Gradient Descent Algorithms

Name – Viswajith Menon

Net ID – VM623

- Take D= 10
- Write a function to generate a D-dimensional vector where each component is drawn from a standard normal distribution.
- Use this function to generate c.
- Generate a D×D matrix A, where every column is a vector generated in this way.
- Take Q = (A Transpose) * (A)

To compare the gradient step size across the different algorithms used I have generated a set of 5 values of Q and C. Denoted as Q1,Q2,Q3,Q4,Q5 and C1,C2,C3,C4,C5. I have used the same start point (x_start) for all the algorithms.

Generating the A, Q and C –

```
C=Generate_C()
A=Generate_Matrix()
A_t=A.transpose()
Q=np.matmul(A,A_t)
```

```
D=10

def Generate_C():
    C = np.random.normal(0,1, size=(D,1))
    return C

def Generate_Matrix():
    matrix = np.random.normal(0,1, size=(D,D))
    return matrix
```

Explain why Q is almost certainly positive definite when generated this way. Q is generated as a product of A and A_t (A transpose). This product is the norm of the vector A. As the norm is always positive, any vector generated as a product of a vector and it's transpose is always positive definite.

- For an $\alpha > 0$ small enough to guarantee convergence, implement gradient descent for this problem. Plot the error of $||\underline{x}_k - \underline{x}^*||$, and show that it agrees with the exponential convergence we expect from the results in class. *How can you verify this?*
- Additionally, we'd like to know in what manner the iterates converge to the minimum. In going from $\underline{x}_k$ to $\underline{x}_{k+1}$, are we aimed directly at the minimizer $\underline{x}^*$, or are we off slightly? We can understand this by looking at the angle between $\underline{x}_{k+1} - \underline{x}_k$ and $\underline{x}^* - \underline{x}_k$. To get at this angle, we can plot

$$\frac{[\underline{x}_{k+1} - \underline{x}_k]^{\mathrm{T}} [\underline{x}^* - \underline{x}_k]}{||\underline{x}_{k+1} - \underline{x}_k|| ||\underline{x}^* - \underline{x}_k||} \tag{6}$$

  as a function of k. What does the plot suggest about how the iterates approach the minimizer?
- Are the rates of convergence of the iterates and the behavior of the approach consistent across different starting points, stepsizes, and $Q, \underline{c}$ choices?

Constant Alpha –

I have chosen the alpha value as 0.03 to execute the gradient descent for X. First, we need to calculate X*, which is the unique minimizer.
X*= Inverse(Q).C

```
Q = np.array(np.random.choice([0, 0], size=(D,D), p=[1, 0]))
Q=np.matmul(A,A_t)
Q_inverse=np.linalg.inv(Q)
x_star=np.matmul(Q_inverse,C)
```

Now we will generate a random start point (X_start), which will be a Dx1 vector of values sampled from a normal distribution.
This same X_start will be used to compare the various gradient descents I will implement.

```
x_start=np.random.normal(0,1, size=(D,1))
```

In every interation X_new is calculated –

X_new=X_old−α∇F(X_old)

The error between X_new and X_star is calculated by the formula – X_new-X_star.

This returns another Dx1 vector. The norm of this vector is then calculated and stored in a list called ‘loss’.
The value of X_old is then made to X_new and process is repeated until the error is less than 0.01.

For every value of X_new, the angle is calculated with the formula given in equation (6) in the question. This angle is stored in a list called ‘angle’.

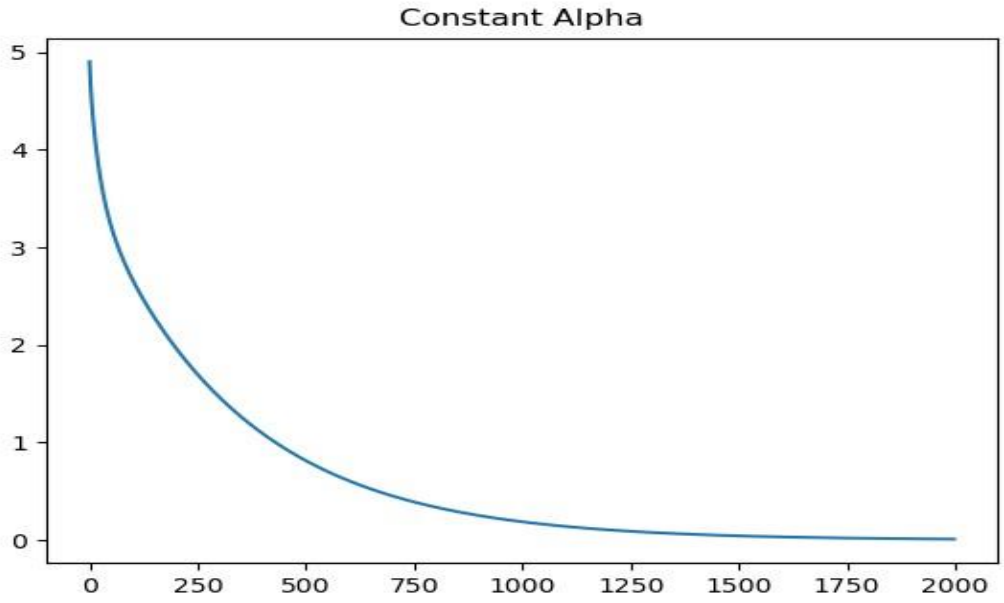
Once the error is below 0.01, the values from the list ‘loss’ and ‘angle’ are plotted to show convergence.

```
def Gradient_descent(x_start):
    x_old=x_start
    error=np.linalg.norm(np.subtract(x_old,x_star))
    count=0
    alpha=0.03
    loss=[]
    angle_list=[]
    while(error>0.01):
        diff=np.subtract(np.matmul(Q,x_old),C)                                  # Differential with respect to x.
        x_new=np.subtract(x_old,(np.dot(alpha,diff)))
        transpose_val=np.subtract(x_new,x_old)
        transpose_val=transpose_val.transpose()
        numerator=np.matmul(transpose_val,np.subtract(x_star,x_old))
        denominator=(np.linalg.norm(np.subtract(x_new,x_old)))*(np.linalg.norm(np.subtract(x_star,x_old)))
        angle=numerator/denominator                                             # Calculating the angle between (X(k+1)-X(k)) and X*-X(k))
        for i in angle:                                                         # Converting a list of list into a value
            b=i
        for i in b:
            angle=i
        angle_list.append(angle)
        error=np.linalg.norm(np.subtract(x_new,x_star))                         # Calculating the error from the true minimum
        loss.append(error)
        x_old=x_new
        count=count+1
```

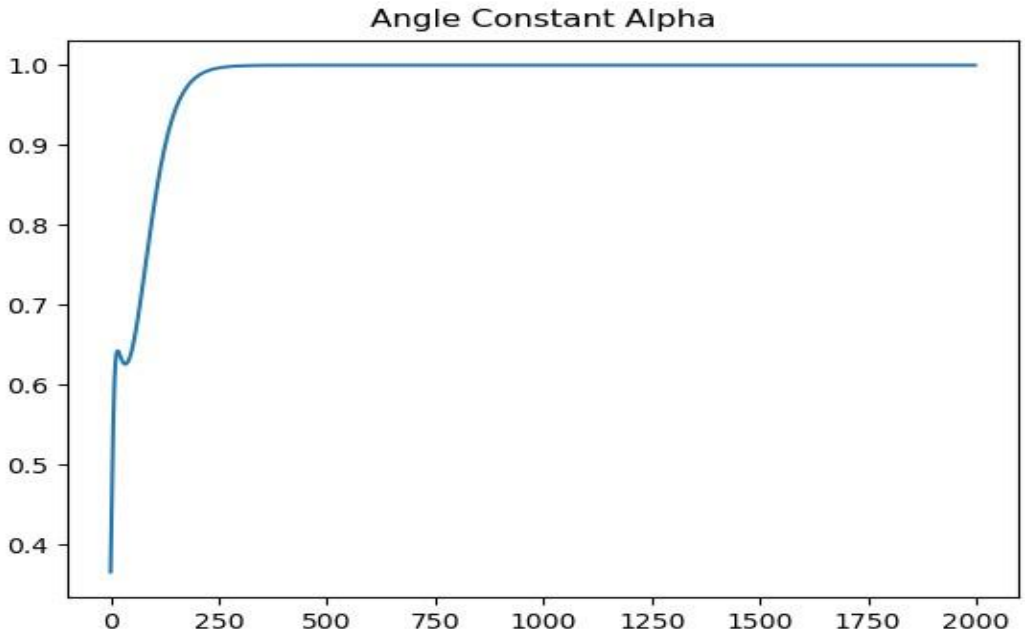
Number of steps taken to converge using this value of constant alpha –
1998
It is observed that the rate of convergence varies for different values of alpha and is dependent on the start point, Q and C.
If the error between the start point and the calculated x* is very high the iterates take a higher number of steps to converge to the minimum.

Alpha value = 0.03 Number of steps taken = 1998

Error Curve –



Angle Plot –



The number of steps taken for 5 different values of Q and C with the same x_start. (Alpha = 0.03)

Q1, C1 = 3724
Q2, C2 = 458
Q3, C3 = 10615
Q4, C4 = 5312
Q5, C5 = 3256

The angle plot suggests that the iterate at the beginning tries to angle itself towards the minimum and once it finds the correct angle it remains tilted towards that angle. The value of the plot reaches 1 to show that the dot product between the descent direction and the minimum is same.
When the angle between the descent direction and the minimum is same –
Dot product = u.v. cos(θ). (θ = 0 and cos (0) = 1.)

- Instead of taking α as a constant, take α_k to be the optimal stepsize for gradient descent as found previously.
- How does this change the rate of convergence? Be as specific as you can.
- How does this change the angle of approach as the iterates converge to the minimum?
- Are these behaviors consistent?

Optimal Alpha –

The optimal value of alpha calculated previously is –

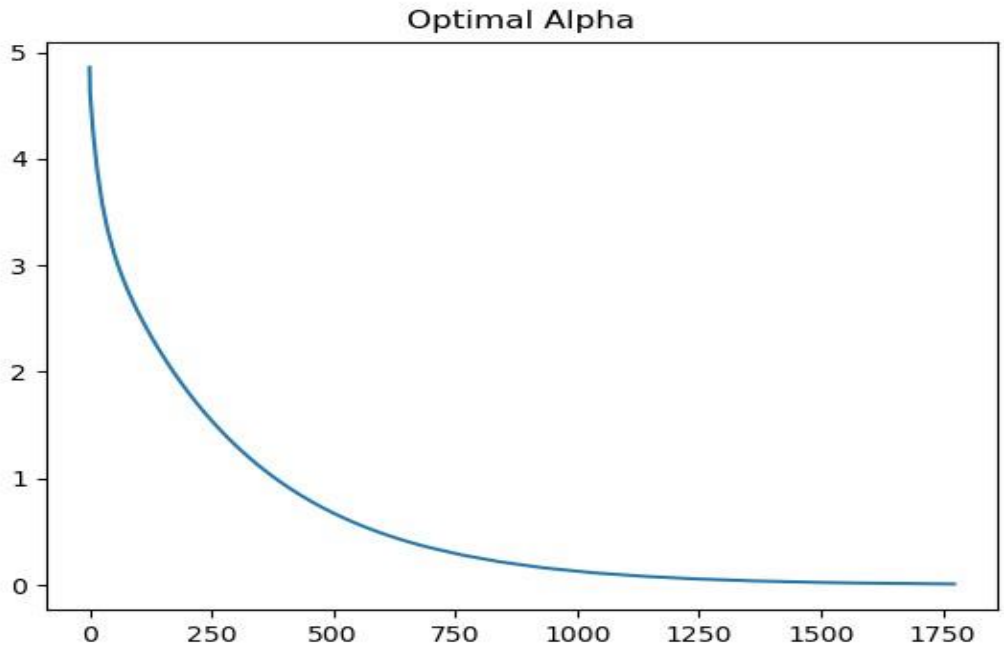
$$\alpha_k = \frac{p_k^T p_k}{p_k^T Q p_k}$$

We now plug this value of alpha to the existing code to calculate X_new and the angle.

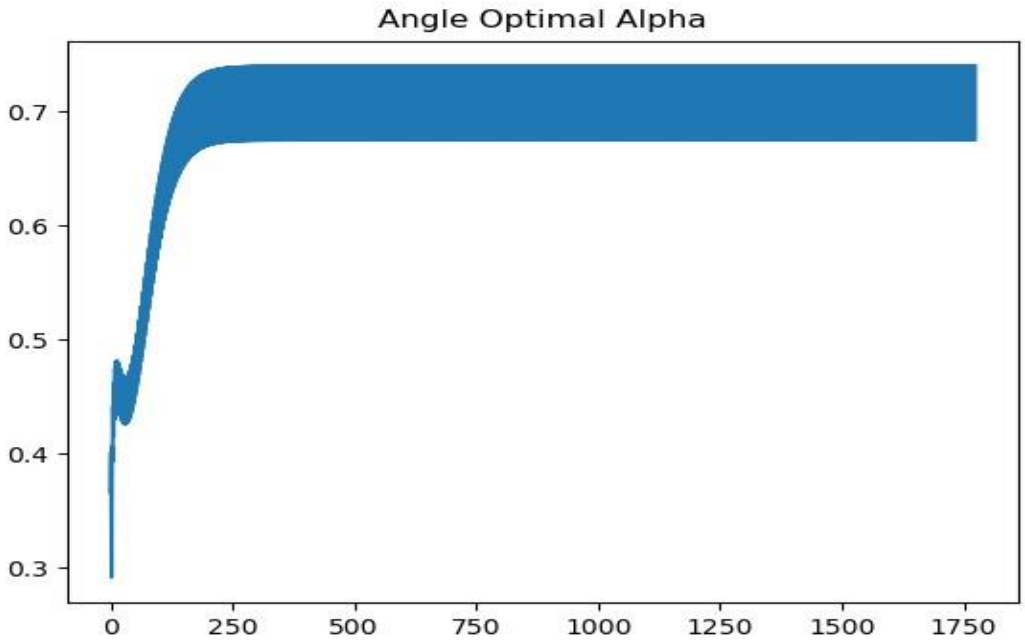
```
def Gradient_descent_alpha(x_start):
    x_old=x_start
    #print("x_start ",x_old)
    error=np.linalg.norm(np.subtract(x_old,x_star))
    count_o=0
    loss=[]
    angle_list=[]
    while(error>0.01):
        p_k=np.subtract(np.matmul(Q,x_old),C)                                   # Differential with respect to x.
        p_k_t=p_k.transpose()
        numerator=np.matmul(p_k_t,p_k)
        denominator_1=np.matmul(p_k_t,Q)
        denominator=np.matmul(denominator_1,p_k)
        a=numerator/denominator                                                 # Calculating the optimal alpha value at every step
        for i in a:
            b=i
        for i in b:
            alpha=i
        #print("alpha ",alpha)
        x_new=np.subtract(x_old,(np.dot(alpha,p_k)))                            # Calculating the error from the true minimum
        #Gradient Angle
        transpose_val=np.subtract(x_new,x_old)
        transpose_val=transpose_val.transpose()
        n=np.matmul(transpose_val,np.subtract(x_star,x_old))
        d=(np.linalg.norm(np.subtract(x_new,x_old)))*(np.linalg.norm(np.subtract(x_star,x_old)))
        angle=n/d                                                               # Calculating the angle between (X(k+1)-X(k)) and X*-X(k))
        #print(angle)
        for i in angle:
            b=i
        for i in b:
            angle=i
        angle_list.append(angle)
        #Gradient Angle
        error=np.linalg.norm(np.subtract(x_new,x_star))
        loss.append(error)
        #print("error ",error)
        x_old=x_new
        count_o=count_o+1
```

Number of steps taken = 1774

Error Curve –



Angle Plot –



Similar to using a constant alpha, the convergence is dependent on the randomly generated start point.

As shown in the graph's which were plotted, we can observe that taking an optimal value of alpha results in convergence in a smaller number of steps compared to taking a constant alpha.

The number of steps taken for 5 different values of Q and C with the same x_start.

Q1, C1 = 1767
Q2, C2 = 293
Q3, C3 = 4712
Q4, C4 = 1648
Q5, C5 = 1673

The angle plot behaves similar to the constant alpha case, but in this case it oscillates by a small margin as the value of alpha is constantly changing in every step which leads the angle to move up and down.

- For a constant $\alpha > 0, \beta > 0$, plot the error $||\underline{x}_k - \underline{x}^*||$ as a function of k to show convergence. How can you find β, α to guarantee convergence? Are these the best constants you can find?
- For the best α, β you can find in the above question, what can you say about the rate of convergence, and how does it compare to gradient descent? Can you find α, β to make the convergence rate better than vanilla gradient descent? How does it compare to optimized gradient descent?
- Again, plot the angle of approach to the minimizer for these momentum iterates. What can you say about the approach to the minimizer, and how does it compare to the previous results?
- Do the trends you observe above generalize, with $\alpha, \beta, Q, \underline{c}$? How does vanilla momentum compare with vanilla gradient descent? With optimized gradient descent?

Constant Alpha and Beta –

I have chosen the alpha value as 0.03 and beta value as 0.8 to execute the gradient descent for X.

The implementation remains relatively the same as gradient descent for constant alpha.

The only change is the calculation of X_New –

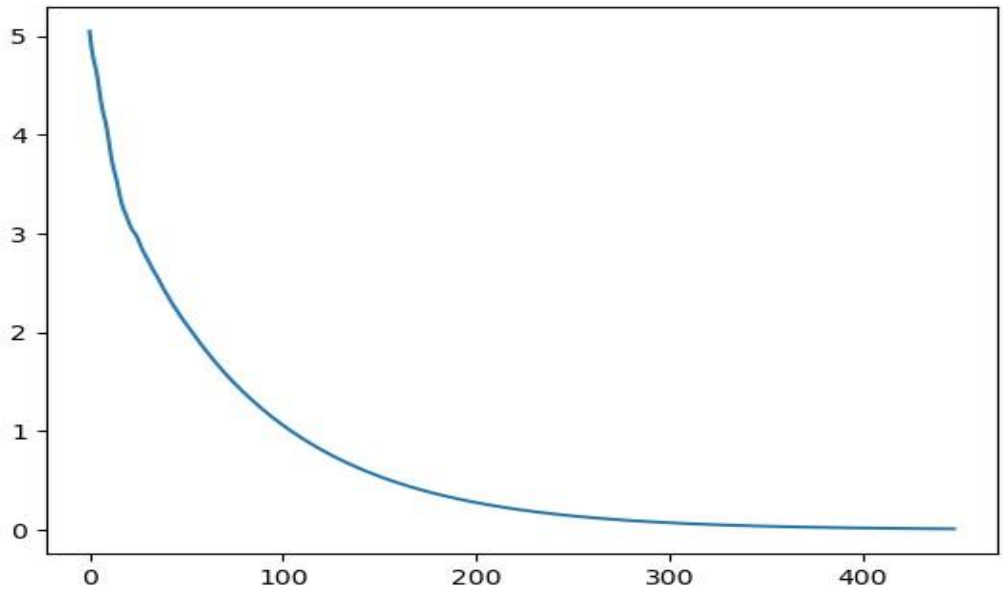
X_new = X_old – α(P_k) + β(Q_k)

P_k = differential(F(X_old))
Q_k = X_Old – X_Prev

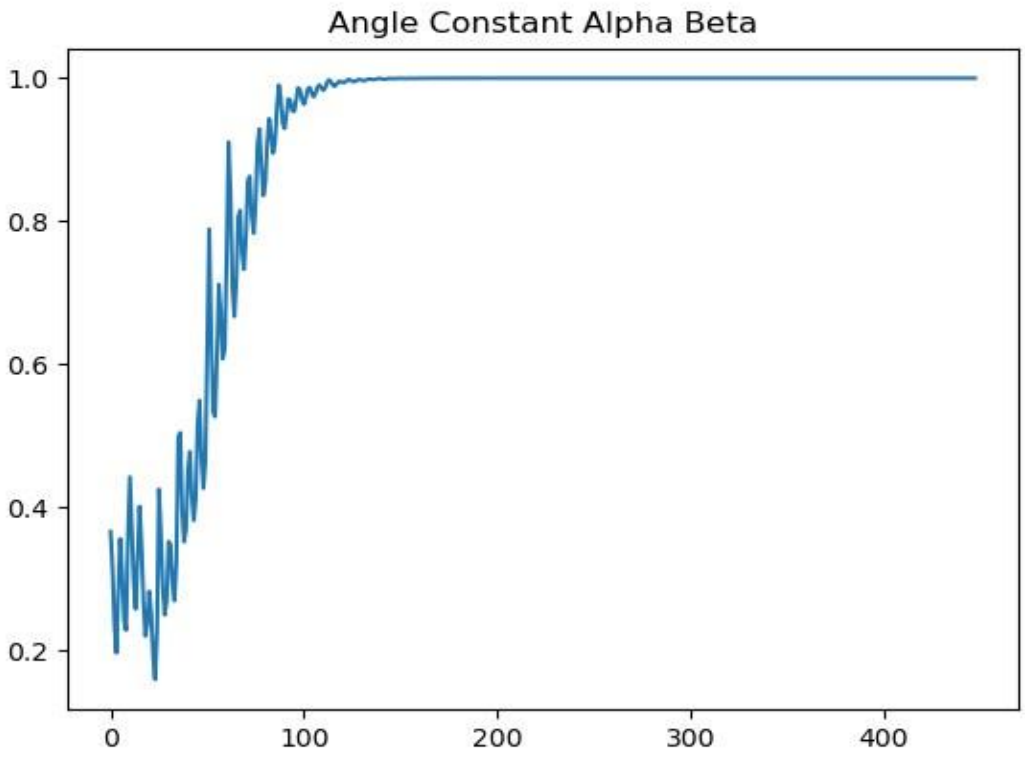
At step 'k', this moves the iterate from k to k+ 1 slightly in the direction it moved from k−1 to k, as though the iterate had some momentum pulling it in this direction.
The magnitude of this pull in the direction it moved previously is what is stored in the β(Q_k) term.

```
def Gradient_descent_Momentum(x_start):
    x_old=x_start
    error=np.linalg.norm(np.subtract(x_old,x_star))
    count_o=0
    alpha=0.03
    beta=0.8
    loss=[]
    q_k=0
    angle_list=[]
    while(error>0.01):
        p_k=np.subtract(np.matmul(Q,x_old),C)                                   # Calculation of the differential of F(x)
        #print("p_k ",p_k)
        if(count_o!=0):
            q_k=np.subtract(x_old,x_prev)                                       # Momentum which will pull the iterate in the direction taken previously
        x_new=np.subtract(x_old,(np.dot(alpha,p_k)))
        x_new=np.add(x_new,(np.dot(beta,q_k)))
        error=np.linalg.norm(np.subtract(x_new,x_star))                         # Error
        loss.append(error)
        #print("error ",error)
        #Gradient Angle
        transpose_val=np.subtract(x_new,x_old)
        transpose_val=transpose_val.transpose()
        n=np.matmul(transpose_val,np.subtract(x_star,x_old))
        d=(np.linalg.norm(np.subtract(x_new,x_old)))*(np.linalg.norm(np.subtract(x_star,x_old)))
        angle=n/d                                                               # Angle
        #print(angle)
        for i in angle:
            b=i
        for i in b:
            angle=i
        angle_list.append(angle)
        #Gradient Angle
        x_prev=x_old
        x_old=x_new
        count_o=count_o+1
```

Number of steps taken = 448 Error Curve –



Angle Plot –



The number of steps taken for 5 different values of Q and C with the same x_start. Q1, C1 = 714
Q2, C2 = 60
Q3, C3 = 2098
Q4, C4 = 1030
Q5, C5 = 622

As seen in the graph and the number of steps taken to reach convergence across 5 different values of Q, X and C, it is clear that using a factor of momentum helps in reaching the minimum quicker that gradient descent without momentum.

Higher the Beta value we plug in, more the influence of the momentum factor.

The angle plot shows that, it behaves similar to the constant alpha case where it tries to find the best angle towards the minimizer and tries to stay in that value.

Repeat the above, but for optimized momentum, using the optimal stepsizes α_k, β_k from before.

Optimal Alpha and Beta –

Other than the values of alpha/beta, the implementation of the gradient descent using an optimal value of alpha/beta remains the same.
The only difference is that instead of using alpha as 0.03 and beta as 0.8, the optimal values which were previously calculated are used.

Optimal Alpha –

$$\alpha_k = \frac{(q_k^T p_k)(p_k^T Q q_k) - (p_k^T p_k)(q_k^T Q q_k)}{(p_k^T Q q_k)(p_k^T Q q_k) - (p_k^T Q p_k)(q_k^T Q q_k)}$$

Implementation of finding the optimal alpha –

```
def calcAlpha(p_k,q_k):
    numerator_1=np.matmul(q_k.transpose(),p_k)
    numerator_1=np.matmul(numerator_1,p_k.transpose())
    numerator_1=np.matmul(numerator_1,Q)
    numerator_1=np.matmul(numerator_1,q_k)

    numerator_2=np.matmul(p_k.transpose(),p_k)
    numerator_2=np.matmul(numerator_2,q_k.transpose())
    numerator_2=np.matmul(numerator_2,Q)
    numerator_2=np.matmul(numerator_2,q_k)

    numerator=np.subtract(numerator_1,numerator_2)

    denominator_1=np.matmul(p_k.transpose(),Q)
    denominator_1=np.matmul(denominator_1,q_k)
    denominator_1=np.matmul(denominator_1,p_k.transpose())
    denominator_1=np.matmul(denominator_1,Q)
    denominator_1=np.matmul(denominator_1,q_k)

    denominator_2=np.matmul(p_k.transpose(),Q)
    denominator_2=np.matmul(denominator_2,p_k)
    denominator_2=np.matmul(denominator_2,q_k.transpose())
    denominator_2=np.matmul(denominator_2,Q)
    denominator_2=np.matmul(denominator_2,q_k)

    denominator=np.subtract(denominator_1,denominator_2)

    a=numerator/denominator
    for i in a:
        b=i
    for i in b:
        alpha=i
    return alpha
```

Optimal Beta –

$$\therefore \beta_k = \frac{(p_k^T p_k)(p_k^T Q q_k) - (q_k^T p_k)(p_k^T Q p_k)}{(q_k^T Q q_k)(p_k^T Q p_k) - (p_k^T Q q_k)(p_k^T Q q_k)}$$

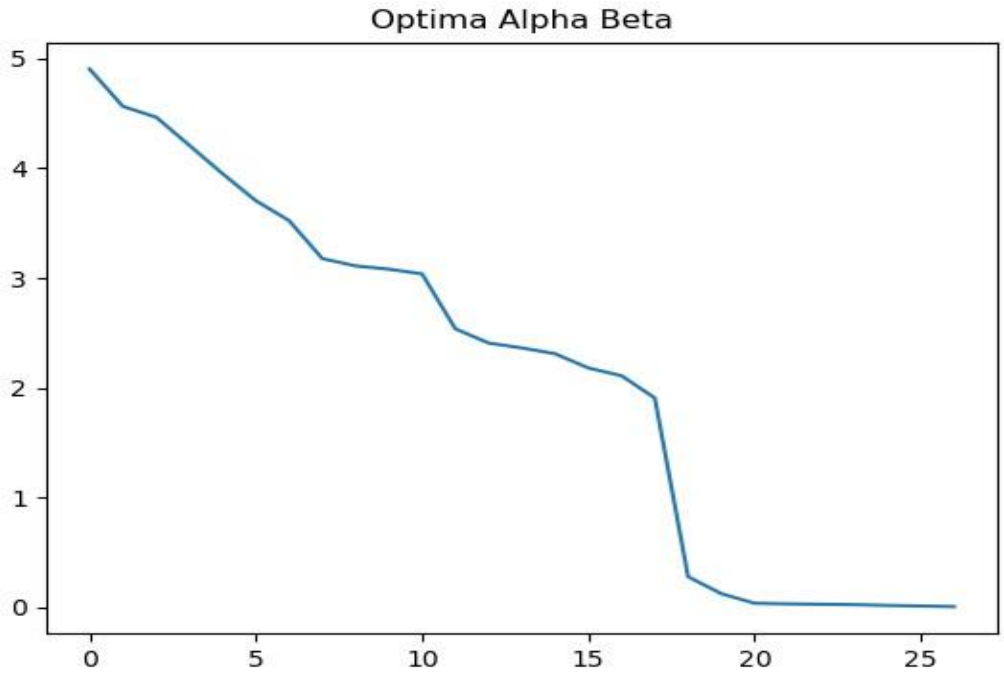
Implementation of finding the optimal beta–

```
def calcBeta(p_k,q_k):
    numerator_1=np.matmul(p_k.transpose(),p_k)
    numerator_1=np.matmul(numerator_1,p_k.transpose())
    numerator_1=np.matmul(numerator_1,Q)
    numerator_1=np.matmul(numerator_1,q_k)

    numerator_2=np.matmul(q_k.transpose(),p_k)
    numerator_2=np.matmul(numerator_2,p_k.transpose())
    numerator_2=np.matmul(numerator_2,Q)
    numerator_2=np.matmul(numerator_2,p_k)
    numerator=np.subtract(numerator_1,numerator_2)

    denominator_1=np.matmul(q_k.transpose(),Q)
    denominator_1=np.matmul(denominator_1,q_k)
    denominator_1=np.matmul(denominator_1,p_k.transpose())
    denominator_1=np.matmul(denominator_1,Q)
    denominator_1=np.matmul(denominator_1,p_k)

    denominator_2=np.matmul(p_k.transpose(),Q)
    denominator_2=np.matmul(denominator_2,q_k)
    denominator_2=np.matmul(denominator_2,p_k.transpose())
    denominator_2=np.matmul(denominator_2,Q)
    denominator_2=np.matmul(denominator_2,q_k)

    denominator=np.subtract(denominator_1,denominator_2)

    a=numerator/denominator
    for i in a:
        b=i
    for i in b:
        beta=i

    return beta
```

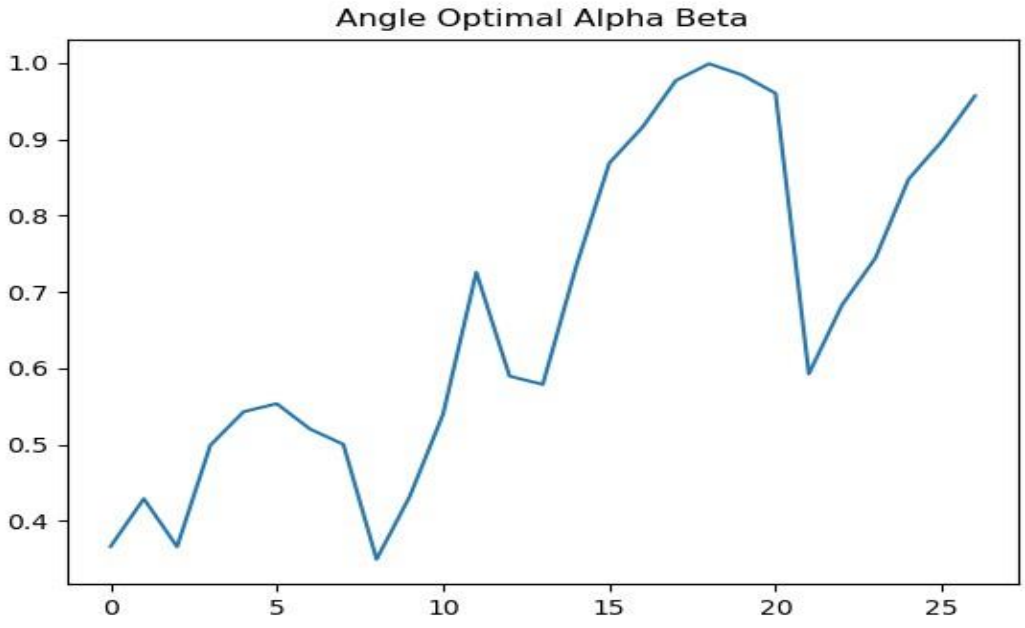
Implementation of gradient descent using the optimal values

```
def Gradient_descent_Momentum_alpha(x_start):
    x_old=x_start
    error=np.linalg.norm(np.subtract(x_old,x_star))
    count_o=0
    loss=[]
    angle_list=[]
    q_k=0
    while(error>0.01):
        p_k=np.subtract(np.matmul(Q,x_old),C)
        if(count_o!=0):
            q_k=np.subtract(x_old,x_prev)
            alpha=calcAlpha(p_k,q_k)
            beta=calcBeta(p_k,q_k)
        else:
            alpha=0.03
            beta=0
        x_new=np.subtract(x_old,(np.dot(alpha,p_k)))
        x_new=np.add(x_new,(np.dot(beta,q_k)))
        error=np.linalg.norm(np.subtract(x_new,x_star))
        loss.append(error)
        transpose_val=np.subtract(x_new,x_old)
        transpose_val=transpose_val.transpose()
        n=np.matmul(transpose_val,np.subtract(x_star,x_old))
        d=(np.linalg.norm(np.subtract(x_new,x_old)))*(np.linalg.norm(np.subtract(x_star,x_old)))
        angle=n/d
        for i in angle:
            b=i
        for i in b:
            angle=i
        angle_list.append(angle)
        x_prev=x_old
        x_old=x_new
        count_o=count_o+1
```

Number of steps taken = 27 Error Curve –



Angle Plot –



The number of steps taken for 5 different values of Q and C with the same x_start. Q1, C1 = 15
Q2, C2 = 16
Q3, C3 = 36
Q4, C4 = 29
Q5, C5 = 35

Using an optimal value of alpha and beta has had a huge impact on the number of steps the iterates take to reach the minimum.
As seen in the 5 test cases we have tested to compare it with the other gradient descents it is clear that the number of steps this method takes is far less in comparison.

Using a dynamic value of alpha and beta which changes according to the point in the function the iterate is in, helps in reducing the number of steps. It decides when importance should be given to the momentum factor and when it should be reduced.

The angle plot shows a huge variation compared to the other gradient descents as the factor of importance the momentum is given changes in every step. This leads to the angle changing constantly at every point in the iterates journey to convergence.

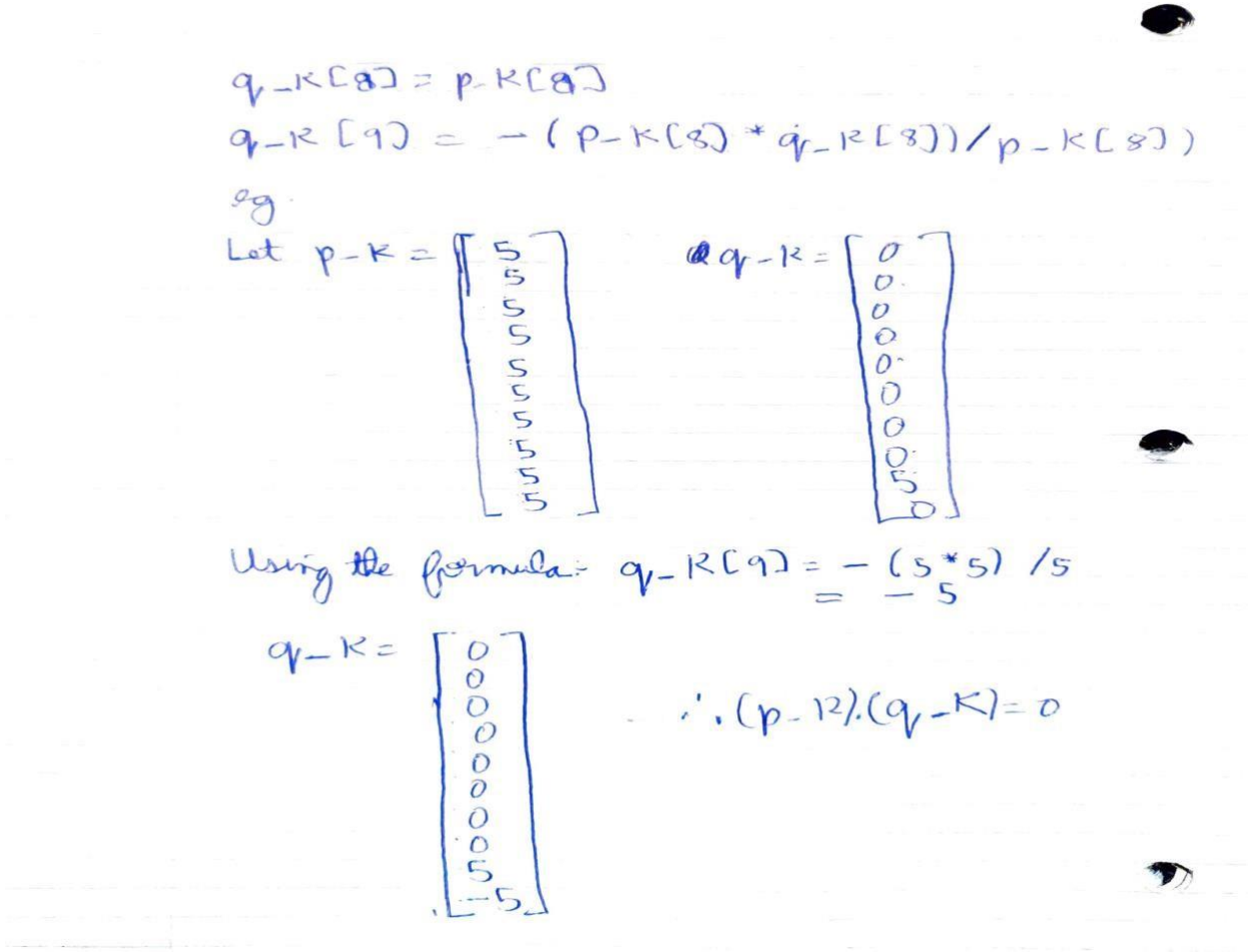
> What would moving directly towards the minimizer as possible 'look like', in terms of the iterates? How does this compare to the behavior of gradient descent and momentum methods as above?

Moving directly towards the minimizer would result in the angle plot consistently maintaining a value of 1. While descending using this method, the iterate is always pointed towards the minimum irrespective of the shape of the curve.

It might perform better than the other gradient descent algorithms only if the function itself is shaped in such a way that it points to the minimum.

> Given a vector $\underline{p}_k$, how can we generate an $\underline{q}_k$ that is orthogonal to $\underline{p}_k$?

A vector is orthogonal to another if and only if the dot product of both vectors is zero.
To generate q_k as an orthogonal vector to p_k, I have first generated a Dx1 vector q_k with all values being 0.
I then make the second last value of the q_k vector the same as the p_k vector and then calculate the last value of q_k in such a way that the dot product remains 0.



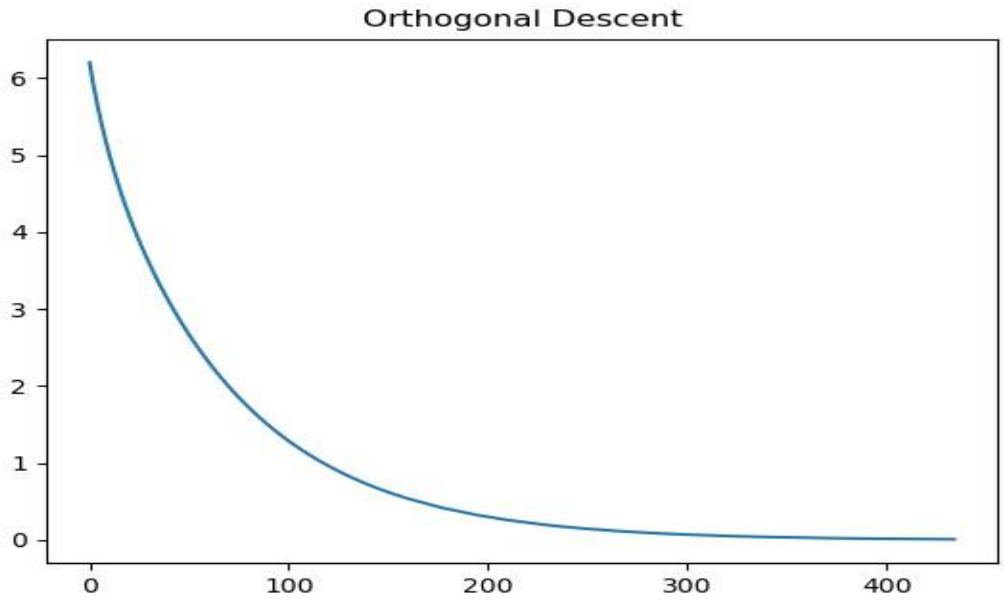
Python Implementation of finding orthogonal vector q_k –

```
def calcOrthogonal(p_k):
    q_k = np.array(np.random.choice([0], size=(D,1)),dtype = float)
    q_k[8][0]=p_k[8][0]
    q_k[9][0]= - ((p_k[8][0]*p_k[8][0]) / p_k[9][0])
    print("dot_prod ",np.dot(p_k.transpose(),q_k))
    return q_k
```

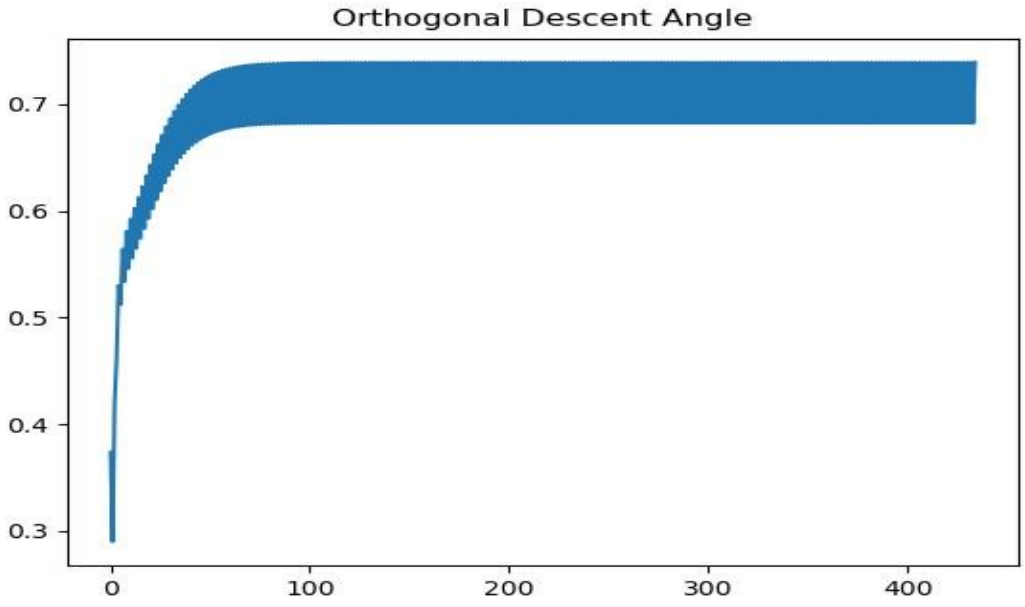
```
def Gradient_Descent_Orthogonal(x_start):
    x_old=x_start
    error=np.linalg.norm(np.subtract(x_old,x_star))
    count_o=0
    loss=[]
    angle_list=[]
    q_k=0
    while(error>0.01):
        p_k=np.subtract(np.matmul(Q,x_old),C)
        q_k=calcOrthogonal(p_k)
        alpha=calcAlpha(p_k,q_k)
        beta=calcBeta(p_k,q_k)
        x_new=np.subtract(x_old,(np.dot(alpha,p_k)))
        x_new=np.add(x_new,(np.dot(beta,q_k)))
        error=np.linalg.norm(np.subtract(x_new,x_star))
        loss.append(error)
        print("error ",error)
        transpose_val=np.subtract(x_new,x_old)
        transpose_val=transpose_val.transpose()
        n=np.matmul(transpose_val,np.subtract(x_star,x_old))
        d=(np.linalg.norm(np.subtract(x_new,x_old)))*(np.linalg.norm(np.subtract(x_star,x_old)))
        angle=n/d
        for i in angle:
            b=i
        for i in b:
            angle=i
        angle_list.append(angle)
        x_prev=x_old
        x_old=x_new
        count_o=count_o+1
    # plt.plot(loss, label="LOSS Random alpha beta ")
```

The gradient descent using this orthogonal vector as q_k instead of the momentum, converges to the minimum in less steps compared to vanilla gradient descent with a constant alpha.
The results of using an orthogonal vector as q_k is very close to the results of using vanilla gradient descent using an optimum alpha.

Number of steps taken = 400 Error Curve –



Angle Plot –



Overall Analysis of the gradient descents using x_start, [Q1,Q2,Q3,Q4,Q5] and [C1,C2,C3,C4,C5]

Constant alpha –
Q1, C1 = 3724
Q2, C2 = 458
Q3, C3 = 10615
Q4, C4 = 5312
Q5, C5 = 3256

Optimal Alpha –
Q1, C1 = 1767
Q2, C2 = 293
Q3, C3 = 4712
Q4, C4 = 1648
Q5, C5 = 1673

Constant Alpha/Beta –
Q1, C1 = 714
Q2, C2 = 60
Q3, C3 = 2098
Q4, C4 = 1030
Q5, C5 = 622

Optimal Alpha/Beta –
Q1, C1 = 15
Q2, C2 = 16
Q3, C3 = 36
Q4, C4 = 29
Q5, C5 = 35

Orthogonal Gradient Descent –
Q1, C1 = 1360
Q2, C2 = 277
Q3, C3 = 4210
Q4, C4 = 1618
Q5, C5 = 1568