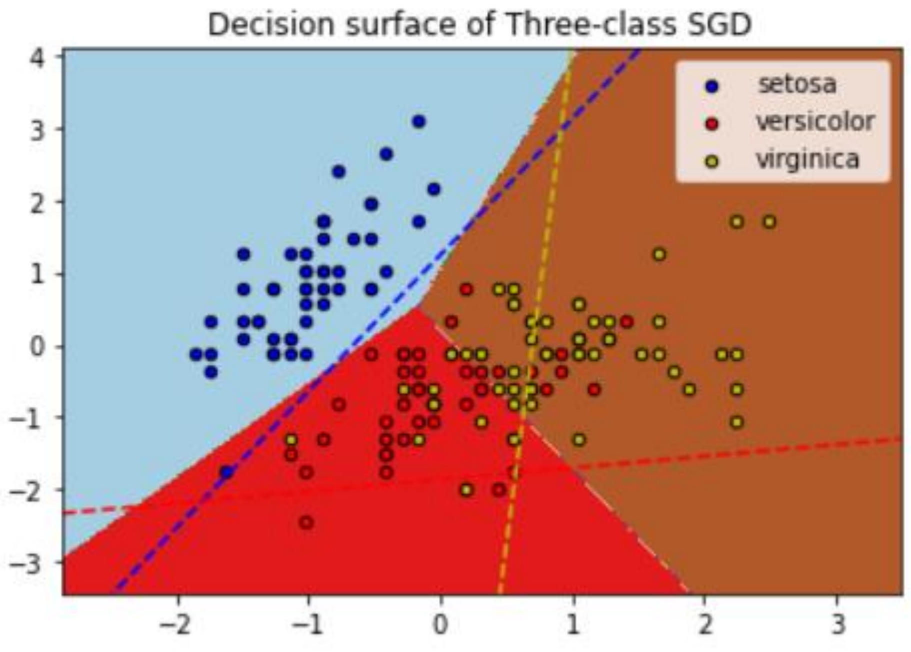
Implementing has been done in python and results have provided below:

Before classification:



After classification:





Confusion Matrix: (Obtained by Python)

$$\begin{bmatrix} \mathbf{50} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{22} & \mathbf{27} \\ \mathbf{0} & \mathbf{6} & \mathbf{44} \end{bmatrix}$$

Confidence Matrix: (Obtained by Python): it obtains with normalizing the Confusion Matrix in column.

$$\begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0.02} & \mathbf{0.44} & \mathbf{0.54} \\ \mathbf{0} & \mathbf{0.12} & \mathbf{0.88} \end{bmatrix}$$

Training data accuracy:

Accuracy=0.7733

Note that I use np.seed(810197494) command to get constant value in each attempt of running code!

Interpreting our findings:

As we know A confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class.

This is the key to the confusion matrix.The confusion matrix shows the ways in which your classification model is confused when it makes predictions. It gives us insight not only into the errors being made by our classifier but more importantly the types of errors that are being made. It is this breakdown that overcomes the limitation of using classification accuracy alone.

"true positive" for correctly predicted event values.

"false positive" for incorrectly predicted event values.

"true negative" for correctly predicted no-event values.

"false negative" for incorrectly predicted no-event values.

A confidence matrix is a confusion matrix where it has been normalized by column!

And at the end, Accuracy is one metric for evaluating classification models. Informally, accuracy is the fraction of predictions our model got right. Formally, accuracy has the following definition:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

For binary classification, accuracy can also be calculated in terms of positives and negatives as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Where TP = True Positives, TN = True Negatives, FP = False Positives, and FN = False Negatives.

Here you can find all of my code refer to part 4.2.1(implementing with Sklearn libraries).

```
import numpy as np
import matplotlib.pyplot as plt
import sklearn
from sklearn import datasets
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import accuracy_score
```

```
# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2]
y = iris.target
colors = "bry"
```

```
# plot unclassified data
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm)
plt.xlabel('Petal length')
plt.ylabel('Petal width')
plt.title('Petal Width & Length')
plt.show()


# shuffle
idx = np.arange(X.shape[0])
np.random.seed(13)
np.random.shuffle(idx)
X = X[idx]
y = y[idx]

# standardize
mean = X.mean(axis=0)
std = X.std(axis=0)
X = (X - mean) / std
```

```
# Plot also the training points
for i, color in zip(clf.classes_, colors):
    idx = np.where(y == i)
    plt.scatter(
        X[idx, 0],
        X[idx, 1],
        c=color,
        label=iris.target_names[i],
        cmap=plt.cm.Paired,
        edgecolor="black",
        s=20,
    )

plt.title("Decision surface of Three-class SGD")
plt.axis("tight")

# Plot the three one-against-all classifiers
xmin, xmax = plt.xlim()
ymin, ymax = plt.ylim()
coef = clf.coef_
intercept = clf.intercept_


def plot_hyperplane(c, color):
    def line(x0):
        return (-(x0 * coef[c, 0]) - intercept[c]) / coef[c, 1]

    plt.plot([xmin, xmax], [line(xmin), line(xmax)], ls="--", color=color)


for i, color in zip(clf.classes_, colors):
    plot_hyperplane(i, color)
plt.legend()
plt.show()
```

```python
h = 0.02  # step size in the mesh
svc = SGDClassifier(alpha=0.001, max_iter=100).fit(X, y)
# create a mesh to plot in
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
Z = svc.predict(np.c_[xx.ravel(), yy.ravel()]);
# Put the result into a color plot
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
test = svc.predict(np.c_[X[: , 0]  , X[: , 1]])
plt.axis("tight")
# Plot also the training points
for i, color in zip(clf.classes_, colors):
    idx = np.where(y == i)
    plt.scatter(
        X[idx, 0],
        X[idx, 1],
        c=color,
        label=iris.target_names[i],
        cmap=plt.cm.Paired,
        edgecolor="black",
        s=20,
    )

plt.title("Decision surface of Three-class SGD")
plt.axis("tight")

# Plot the three one-against-all classifiers
xmin, xmax = plt.xlim()
ymin, ymax = plt.ylim()
coef = clf.coef_
intercept = clf.intercept_


def plot_hyperplane(c, color):
    def line(x0):
        return (-(x0 * coef[c, 0]) - intercept[c]) / coef[c, 1]

    plt.plot([xmin, xmax], [line(xmin), line(xmax)], ls="--", color=color)


for i, color in zip(clf.classes_, colors):
    plot_hyperplane(i, color)
plt.legend()
plt.show()
```

```python
print(accuracy_score(test , y))
print(sklearn.metrics.confusion_matrix(y,test))
print(sklearn.metrics.confusion_matrix(y,test,normalize = 'true'))
```

4.2.2 Implementing Without Libraries!

In this part we implement classifier using One Vs All algorithm in both Stochastic Gradient descent & Steepest Descent(SD) using Armijo rule without using any library! and obtain all of below values according to question description:

Note that in this section I implement all of function just with using **numpy** and **matplotlib** for math and for plotting graphs!

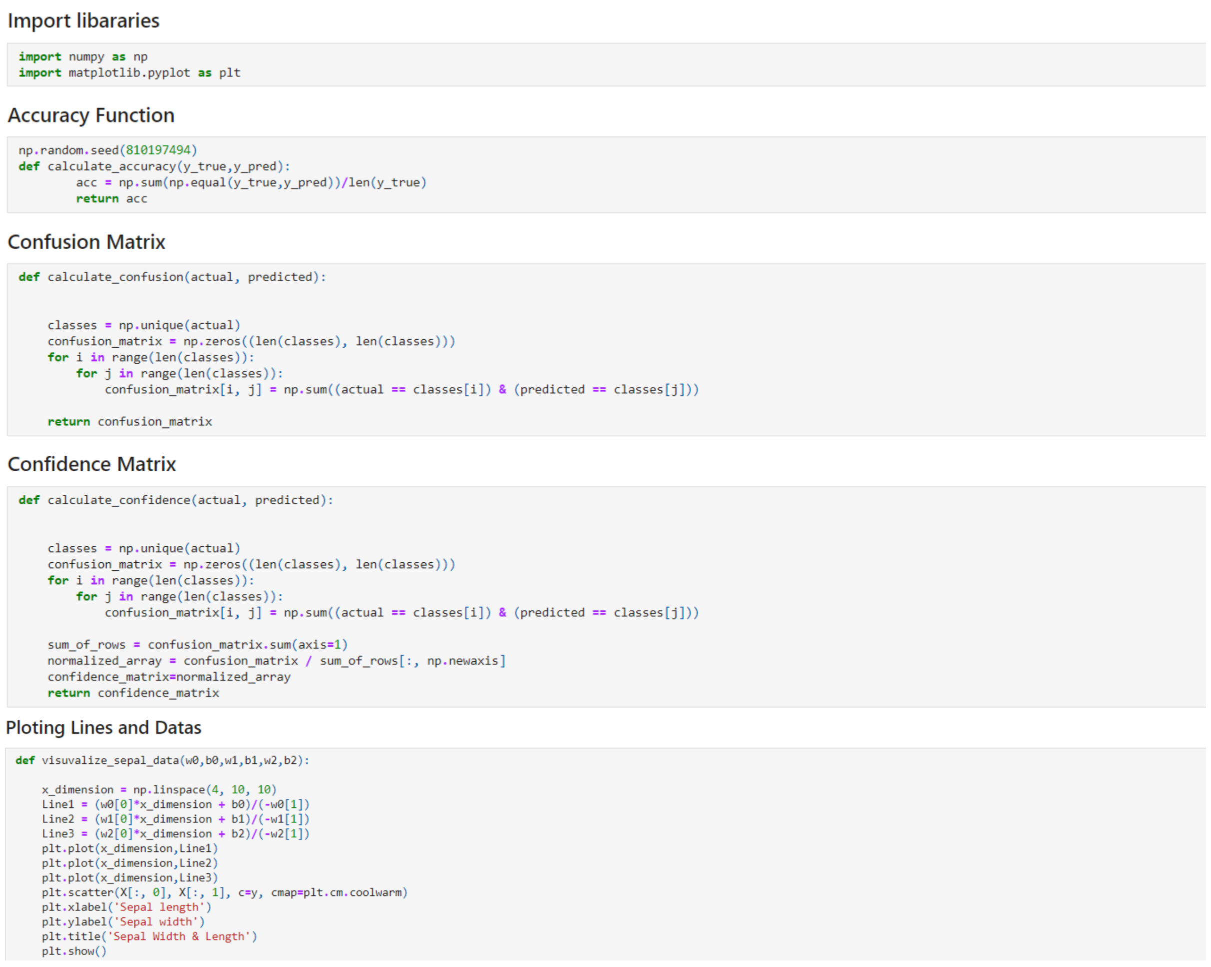
1) Confusion Matrix

2) Confidence Matrix

4) Determine and graphing the area of different classifications using One Vs All.

First: Implementing SGD

Implementing has been done in python and results have provided below:



Import libararies

```
import numpy as np
import matplotlib.pyplot as plt
```

Accuracy Function

```
np.random.seed(810197494)
def calculate_accuracy(y_true,y_pred):
        acc = np.sum(np.equal(y_true,y_pred))/len(y_true)
        return acc
```

Confusion Matrix

```
def calculate_confusion(actual, predicted):


    classes = np.unique(actual)
    confusion_matrix = np.zeros((len(classes), len(classes)))
    for i in range(len(classes)):
        for j in range(len(classes)):
            confusion_matrix[i, j] = np.sum((actual == classes[i]) & (predicted == classes[j]))

    return confusion_matrix
```

Confidence Matrix

```
def calculate_confidence(actual, predicted):


    classes = np.unique(actual)
    confusion_matrix = np.zeros((len(classes), len(classes)))
    for i in range(len(classes)):
        for j in range(len(classes)):
            confusion_matrix[i, j] = np.sum((actual == classes[i]) & (predicted == classes[j]))

    sum_of_rows = confusion_matrix.sum(axis=1)
    normalized_array = confusion_matrix / sum_of_rows[:, np.newaxis]
    confidence_matrix=normalized_array
    return confidence_matrix
```

Ploting Lines and Datas

```
def visuvalize_sepal_data(w0,b0,w1,b1,w2,b2):

    x_dimension = np.linspace(4, 10, 10)
    Line1 = (w0[0]*x_dimension + b0)/(-w0[1])
    Line2 = (w1[0]*x_dimension + b1)/(-w1[1])
    Line3 = (w2[0]*x_dimension + b2)/(-w2[1])
    plt.plot(x_dimension,Line1)
    plt.plot(x_dimension,Line2)
    plt.plot(x_dimension,Line3)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm)
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.title('Sepal Width & Length')
    plt.show()
```

Predict Function

```
def predict(data,w,b):
    predict_vector = []
    for i in range(data.shape[0]):
        if (np.dot(w, data[i,:])-b) > 0:
             predict_vector.append(1)
        else:
            predict_vector.append(-1)

        return predict_vector
```

Label convertor for One Vs all Algorithm

```
def label_convertor(label,class_Num):
    new_label = [1 if x==class_Num else -1 for x in label]
    return new_label
```

LosGradient for Stochastic gradient descent

```
def lossGradient(W,x,y,bias,lammbda):

    lossGradw = np.zeros_like(W)
    biaslossgrad = 0
    distance = np.max([0, 1 - y * ((W @ x)-bias)])
    if distance == 0:
        lossGradw = 2*lammbda*W
        biaslossgrad =0
    else:
        lossGradw = 2*lammbda*W -  y * x
        biaslossgrad = y

    return lossGradw,biaslossgrad
```

W & bias corresponed to each Class

```
def Wfinder(w,X,Class_label,bias,lammbda,steps,Learning_rate):
    for step in range(steps):
        for index ,result in enumerate(X):
            lossgradw_output , lossgradbias_output = lossGradient(w,result,Class_label[index],bias,lammbda)
            w = w - Learning_rate* lossgradw_output
            bias = bias - Learning_rate*lossgradbias_output

        return w,bias
```

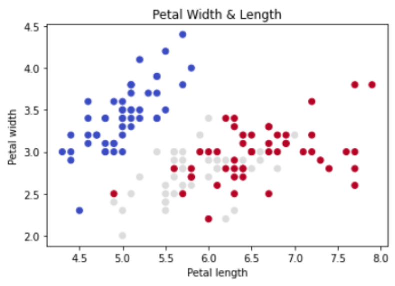
Start testing the program

Import some data to play with

```
iris = datasets.load_iris()
X = iris.data[:, :2]
y = iris.target
colors = "bry"
```

plot unclassified data

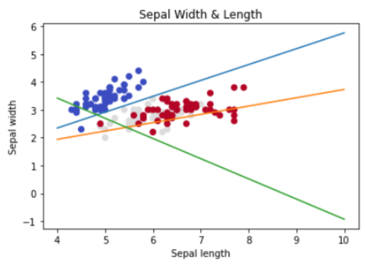
```
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm)
plt.xlabel('Petal length')
plt.ylabel('Petal width')
plt.title('Petal Width & Length')
plt.show()
```



```
label=y
Class1=label_convertor(label,0)
Class2=label_convertor(label,1)
Class3=label_convertor(label,2)

w = np.random.random(X.shape[1])
Learning_rate = 1
lammbda = 1e-4
steps = 1000
bias = 0

W0,bias0=Wfinder(w,X,Class1,bias,lammbda,steps,Learning_rate)
W1,bias1=Wfinder(w,X,Class2,bias,lammbda,steps,Learning_rate)
W2,bias2=Wfinder(w,X,Class3,bias,lammbda,steps,Learning_rate)

print(W0)
print(bias0)
print(W1)
print(bias1)
print(W2)
print(bias2)
```

Ploting Result

```
visuvalize_sepal_data(w0,b0,w1,b1,w2,b2)
```



Calculating Confusion & Confidence Matrix & Calculating Accuracy Score

Class One

```
Class1_label=predict(X,w0,b0)
accuracy=calculate_accuracy(y,Class1_label)
confusion_matrix=calculate_confusion(y,Class1_label)
confidence_matrix=calculate_confidence(y,Class1_label)
print(accuracy)
print(confusion_matrix)
print(confidence_matrix)
```

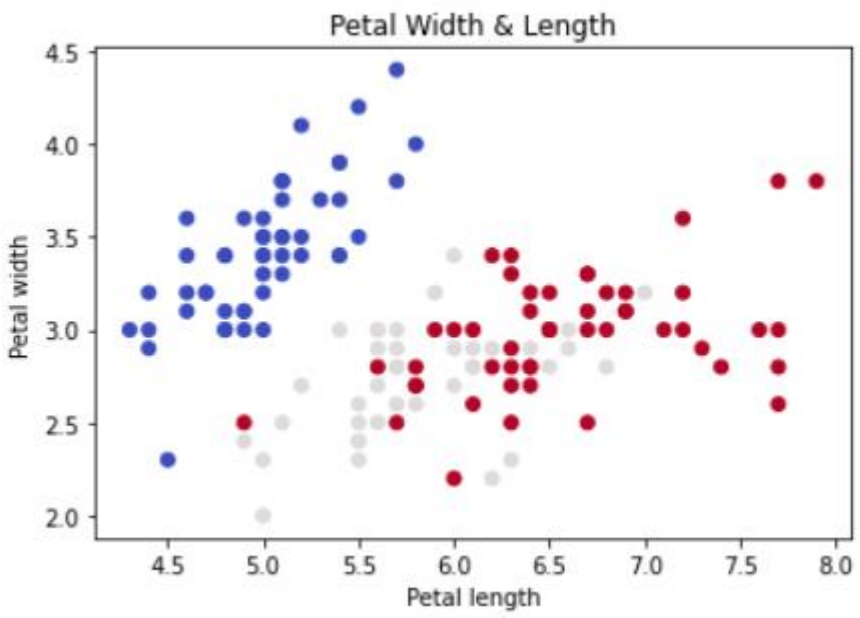
Class Two

```
Class2_label=predict(X,w1,b1)
accuracy=calculate_accuracy(y,Class1_label)
confusion_matrix=calculate_confusion(y,Class1_label)
confidence_matrix=calculate_confidence(y,Class1_label)
print(accuracy)
print(confusion_matrix)
print(confidence_matrix)
```

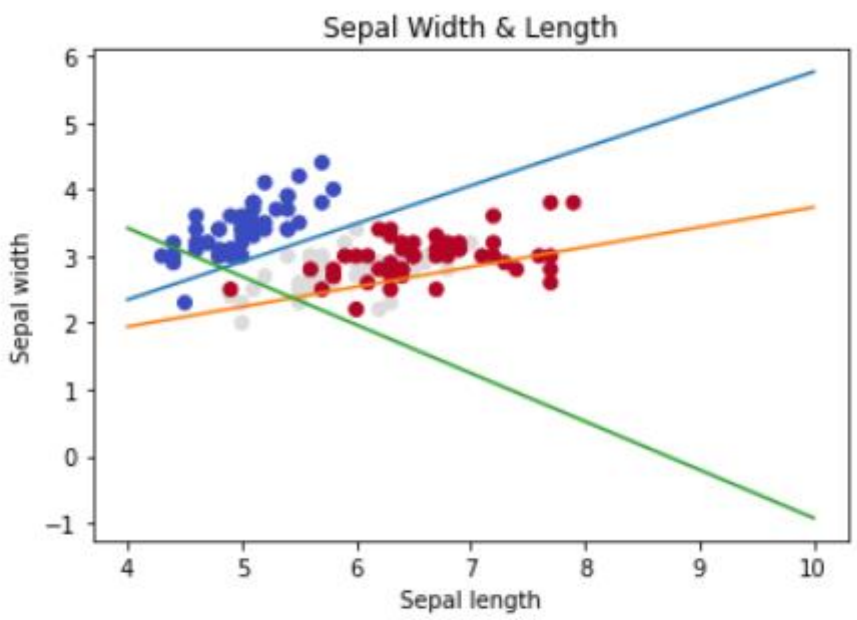
Class Three

```
Class3_label=predict(X,w2,b2)
accuracy=calculate_accuracy(y,Class1_label)
confusion_matrix=calculate_confusion(y,Class1_label)
confidence_matrix=calculate_confidence(y,Class1_label)
print(accuracy)
print(confusion_matrix)
print(confidence_matrix)
```

Before classification:



After classification:



Second: Implementing GD (Based on Armijo Rule)

Implementing has been done in python and results have provided below:

Steepest Descent Using Armijo Rule

```python
def gradient_Armojo(lambdaa, b,w,x_first,y_first):
    gradient_b = 0
    gradient_w = np.zeros(data.shape[1])
    for i in range(x_first.shape[0]):
        x = x_first[i]
        y= y_first[i]
        if (1 - y*(np.dot(w, x) - b)) > 0:
            gradient_b += y
            gradient_w += 2*lambdaa*w - y*x
        else:
            gradient_b += 0
            gradient_w += 2*lambdaa*w

    return gradient_b/x_first.shape[1], gradient_w/x_first.shape[1]
```

Steepest Descent gradient Function

```python
def Steepest_descent(lambdaa, train_data, train_labels):
    Mean = 0
    variance = 0.1
    steps = 100
    b = 0
    alpha = 10**-3
    w = np.random.normal(Mean,variance,train_data.shape[1])
    for step in range(steps):
        grad_b , grad_w = gradient_Armojo(lambdaa , b , w , train_data , train_labels)
        w = w - alpha*grad_w
        b = b - alpha*grad_b
    return w,b
```

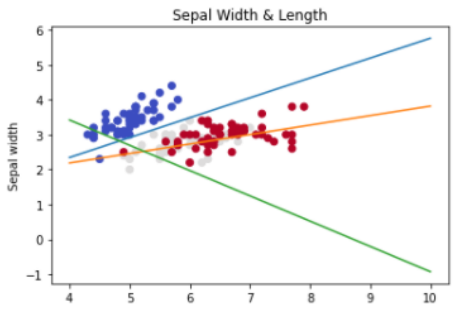
Testing Steepest Descent Using Armijo

```
data=X
train_data = data
label=y

Class1=label_convertor(label,0)
Class2=label_convertor(label,1)
Class3=label_convertor(label,2)

lam = 1e-2
wo, bo = gradient_descent(lam ,train_data,Class1)
w1, b1 = gradient_descent(lam ,train_data,Class2)
w3, b3 = gradient_descent(lam ,train_data,Class3)

print(wo)
print(b0)
print(w1)
print(b1)
print(w2)
print(b2)
```

```
[-0.96159501  1.64324628]
-0.15999999999999867
[ 0.17008052 -0.62637202]
-0.01850000000000001
[-52.98394628, -73.14606288]
462
```

Ploting Classifier for Armojo implementation

```
plot_classifier(w0,b0,w1,b1_,w2,b2)
```



Calculating Confusion & Confidence Matrix & Calculating Accuracy Score_Steepsest Descent

Class One

```
Class1_label=predict(X,w0,b0)
accuracy=calculate_accuracy(y,Class1_label)
confusion_matrix=calculate_confusion(y,Class1_label)
confidence_matrix=calculate_confidence(y,Class1_label)
```

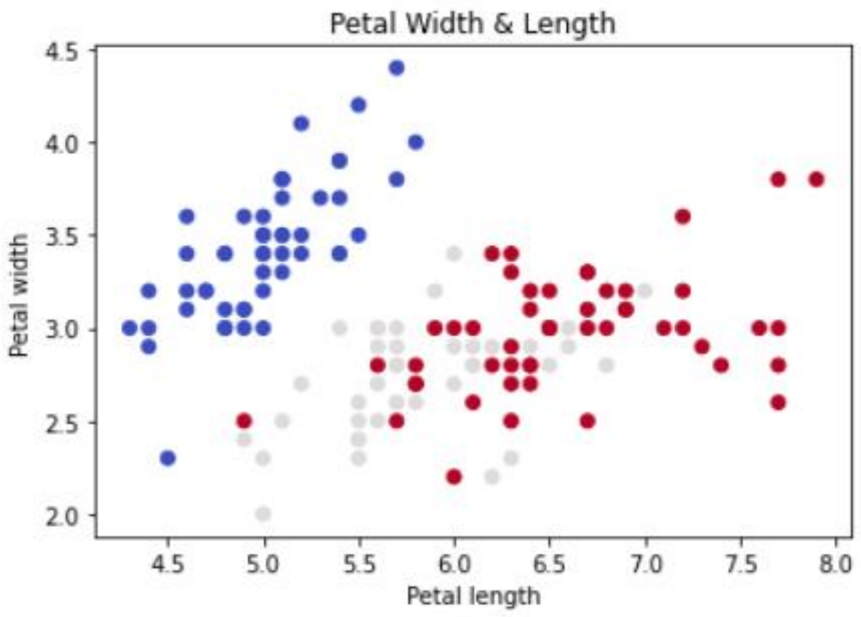
Class Two

```
Class2_label=predict(X,w1,b1_)
accuracy=calculate_accuracy(y,Class1_label)
confusion_matrix=calculate_confusion(y,Class1_label)
confidence_matrix=calculate_confidence(y,Class1_label)
```

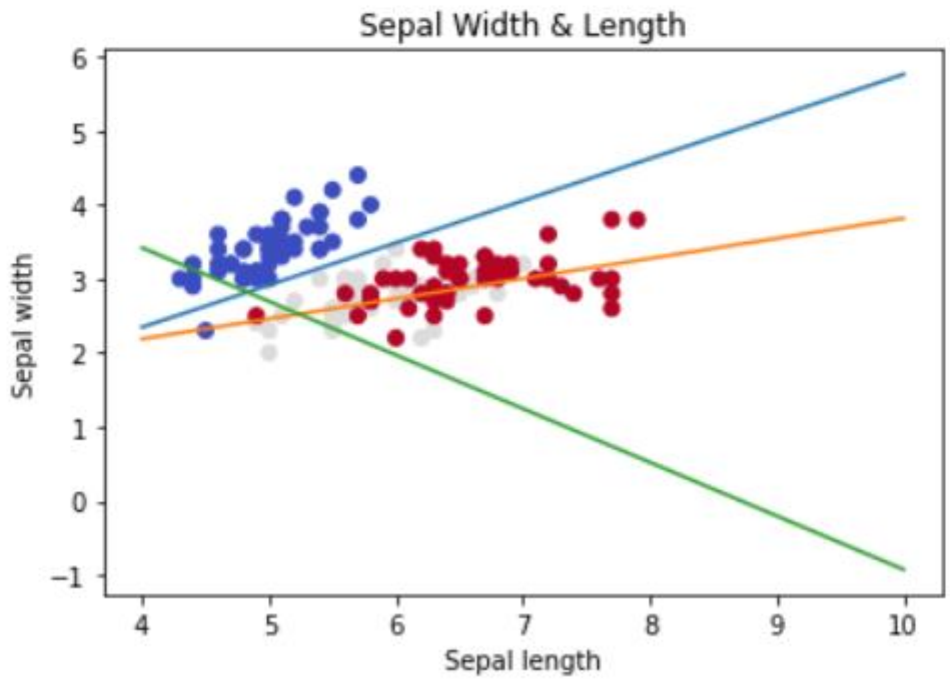
Class Three

```
Class2_label=predict(X,w2,b2)
accuracy=calculate_accuracy(y,Class1_label)
confusion_matrix=calculate_confusion(y,Class1_label)
confidence_matrix=calculate_confidence(y,Class1_label)
```

Before classification:



After classification:



For running this part please have an accurate look on related directory and There you can easily find all of you need and also I would appreciate it if you would consider them ☺

Conclusion:

We found that despite the second implementation without using libraries has very low accuracy but this implementation helped us to have a dipper investigation on discussed concepts and materials.

And as we expect the results of the sklearn implementation are better, but our result is still fine for a really simple model!

Final Note:

in some part of this project specially in part 1 & 2 I implement functions to process theoretical results in MATLAB which provided useful knowledge about numerical process which has been done behind the problems!

You can find all of my additional code in related directory, Thank you so much for your consideration ;)

5 ACKNOWLEDGEMENT

I am really grateful for Mr MohammadHossein vaeedi (810197605) because in some part of this project we had a nice and effective discussion which highly helped us to have a better analyse and also have more adequate results! (Note that we only talked about idea behind different problems.)

Afterwards, I am thankful to all of course teaching assistants: Mahsa Massoud (mahsamassoud@gmail.com) and Fateme Jafarian (jafarian.fateme7899@gmail.com) and Narjes Noorzad (njnoorzad@gmail.com) who designed this project with high quality.

6 REFERENCES

[1] https://machinelearningmastery.com/simulated-annealing-from-scratch-in-python/?__cf_chl_captcha_tk__=pmd_0XvvsjRpjFLQdr1K5orwB5hVql_hKSTb4qIuYsjqsSI-1635681138-0-gqNtZGzNAyWjcnBszQsl
[2] http://apmonitor.com/me575/index.php/Main/SimulatedAnnealing
[3]