

## CMO Assignment 2

**Biswadeep Debnath**

24528

BISWADEEPD@IISC.AC.IN

### QUESTION 1

#### 1. Solution

(a) Let's verify if  $\{d_k\}$  vectors are  $Q$ -conjugate by induction. We have,

$$d_{k+1} = p_{k+1} - \sum_{i=0}^k \frac{p_{k+1}^\top Q d_i}{d_i^\top Q d_i} d_i$$

**Case:  $k = 0$**

$$d_1 = p_1 - \frac{p_1^\top Q d_0}{d_0^\top Q d_0} d_0$$

Multiplying  $d_0^\top Q$  on both sides we get

$$d_0^\top Q d_1 = d_0^\top Q p_1 - \frac{p_1^\top Q d_0}{d_0^\top Q d_0} d_0^\top Q d_0$$

Simplifying,

$$d_0^\top Q d_1 = d_0^\top Q p_1 - p_1^\top Q d_0 = 0$$

since  $p_1^\top Q d_0$  is scalar

$$(p_1^\top Q d_0)^T = d_0^\top Q p_1$$

Hence,  $d_0$  and  $d_1$  are  $Q$ -conjugate.

**Case:  $k \geq 1$**

Let's assume that for some  $k \geq 1$  all  $\{d_0, d_1, \dots, d_k\}$  are mutually  $Q$ -conjugate

$$d_i^\top Q d_j = 0 \quad \forall i \neq j, \quad i, j \leq k$$

Now for  $d_{k+1}$

$$d_{k+1} = p_{k+1} - \sum_{i=0}^k \frac{p_{k+1}^\top Q d_i}{d_i^\top Q d_i} d_i$$

We need to show that  $d_j^\top Q d_{k+1} = 0$  for all  $j \leq k$

Multiplying by  $d_j^\top Q$  on both sides

$$d_j^\top Q d_{k+1} = d_j^\top Q p_{k+1} - \sum_{i=0}^k \frac{p_{k+1}^\top Q d_i}{d_i^\top Q d_i} d_j^\top Q d_i$$

Since  $d_j^\top Q d_i = 0$  for  $i \neq j$  and  $d_j^\top Q d_j$  for  $i = j$ . So, only the  $i = j$  term stays in the summation

$$d_j^\top Q d_{k+1} = d_j^\top Q p_{k+1} - \frac{p_{k+1}^\top Q d_j}{d_j^\top Q d_j} d_j^\top Q d_j$$

Simplifying,

$$d_j^\top Q d_{k+1} = d_j^\top Q p_{k+1} - p_{k+1}^\top Q d_j = 0$$

$\therefore d_j^\top Q d_{k+1} = 0$  for all  $j \leq k$ .

So, by induction all  $\{d_0, d_1, \dots, d_k, d_{k+1}\}$  are mutually  $Q$ -conjugate.

**(b)**

When  $Q = I$ , we have

$$d_{k+1} = p_{k+1} - \sum_{i=0}^k \frac{p_{k+1}^\top d_i}{\|d_i\|^2} d_i$$

This is the Gram-schmidt orthogonalization where  $\{d_k\}$  are orthogonalized versions of  $\{p_k\}$ . So  $\{d_k\}$ s are set of mutually orthogonal vectors that span the same subspace as  $\{p_k\}$ . (Strang (2006))

**(c)** Values of

$$(\alpha_k, -\nabla f(x_k)^T u_k, \lambda_k)$$

for first 7 iterations:

(-0.04793390144441663, -0.9673315291005838, 20.180529853641985)  
 (-0.14388242487974906, -2.9879422667129476, 20.766554839551436)  
 (-0.04442808714652065, -0.9532038402760677, 21.45498268094392)  
 (0.0427251256671488, 0.923255897977125, 21.609202631017975)  
 (-0.056177461943656665, -1.2616950218365033, 22.45909619594285)  
 (0.023672576073056895, 0.5974466836245178, 25.237924329853797)  
 (0.010208178997097202, 0.2694091463487003, 26.391499054367067)

## 2. Solution

Number of  $m$  directions computed = 14

## 3. Solution

Matrix  $M$ : A tolerance of  $10^{-16}$  is used to print the values in a concise manner, this can be adjusted in the code provided if required.



Figure 1 shows the residual norm  $\|r_k\|_2$  against the iteration number.

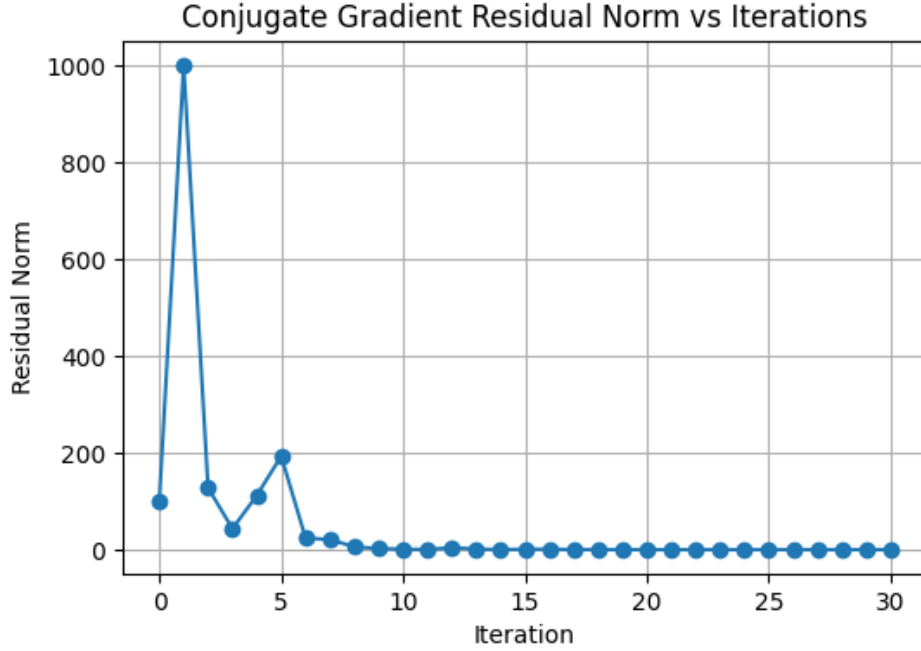


Figure 1: Residual Norm  $\|r_k\|_2$  vs Iteration Number

## 2. Solution

I used the *Preconditioned Conjugate Gradient* method from (Nocedal and Wright (2006)) to solve the system  $Ax = b$ . The idea is to make the gradient descent converge faster by using a *preconditioner*  $M$ , which is a matrix that approximates  $A$ .

The preconditioner  $M$  improves convergence by effectively solving the system with a better-conditioned matrix.

I used the *Jacobi preconditioner* (Häusner et al. (2024)), which approximates  $A$  as a diagonal matrix:

$$M = \text{diag}(a_{11}, a_{22}, \dots, a_{nn})$$

This makes it very easy to compute and invert, since it's just scaling each variable. The basic steps of the algorithm are:

---

**Algorithm 1** Preconditioned Conjugate Gradient (PCG) Method

---

```
1: Initialize  $r_0 \leftarrow Ax_0 - b$ 
2: Solve  $My_0 = r_0$ 
3: Set  $p_0 \leftarrow -y_0$ 
4:  $k \leftarrow 0$ 
5: while  $\|r_k\|/\|r_0\| > \text{tolerance}$  do
6:   Compute  $\alpha_k = \frac{r_k^T y_k}{p_k^T A p_k}$ 
7:   Update  $x_{k+1} = x_k + \alpha_k p_k$ 
8:   Update  $r_{k+1} = r_k + \alpha_k A p_k$ 
9:   Solve  $My_{k+1} = r_{k+1}$ 
10:  Compute  $\beta_{k+1} = \frac{r_{k+1}^T y_{k+1}}{r_k^T y_k}$ 
11:  Update  $p_{k+1} = -y_{k+1} + \beta_{k+1} p_k$ 
12:   $k \leftarrow k + 1$ 
13: end while
```

---

Figure 2 shows the residual norm  $\|r_k\|_2$  against the iteration number for CG\_SOLVE\_FAST.

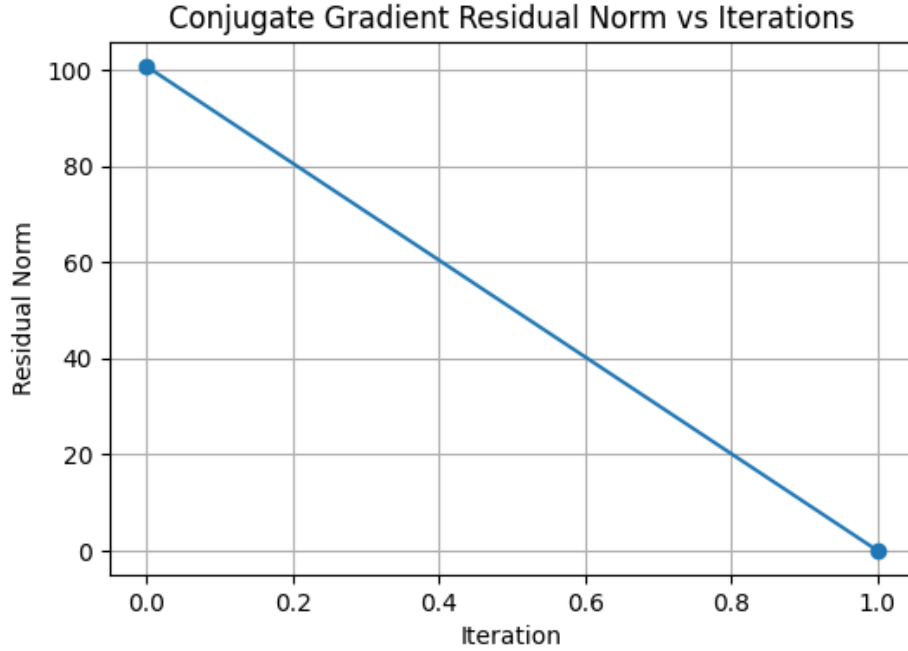
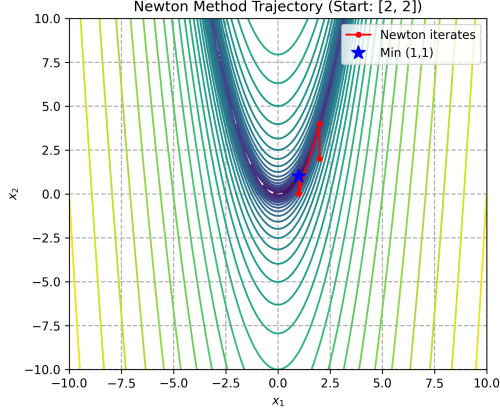


Figure 2: Residual Norm  $\|r_k\|_2$  vs Iteration Number for CG\_SOLVE\_FAST

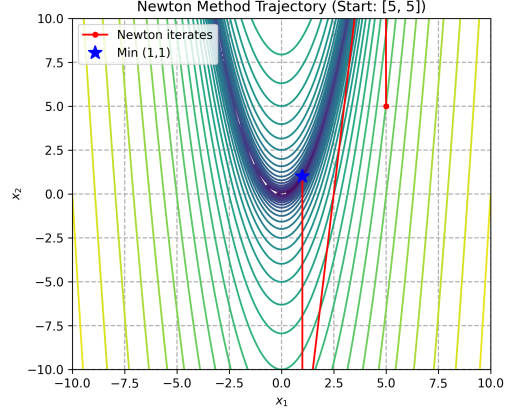
Surprisingly, the CG\_SOLVE\_FAST converged within one step which seemed unusual at first, implying that matrix  $M$  approximates  $A$  well. In other words,  $A$  is nearly a diagonal matrix and sparse elsewhere. To verify, tried a script to check the relative size of off-diagonal elements compared to diagonal ones across columns and it seemed that  $A$  is nearly diagonal.

## QUESTION 3

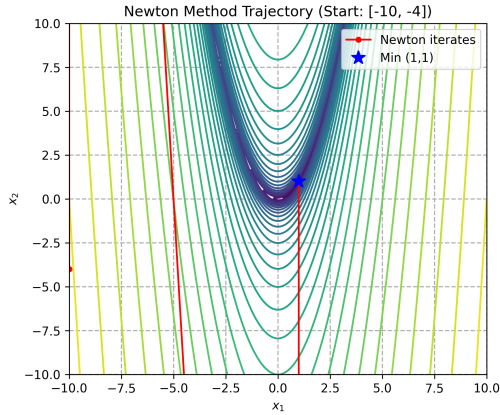
### 1. Solution



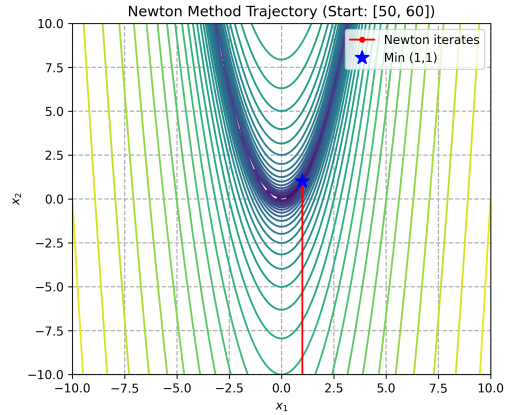
(a) Newton trajectory starting at [2, 2]



(b) Newton trajectory starting at [5, 5]



(c) Newton trajectory starting at [-10, -4]



(d) Newton trajectory starting at [50, 60]

Figure 3: Newton trajectory figures for different starting points

### 0.1 2. Solution

- All four points converge in this case which is evident from the plot in Figure 4. Starting from (50, 60), the error increases very high till 2<sup>nd</sup> iteration and then it converges in next iterations.
- In this 4 cases none of the points failed to converge, surprisingly all 4 of them took 5 iterations to converge.
- In Newton's Method, the starting point must be close enough to the solution for the method to work well. Specifically, if the initial residual size  $\|r_0\| < \frac{2L}{3M}$ , where  $L$  and

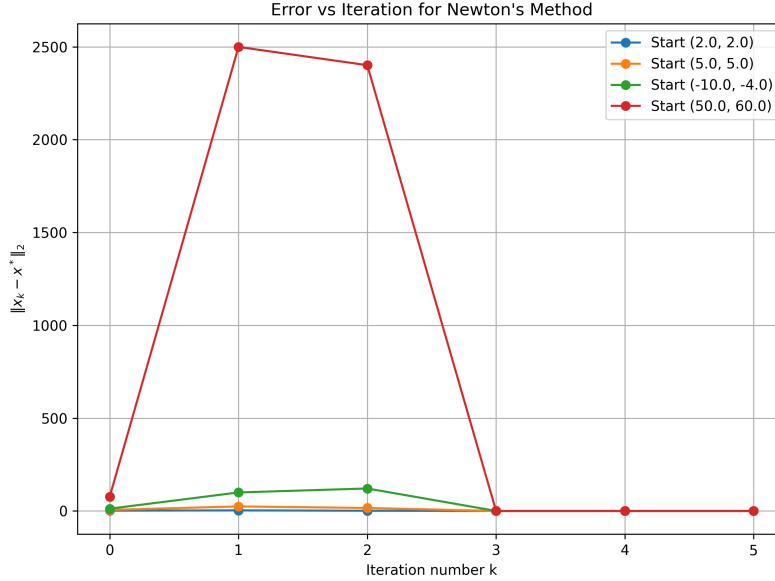


Figure 4: Newton method error vs iteration

$M$  are constants describing how fast the gradient and Hessian can change, then we can guarantee that the method will converge to the solution.

In this case, all four starting points converged exactly in 5 iterations. This suggests that these points are likely inside the region defined by the initial residual size restriction mentioned earlier.

Also the method was tested with points very far from the minimum, such as  $(10^{10}, 10^{20})$ , which resulted in an error: `numpy.linalg.LinAlgError: Singular matrix`.

The error log, which can be plotted using the provided code, shows that among the four points, those closer to the minimum take smaller, more cautious steps. And points farther away reduce the error faster, possibly due to the curvature of the function in those regions.

The starting point  $(2, 2)$  only has all its iterates inside the  $(-10, 10)$  region of the plot and rest other points moved outside of the  $(-10, 10)$  region.

## References

Paul Häusner, Ozan Öktem, and Jens Sjölund. Neural incomplete factorization: learning preconditioners for the conjugate gradient method, 2024. URL <https://arxiv.org/abs/2305.16368>.

Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2 edition, 2006.

Gilbert Strang. *Linear algebra and its applications*. Thomson, Brooks/Cole, Belmont, CA, 2006. ISBN 0030105676 9780030105678 0534422004 9780534422004. URL <http://www.amazon.com/Linear-Algebra-Its-Applications-Edition/dp/0030105676>.