AE588 HW3

3.1 and 3.2

Implemented both the **line search, backtracking and bracketing & pinpointing.** However, instead of passing $\emptyset(0)$ and $\emptyset'(0)$, $f(x_k + \alpha p_k)$ gradient was $\nabla f(x_k + \alpha p_k)$ was used as input, to reduce the function calls and $\emptyset(0)$) and $\emptyset'(0)$ were computed inside the functions.

For **search direction, steepest descent, conjugate gradient, and quasi-newton methods** were implemented.

- In steepest descent algorithm, for only the first iteration $\alpha_{\text{int}} = 1.2$, however for further iterations α_{int} was estimated using the following equation, it had a considerable effect on iterations, but function calls were greatly improved as the line search iteration was decreased extensively due to a good initial guess.

$$\alpha_k = \alpha_{k-1} \frac{\nabla f_{k-1}^{\intercal} p_{k-1}}{\nabla f_k^{\intercal} p_k}.$$

- In conjugate gradient, Fletcher-Reeves formula was used to obtained β_k. Thus, there was no need to reset the step direction to steepest direction in further iteration. However, the algorithm 4.6 suggest the α_{int} can be estimated as done in steepest descent. However, it performed considerably worse than by just setting $\alpha_{\text{int}} = 1.2$ throughout all iterations.

$$\beta_k = \frac{\nabla f_k^{\intercal} \nabla f_k}{\nabla f_{k-1}^{\intercal} \nabla f_{k-1}}.$$

- In quasi-newton method, approximated Hessian was reset when the directional derivative $\nabla f^T p$ is less than some threshold. In the book it says greater, however that lead optimization failing sometimes or worse overall performance. Through testing the most optimal threshold was found to be 1e-8, going lower than this had no effect on either iteration or function calls. But increasing the threshold value especially after 1e-7 had a detrimental effect on both iteration and function calls.

Algorithm 1 Quasi-Newton Hessian reset

```
while ||∇f||_∞ ≥ tolerance do
    compute search direction
    perform line search
    if |∇f^T p_k| ≤ 1e-8 then
        reset = TRUE
    else
        reset = FALSE
```

Thus, resulting in six different combination of optimization algorithm, namely steepest descent with backtracking, steepest descent with bracketing & pinpointing, conjugate gradient with backtracking, conjugate gradient with bracketing & pinpointing, quasi-newton with backtracking and quasi-newton with bracketing & pinpointing.

Each function returns the optimal x*, f*, convergence history and error history.

Additionally, in each optimization function (search direction), $f(x_k + \alpha p_k)$ and $\nabla f(x_k + \alpha p_k)$ was initialized outside the while loop, and the updated $f(x_k + \alpha p_k)$ and $\nabla f(x_k + \alpha p_k)$ was returned from the line search algorithm, as these values were already computed when performing the line search as shown in the following algorithm. Hence, the search direction function only has one function call throughout the optimization process reducing the total function calls. However, the line search algorithm must initialize $f(x_k + \alpha p_k)$ and $\nabla f(x_k + \alpha p_k)$ once before it begins its loop.

Algorithm 2 Optimization Algorithm Set-up

1: Initialize gradient
2: f, ∇f = func(x0)
3: ϕ, $\nabla\phi$ = f, ∇f
4: ($\nabla\phi(0)$ can computed by $\nabla\phi \bullet p_k$)
5: **while** $||\nabla f||_\infty \geq$ tolerance **do**
6: compute search direction
7: α^*, ϕ, $\nabla\phi$ = line search(α^*, p_k)
8: $||\nabla f||_\infty = ||\nabla\phi||_\infty$

This approach is especially exceptional for algorithm where the search direction holds information of a good guess of the initial step size for the line search. As most the time the line search will exit before entering the loop, thus reducing any unnecessary function calls. (Comparing it to how I did in HW2)

***To note: if the option was set to none the best overall performing algorithm was implemented.*

Both backtracking and bracketing & line search was implemented as it was obvious from HW2 that, bracketing and pinpoint in most cases will result in fewer iteration for optimization to converge. However, it was evident that the function calls will be less for backtracking as it didn't just need to Armijo condition. Thus, both were implemented to test this theory.

For the search direction, steepest descent and quasi newton are polar opposite in how they compute the direction, hence these were selected. Conjugated gradient performed in between, assumed from the book, thus it helped as benchmark to verify the performance of other search direction algorithm.

All six different algorithm passed the test given by *tests_uncon_optimizer.py*.

Only results for **bean function** are shown, as result for slanted quadratic function are shown in **3.3**. All parameters were kept the same and initial guess was also same throughout all runs.

Optimization Algorithm	x* Optimal	f* Optimal	Iterations
steepest descent with backtracking	1.21341151, 0.82412242	0.0919438164	36
steepest descent with bracketing & pinpointing	1.21341175, 0.82412272	0.0919438164	18
conjugate gradient with backtracking	1.21341199, 0.82412294	0.0919438164	38
conjugate gradient with bracketing & pinpointing	1.21341185, 0.82412288	0.0919438164	19
quasi-newton with backtracking	1.21341166, 0.82412262	0.0919438164	9
quasi-newton with bracketing & pinpointing	1.21341149, 0.82412254	0.0919438164	6

3.3

The parameters are the following and they were kept same for all the optimizations run:

$$\mu_1 = \text{1e-4}, \quad \mu_2 = 0.9, \quad \rho = 0.3, \quad \sigma = 2, \quad \text{reset point} = \text{1e-8} \quad \text{and} \quad x_0 = [100, 100]$$

Results for Slanted Quadratic Function

Optimization Algorithm	Iterations	Function Calls
steepest descent with backtracking	74	98
steepest descent with bracketing & pinpointing	5	14
conjugate gradient with backtracking	83	105
conjugate gradient with bracketing & pinpointing	6	15
quasi-newton with backtracking	2	4
quasi-newton with bracketing & pinpointing	2	8
SciPy Minimize – BFGS	3	16

It is evident that Quasi-newton method is the most efficient optimizer. However, surprisingly conjugate gradient method performed worse than steepest descent method. Additionally, for both conjugated gradient and steepest descent method, bracketing & pinpointing line search algorithm performed better than backtracking, resulting in a significant decrease in iterations and therefore reducing the function call. However, for quasi-newton method both line search algorithm had the same number of iteration but backtracking had half the number of function call. SciPy performed little bit worse than quasi but considerably better than other two search direction algorithms. But it had more function calls than all other optimization algorithm.

It can be concluded that **quasi-newton method is the best search direction algorithm**, but it is inconclusive which line search algorithm is better.

$$\mu_1 = \text{1e-4}, \quad \mu_2 = 0.9, \quad \rho = 0.3, \quad \sigma = 2, \quad \text{reset point} = \text{1e-8} \quad \text{and} \quad x_0 = [0, 0]$$

Results for 2D Rosen Brock Function

Optimization Algorithm	Iterations	Function Calls
steepest descent with backtracking	12700	12730
steepest descent with bracketing & pinpointing	10003	10459
conjugate gradient with backtracking	Max iteration	Max iteration
conjugate gradient with bracketing & pinpointing	Max iteration	Max iteration
quasi-newton with backtracking	22	34
quasi-newton with bracketing & pinpointing	22	63
SciPy Minimize – BFGS	19	68

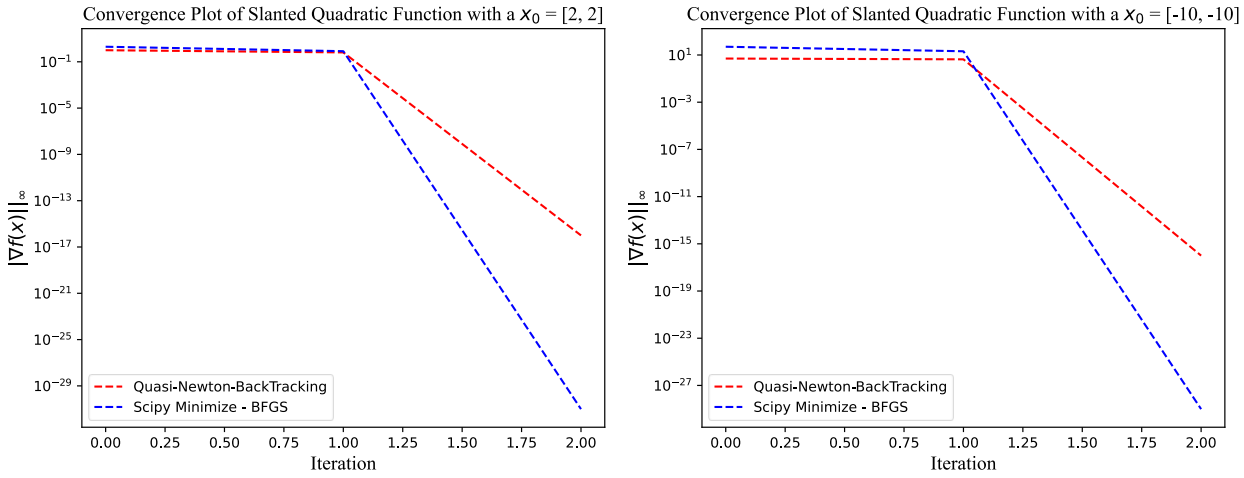
The conjugate gradient method performed the worse, failing to converge before 30000 iterations, this could be due to some incorrect implementation (it works on every other function expect Rosen Brock). Same conclusion can be made about steepest descent method coupled with bracketing & pinpointing performed better than the backtracking. However, it performed significantly worse than quasi-newton. Quasi-newton method performed the best, with both line search algorithms converging in same number of iterations. However, examining the function call it become evident that **quasi-newton coupled with backtracking is the best performing optimization algorithm.** SciPy performed moderately better than quasi-newton however it still had more function calls than both quasi-newton method. But this could be because SciPy is trying to converge to a tolerance of 1e-12.

3.4

The parameters are the following and they were kept same for all the optimizations run:

Slanted Quadratic Function:

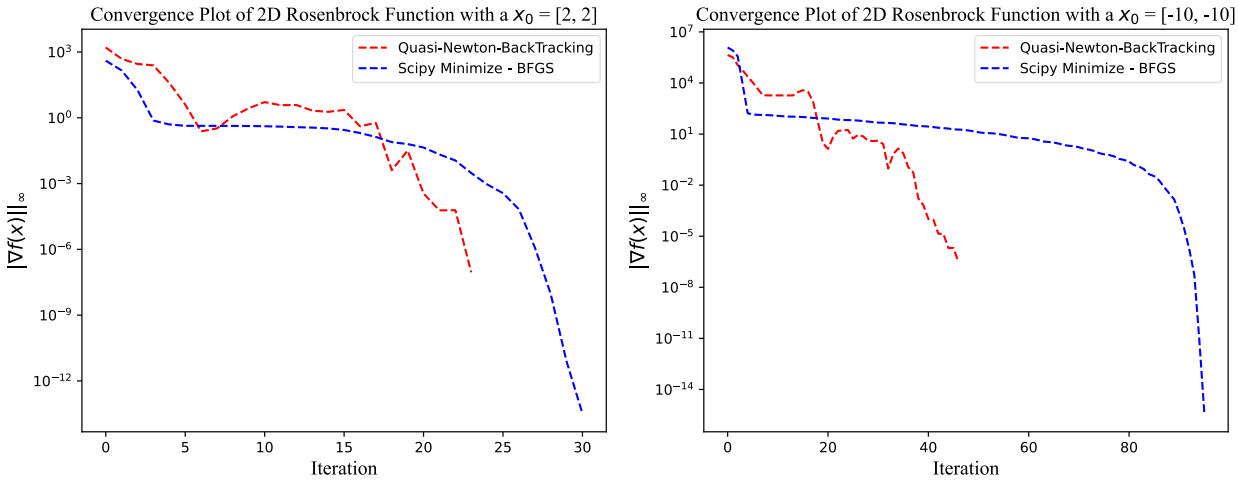
$$\mu_1 = \text{1e-4}, \quad \mu_2 = 0.9, \quad \rho = 0.3, \quad \sigma = 2, \quad \text{reset point} = \text{1e-8}$$



As the slanted quadratic function is positive definite hessian, both SciPy and my optimizer converged in 2 iterations.

2-D Rosenbrock Function:

$$\mu_1 = \text{1e-4}, \quad \mu_2 = 0.9, \quad \rho = 0.3, \quad \sigma = 2, \quad \text{reset point} = \text{1e-8}$$



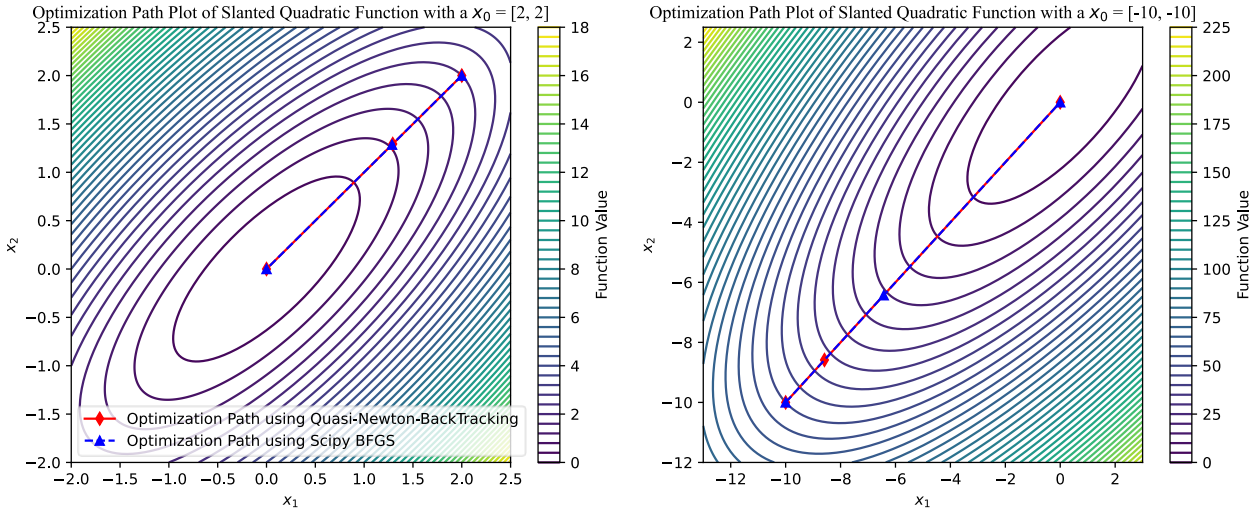
It is evident from the convergence plot that quasi-newton method behaves as a **super-linear convergence rate**, where at the end it begins to exhibit quadratic convergence depicted by sudden drop in error. For an initial guess closer to optimal, both optimizers performed similarly, but clearly Quasi-Newton with backtracking converged faster than SciPy for guess far away from optimum. As both the optimizer as based BFGS, both show same super-linear convergence.

3.5

The parameters are the following and they were kept same for all the optimizations run:

Slanted Quadratic Function:

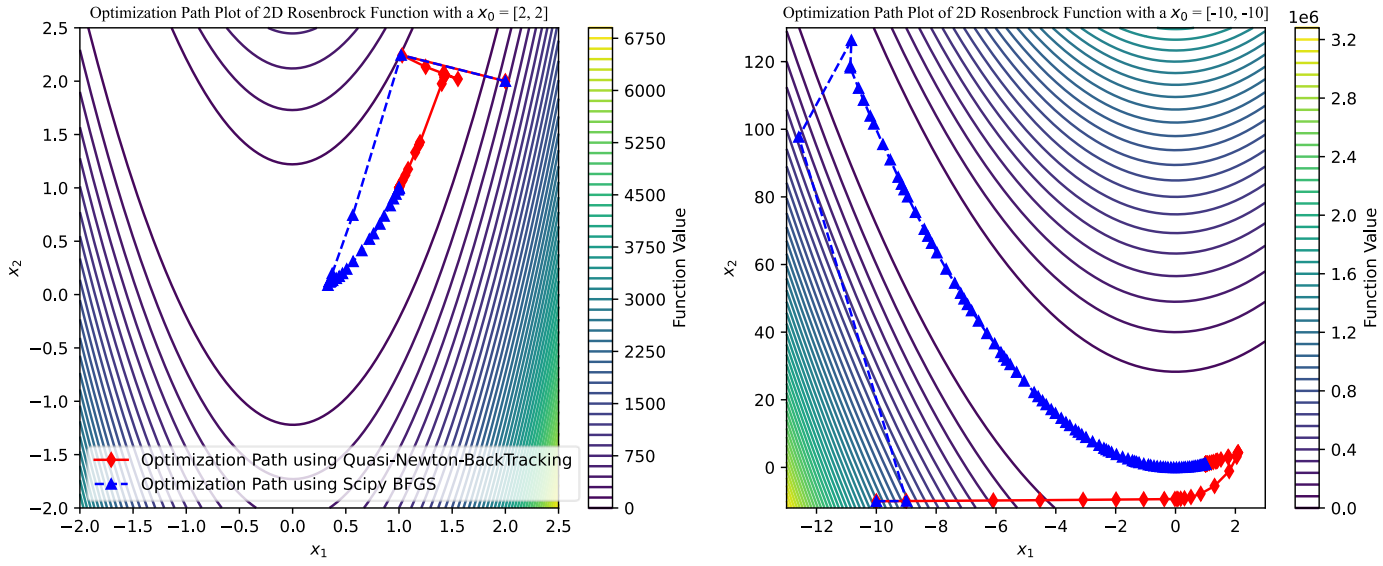
$$\mu_1 = \text{1e-4}, \quad \mu_2 = 0.9, \quad \rho = 0.3, \quad \sigma = 2, \quad \text{reset point} = \text{1e-8}$$



Both SciPy and Quasi-Newton method converges following the same path, in fact for the close point both of them step to same point. However, when moving far from minimum, quasi-newton takes a smaller step compared to SciPy, however, its next step is much larger.

2-D Rosenbrock Function:

$$\mu_1 = \text{1e-4}, \quad \mu_2 = 0.9, \quad \rho = 0.3, \quad \sigma = 2, \quad \text{reset point} = \text{1e-8}$$



For both the starting point, SciPy and Quasi-Newton self-made optimizer follow completely different convergence path. Even though their first step is similar, even though both methods are using BFGS for search direction, SciPy sometimes completely overshoots the minimum. This could explain a significant difference in iteration count. The quasi-newton method seems to go straight to the minimizer, but SciPy takes unnecessary path, explaining the higher iteration count.

3.6

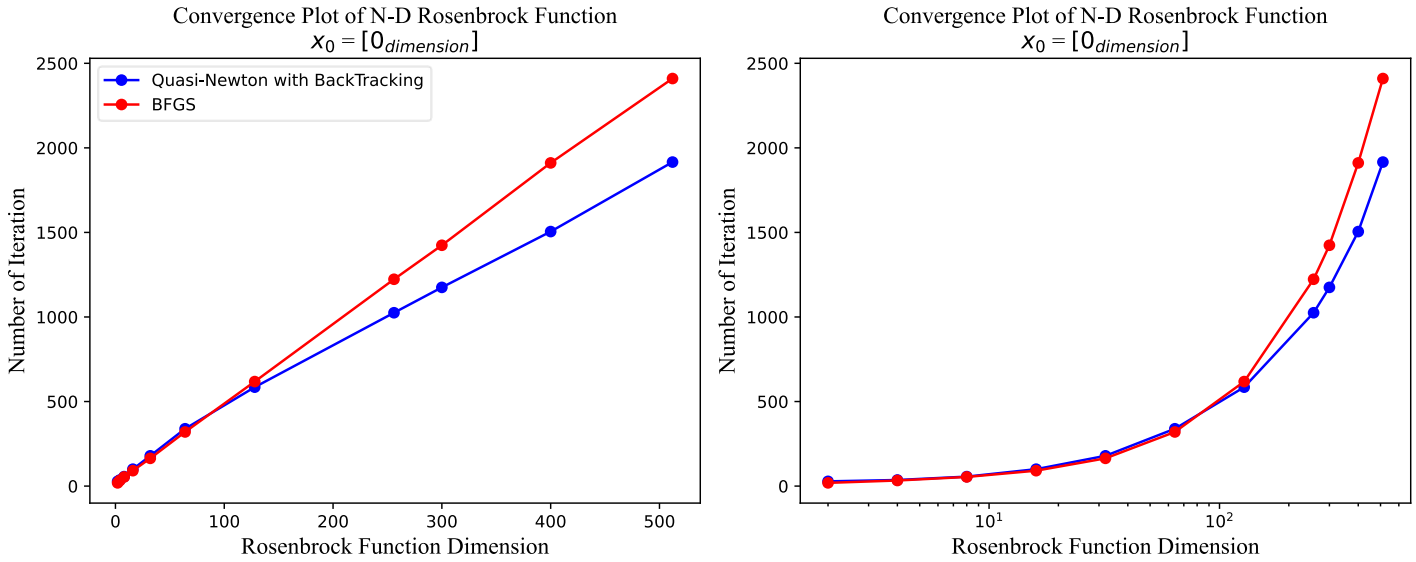
SciPy Setup:

```
1. result = minimize(lambda x: func(x)[0], x0, tol=1e-6, method='BFGS', jac=lambda x: func(x)
   [1])
```

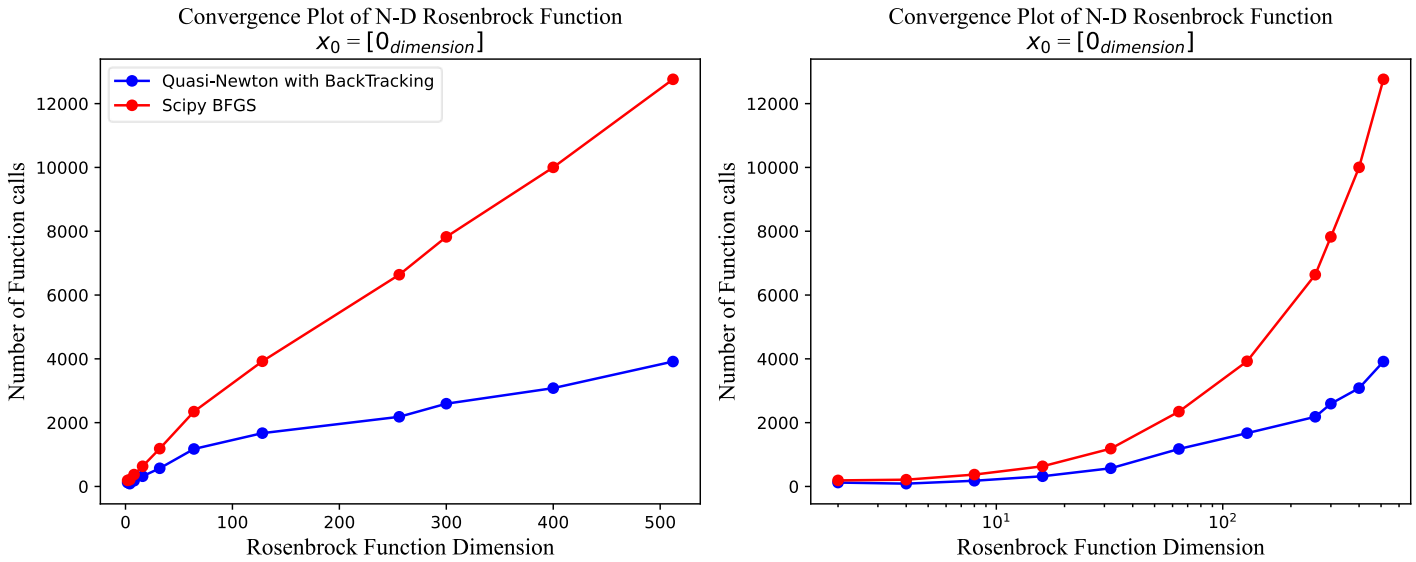
The parameters are the following and they were kept same for all the optimizations run:

N-D Rosenbrock Function:

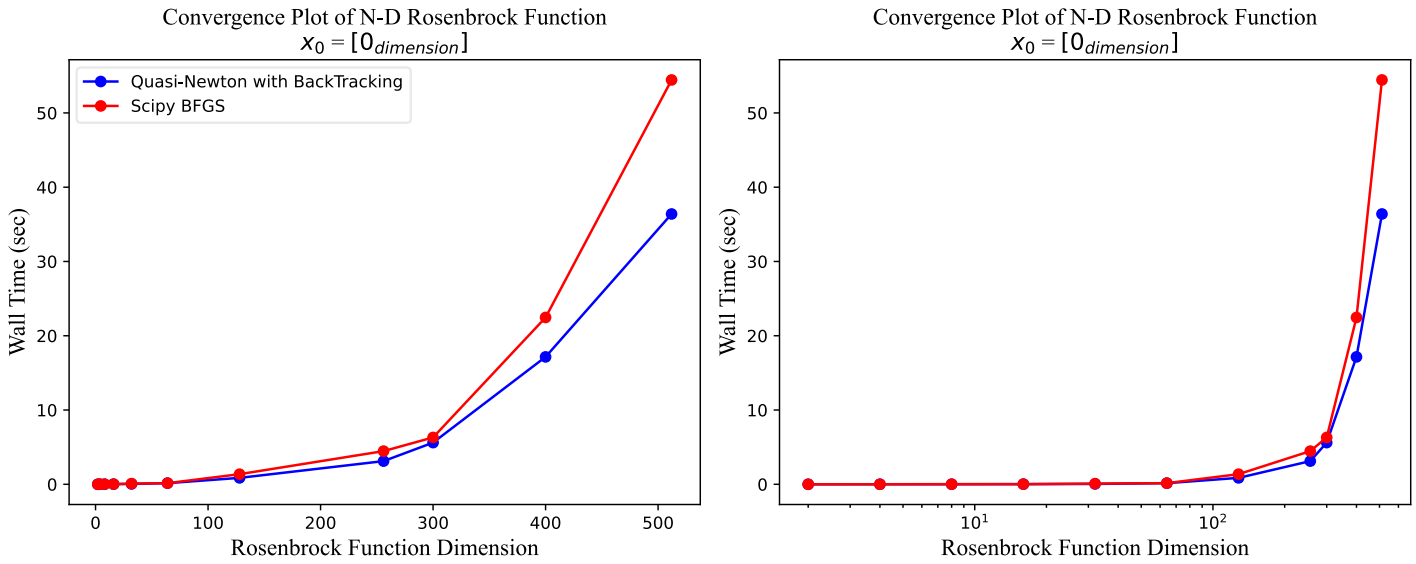
$$\mu_1 = \text{1e-4}, \quad \mu_2 = 0.9, \quad \rho = 0.3, \quad \sigma = 2, \quad \text{reset point} = \text{1e-8}$$



It is evident that as the dimension of the rosenbrock function increases there is linear increase in the number of iterations required to converge. This might suggest that computational cost or time will also scale linearly with the dimension. However, examining the function calls seems to contradict with that assumption.



The function calls also increase as the dimension of the rosenbrock function increases; however, it seems to follow an almost linear and almost an exponential curve. This suggests that computational cost or time will increases drastically as higher dimension problem are optimized.



Examining the wall-time plot, gives us a better idea about overall cost of the optimization in relation to increase in dimension of the function. There is a clear exponential trend observed, especially at around 300 or 10^2 dimension the time taken to converge increase drastically, before that there was barely any noticeable difference in wall time in grand scale.

However, these observations don't make the quasi-newton approach a bad optimizer, as if in a optimization problem, your main computational cost was to compute function or say a CFD solver, **then a linear relationship between dimensionality of the function is a desirable outcome.**

Comparing it with SciPy:

- As the dimension of the rosenbrock function increases the difference between number of iteration quasi newton that I made versus SciPy diverges. For higher dimension SciPy optimizer seems to perform worse.

- This problem is made worse when looking at the function call. There is dramatic difference between the optimizer, for 512D SciPy is almost 10000 more function calls than mine. Hence, I would assume computational time will be also greatly differed.

- However, the computational time until 300D is almost identical. But for 512D we can see a huge spike, leading me to believe SciPy will get far worse for higher dimensional.

- These results are very unexpected as I assume SciPy will be far superior at least for higher order. There is possibility I setup SciPy incorrectly, but I have double checked it by running it in a different environment and with different methods to setup SciPy, however each of the time it produces the same results.

3.7

- Implementation of quasi-newton method was difficult especially for backtracking as the computation of $s = x_k - x_{k-1}$. Because for some iteration x_k and x_{k-1} will be extremely close to each other especially when it is approaching the minimum making the s extremely small, resulting $\sigma = 1/s^T y$ to become *NA*. I suspect this could be due to optimizer sometimes not taking large enough step, and for quasi newton this problem most likely arise from computing the direction p_k as it holds the information for the initial guess line search. Adjusting the input parameters helped a lot to eliminate this problem, but in certain case this can still error can still occur.

- Another challenging aspect of the optimizer is tuning all the parameters, so it performs well for any function. However, there were no value for which any optimizer performed well for any function. As seen in **3.6** the computational cost dramatically increases at higher dimension and for 2D problems the difference was considerable, thus I made the choice to tune the parameters for 8-D Rosenbrock function as gave the performance best for most of the functions.

- Moreover, it became evidently clear that to decrease functions call further, a better line search algorithm is needed. As the way search direction algorithm has been setup (it only calls the function once throughout the problem) all the function call are happening inside the line search. I also believe the quasi-newton direction vector often times lead to conservative jump to next point, which also lead to increase in function call.

- I would like to explore a different line search algorithm, as this contributed to most function calls leading to increase in computational cost. Additionally, I would like to compute a partial hessian but with more information than what quasi-newton does, using auto differential packages.