1 First pass

At iteration k (crosscheck this),

$$P : \min_{\alpha \in \mathbb{R}} \phi(\alpha)$$
$$\text{where } \phi(\alpha) = x_k - \alpha \nabla f(x)|_{x=x_k}$$

and then

$$\hat{\alpha_k} = \underset{\alpha \in \mathbb{R}}{\operatorname{argmin}}\ \phi(\alpha)$$
$$x_{k+1} = x_k - \hat{\alpha_k} \nabla f(x)|_{x=x_k}$$

How to do line search?

1. Start with a bracket.
2. How? Go forward and backward.
3. Once we have $[a, b]$, we can do golden section or fibonacci section method.
4. We have I_1 from $[a, b]$. We have to pick an ε and then calculate n from it.
5. From n, calculate F_n. Write function to calculate p_j and q_j. Write function to select left interval or right interval.
6. Details in notes.

2 Second pass

Exact steps of forward and backward:

1. Start with α_0 and an h (baby steps, so h should be small value).
2. Go to $\alpha_0 + h$, see if $\phi(\alpha_0) > \phi(\alpha_0 + h)$.
3. If yes, go to $2h$, $4h$, $8h$, and keep checking same condition.
4. If no, then revert backwards using a different GP (or for simplicity use the same GP.)
5. As long as the function is decreasing you keep going forward.
6. Then as long as the function is decreasing you keep going forward.
7. You end up with a small bracket where there should be a minima.

Exact steps of fibonacci method:

1. $I_n = \dfrac{I_1}{F_n}$
2. $I_n < \varepsilon$
3. $I_k = I_{k+1} + I_{k+2} = (F_{n-k} + F_{n-k-1}) I_n = F_{n-k+1} I_n$

4. $I_{k+2} = I_k - I_{k+1}$
5. Either $x_p^k = x_u^k - I_{k+1}$ or $x_q^k = x_l^k + I_{k+1}$
6. Last mei $x_p^k = x_q^k$, then use a δ-disturbance.
7. For numerical reasons this can happen before, to δ wala ek iteration chalaya jayega.
8. Choose $\frac{\delta}{2} < \frac{I_1}{2F_n}$
9. See image for when to choose which interval.
10. Due to numerical issues, at some point x_p^k might be $> x_q^k$. In such case, choose x^* to be the mid point of x_l^k and x_u^k.

3 The third idea

Since we are doing quadratic optimization, we can find a closed form solution for α:

$\phi(\alpha) = x^k - \alpha \nabla f(x^k)$
and $f(x) = \frac{1}{2} x^T Q x - b^T x$
$\hat{\alpha_k}$ is the minimizer of $\phi(\alpha)$
setting $\phi'(\alpha) = 0$
$\nabla f(x) = Qx - b$

$$\phi'(\alpha) = \nabla f\left(x^k - \alpha \nabla f(x^k)\right)^T \nabla f(x^k)$$

let $g = \nabla f(x^k)$ and using x instead of x^k in the following for simpler notation
$\Rightarrow (Qx - \alpha Qg - b)^T (Qx - b) = 0$
$\Rightarrow \left(x^T Q^T - \alpha g^T Q^T - b^T\right)(Qx - b) = 0$
$\Rightarrow x^T Q^T Q x - x^T Q^T b - \alpha g^T Q^T Q x + \alpha g^T Q^T b - b^T Q x + \|b\|^2 = 0$

Note: Q is symmetric pd and $x^T Q^T b = b^T Q x$ (transpose of as scalar)
$\Rightarrow x^T Q^2 x - 2x^T Q b - \alpha g^T Q^2 x + \alpha g^T Q b + \|b\|^2 = 0$
$\Rightarrow \alpha = \frac{x^T Q^2 x - 2x^T Q b + \|b\|^2}{g^T Q^2 x - g^T Q b}$

given that the denominator is not zero (it is a scalar)
The denominator is zero only when either the gradient is zero or $Qx = b$
both of which only happen at the optimum point
(because $g^T Q(Qx - b) = 0$ only when either $g = 0$ or $Qx - b = 0$)

Note: x here is not a variable, but actually x^k (a fixed value)

4 Steepest Descent for Quadratic Case

I've written a code in `python` to generate a convex optimization problem at random and then solve it using *steepest descent* algorithm. It can be run using the following command:

```
$ python3 steepest_descent.py -n 100 -eps 1e-3 -seed 150
```

Parameters:

- `n`: the n in $\mathbb{R}^n \ni x$. This is optional, default value is 100. Type: integer.
- `eps`: the ε used for stopping criteria. Note: the stopping criteria is $\|\nabla f\| < \varepsilon$. This is also optional, if not provided the default value is taken to be 10^{-3}. Type: float.
- `seed`: all random number generators are *pseudo-random* sequence generators. Given a `seed` value, the random sequence is determined. This is good for repeatability. This is optional, if not set a new problem will be generated in each run. Type: integer.

I also wrote another code which is specialized for $n = 2$ and visualized the results in figure 1. The code is for a particular problem (value of Q and b) only, since I had to fine tune the level set plots. Plotting level sets at a particular value of the function in python is not convenient. Run the code as (to check repeatability, and also you can zoom in on the tiny parts of the figure using the magnifying glass tool):

```
$ python3 steepest_descent_plot.py
```

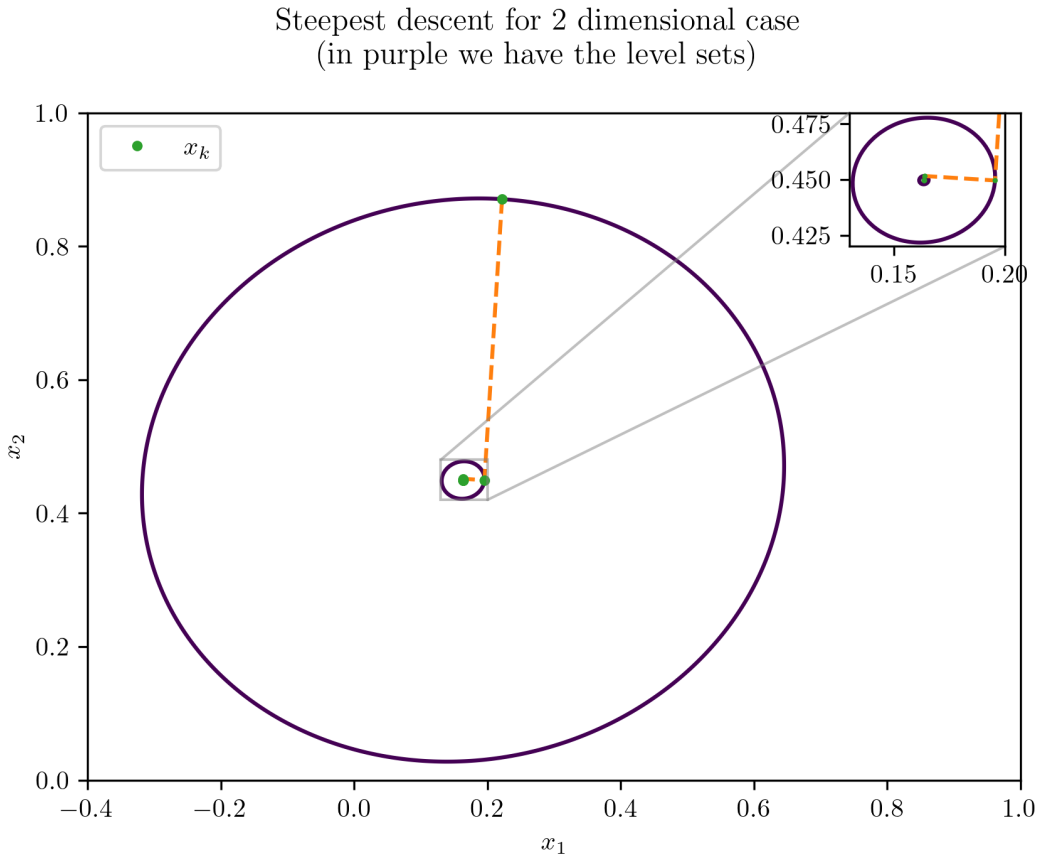

Figure 1: Movement of x^k towards the optimum value

I then ran this code for $n \in \{2, \ldots, 100\}$ and plotted the number of iterations it takes for the method to converge (figure 2). The number fluctuates a lot (because each problem is different), but it sometimes takes steepest descent more than 6000 steps to converge where Newton's method would have taken just 2.

I also plot the distribution of the number of iterations it takes steepest descent to converge for $n = 30$ from 500 runs of SD.

I then run the steepest descent algorithm for different n but keep the condition number fixed (and = 1000 which is very large), we see that ill conditioning results in increase of iterations (fig 4).

Naturally, I also ran the algorithm for varying values of n when the condition number is good. See figure 5. The number of iterations are less by a factor of 1000.

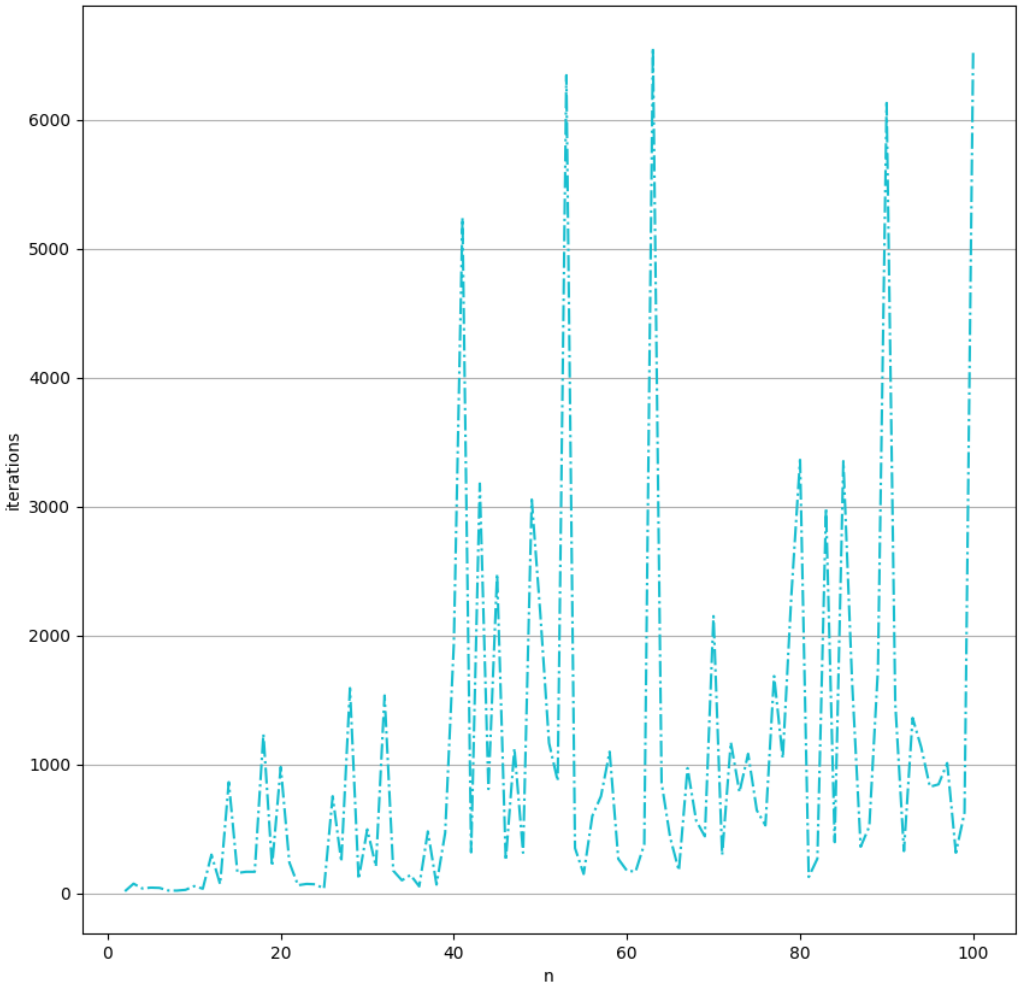

Figure 2: Number of iterations for convergence with varying values of n

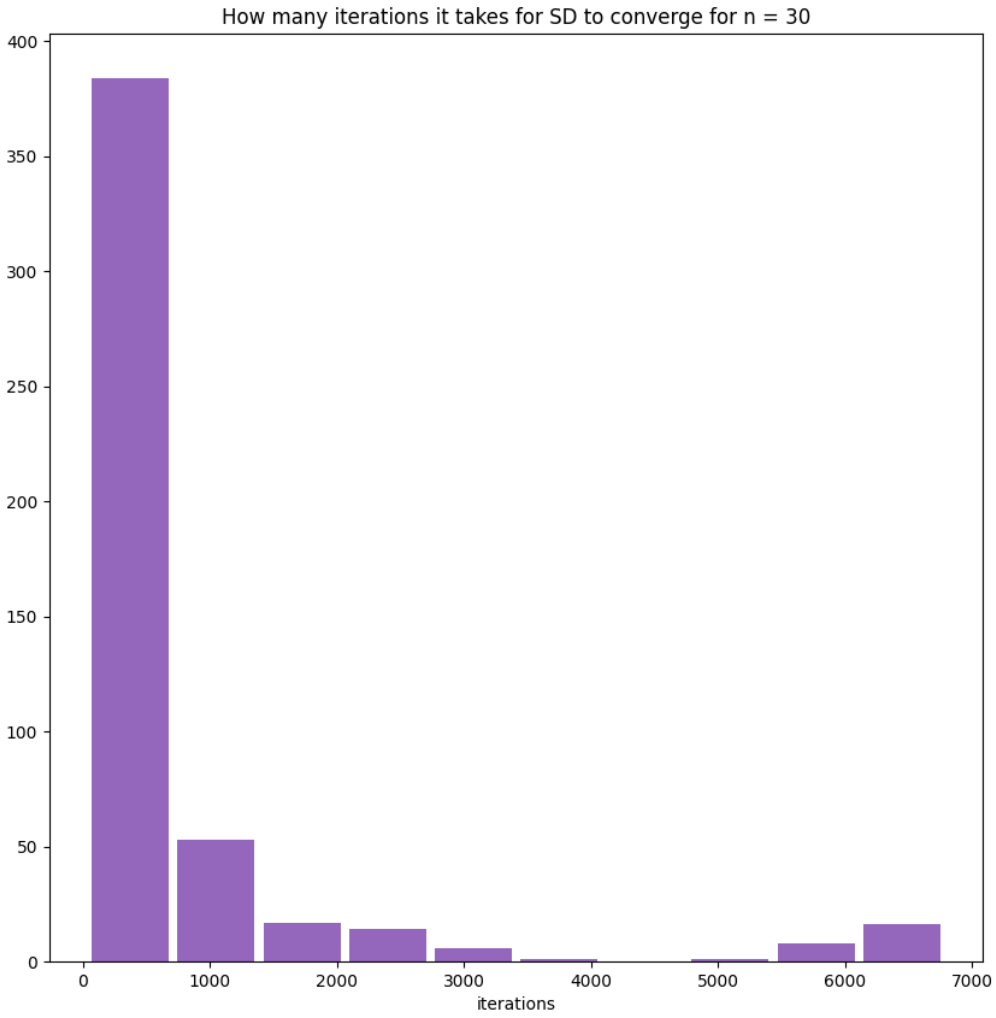

Figure 3: Number of iteration distribution for $n = 30$

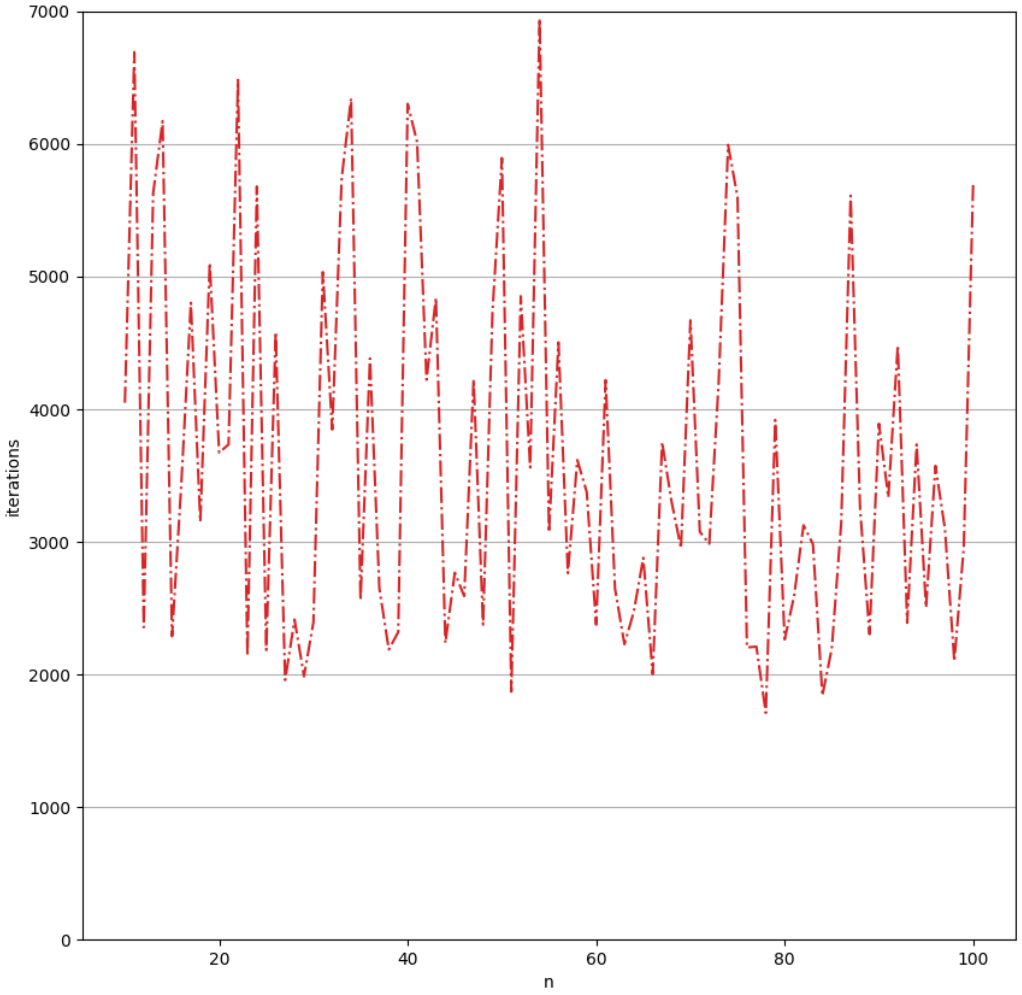

Figure 4: Number of iterations for convergence with varying values of n with a fixed value of condition number $r = 1000$

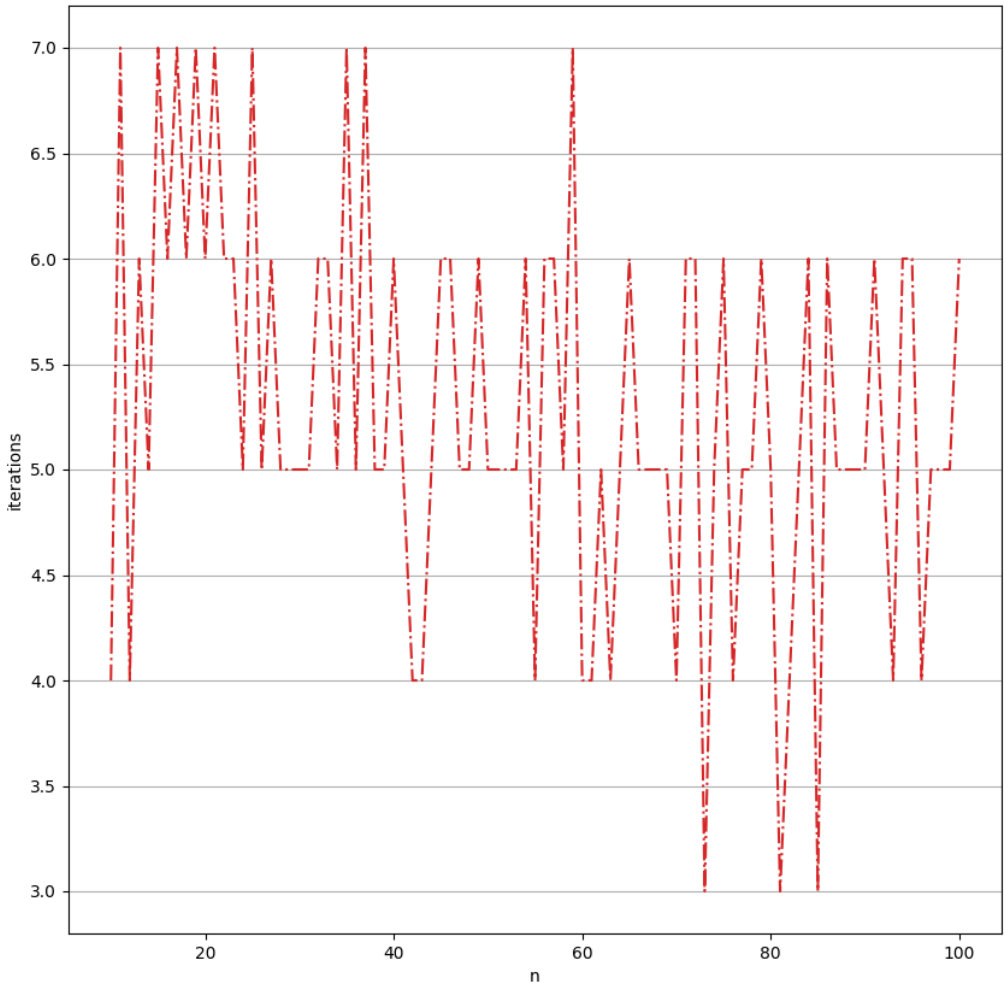

Figure 5: Number of iterations for convergence with varying values of n with a fixed value of condition number $r = 1.2$