AEROSP 588 Homework 3

Andi Zhou

October 11, 2023

Within this assignment, we implement a unconstrained gradient optimization algorithm. However, we originally chose the Quasi-Newton algorithm (BFGS). Due to some bugs however, we have to switch to **steepest descent** in the end. We will still include the general algorithm for both the steepest descent and BFGS, and speculate on the possible bugs that we might have within the code.

1 Algorithm Implementation

Question 1 involves implementing the solvers and run them through the auto grader to pass the bean function test and the slanted quadratic function. This was done and passed, using the method of steepest descent. The BFGS method worked for the bean function test case, but could not pass the slanted quadratic function. We will include the pseudo-code for both algorithms in the next section.

2 Algorithm Discussion

In this section, we first present the algorithm and its pseudo-code, and we later discuss their differences and why we originally picked the BFGS

2.1 Steepest Descent

Perhaps the simplest algorithm to code. The steepest descent algorithm relies on taking the gradient at each step locations. The idea is to move in the direction of the steepest decrease of the function, which is given by the negative of the gradient.

Algorithm 1 Steepest Descent Pseudocode

```
function SD($x_{\text{init}}, \tau$)
    k = 0
    Initialize various other storage vector needed
    while $||\nabla f||_{>}\tau$ do $p_k = -\frac{\nabla f_k}{||\nabla f_k||}$
        Estimate $\alpha_{\text{init}}$
        $\alpha_k$ = linesearch($p_k, \alpha_{\text{init}}$)
        $x_{k+1} = x_k + \alpha_{kk}$
        $k = k + 1$
    end while
    return $x^*$, $f(x^*)$
end function
```

Where to estimate the α_{init} we use the following formula:

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

However, the steepest descent is prone to a zig-zag behavior if the function curvature varies significantly with direction. We could solve this by either introducing the conjugate gradient method, or second order method that uses the curvature information themselves: Newton and Quasi-Newton.

2.2 Quasi-Newton Algorithm

Newton's method is an optimization technique that aims to find where a function reaches its minimum or maximum value. To do this, it doesn't just consider the slope (or steepness) of the function at the current point (as is done in the steepest descent), but also takes into account the curvature information, Hessian. This extra piece of information allows Newton's method to not only find the appropriate descent direction but also guess its distance as well. However, Newton's method involves a number of calculations involving solving linear systems with respect to Hessian matrix, which could be computationally expensive.

Instead of calculating the curvature exactly, Quasi-Newton methods try to approximate the matrix iteratively as the solver converges. This makes them faster and more efficient than the standard Newton's method while still benefiting from the extra information the curvature provides. The Quasi-Newton Method we pick here is called the BFGS method.

Algorithm 2 Quasi-Newton Pseudocode

```
function BFGS($x_{\text{init}}, \tau$)
    k = 0
    Initialize various other storage vector needed
    while $||\nabla f||_{>}\tau$ do
        if $k = 0$ or reset = true then
            $\tilde{V}_k = \frac{1}{||\nabla f||} I$
        else
            $s = x_k - x_{k-1}$
            $\nabla f_k - \nabla f_{k-1}$
            $\sigma = \frac{1}{s^T y}$
            $\tilde{V}_k = (I - \sigma s y^T)\tilde{V}_{k-1}(I - \sigma s y^T) + \sigma s s^T$
        end if
        $p = -\tilde{V}_k \nabla f_k$
        $\alpha_k = \texttt{linesearch}(p_k, \alpha_{\text{init}})$
        $x_{k+1} = x_k + \alpha_{kk}$
        $k = k + 1$
    end while
    return $x^*$, $f(x^*)$
end function
```

The magical equation is here:

$$\tilde{V}_k = (I - \sigma s y^T)\tilde{V}_{k-1}(I - \sigma s y^T) + \sigma s s^T \tag{2}$$

where this is an approximated inverse Hessian that we update each iteration, and we can replace the linear system required by Newton's method with a much less demanding and memory taking product:

$$p = -\tilde{V}_k \nabla f_k \tag{3}$$

where we could obtain the direction. Note that since the Quasi-Newton accounts for curvature, we no longer take the unit vector of the direction p_k. Instead, we take the full direction by assuming an $\alpha_{\text{init}} = 1$. This completes our algorithm implementation.

The BFGS algorithm was chosen initially for its ability to take in curvature information and potential superlinear convergence speed. Figure 1 illustrates the difference in convergence speed between the Quasi-Newton BFGS method and the steepest descent method. As expected, the steepest descent exhibited linear convergence speed while the BFGS exhibited a much faster, superlinear convergence due to the additional curvature information the algorithm incorporates. The BFGS is not `SciPy`, but rather my own codes. However, as mentioned in the opening, some issues within the algorithm prevented it from passing the Slant Quadratic test, but we could still use it to analyze the bean function.

Note the difference between the optimization path as well. The signature zig-zag behavior of the steepest descent and the BFGS algorihm that approaches the minimum more directly. Although it may seem like the SD algorithm approached the minimum faster, BFGS convergence accelerated drastically as it approached the minimum, while SD's convergence rate is still linear. This super-linear convergence behavior based on the incoporation of second order information is the exact reaason why we picked BFGS initially.

Again, our BFGS algorithm only worked on the bean function, which is why we could compare it here. On the slanted quadratic function, the algorithm exhibited a weird bug that I could not fix. Therefore, for the rest of this report, we are going to continue using the algorithm of Steepest Descent. From this comparison, however, it is easy to see that BFGS would have definitely been a superior alternative if it was coded correctly.

3 Slanted Quadratic and 2D Rosenbrock Function

We run the Steepest Descent algorithm on both the Slanted Quadratic (Equation 4):

$$f(x_1, x_2) = x_1^2 + x_2^2 - \beta x_1 x_2 \tag{4}$$

and 2D Rosenbrock (Equation 5)

$$f(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2 \tag{5}$$

3.1 Function Evaluation and Iteration

For the first task, we look at the final predicted optimum location and the optimum value by the steepest descent and the off-the-shelf optimzer `SciPy`, and we obtain the following value:

Evaluation for the Slanted Quadratic Function starting at $x_0 = \big(-1, 2\big)$

	x^*	$f(x^*)$	N
SD	$\big[7.1 \times 10^{-7}, 5.8 \times 10^{-7}\big]$	2.2×10^{-13}	45
BFGS (`SciPy`)	$\big[-1.07 \times 10^{-9}, -1.14 \times 10^{-9}\big]$	6.18×10^{-19}	7

Evaluation for the 2D Rosenbrock starting at $x_0 = \big(-3, 1\big)$

	x^*	$f(x^*)$	N
SD	$[0.9999992, 0.999998]$	5.71×10^{-13}	15168
BFGS (`SciPy`)	$[0.9999950.999989]$	2.84×10^{-11}	19

For both the Slanted Quadratic Function and the 2D Rosenbrock, we observe complete convergene for both functions. However, the BFGS exhibited a much faster convergence rate for the `SciPy` compared to Steepest Descent. For example, the Slanted Quadratic function took the Steepest Descent 45 iterations, while it only took `SciPy` 7 iterations to complete. Perhaps a more drastic comparison exists at the 2D Rosenbrock, due to the narrow valley of Rosenbrock and a far away starting point that is completely out of the valley, it took Steepest Descent a total of $N = 15168$ iterations to reach the optimal point, but it only took `SciPy` $N = 19$ iterations. This is the super-linear convergence that we talked about in the last section. Since BFGS has access to a higher order of information, it can better predict the behavior of the function and a reasonable initial step as a result.

3.2 Convergence Plot

Figure 2 illustrates the convergence plot for both Steepest Descent and BFGS (`SciPy`) for Slanted Quadratic and Rosenbrock Function respectively. We again observe the linear convergence rate for Steepest Descent and the super-linear rate for BFGS. The convergence criterion is assessed via the optimality condition, where $||y||_{<} 1 \times 10^{-}6$

3.3 Optimization Path

We also obtain the algorithm's optimization path (Figure 3). Again, note that Steepest Descent took longer time to converge due to the zig-zag motion due to Rosenbrock's (and Slanted Quadratic's) extreme change in curvature as a function of position.

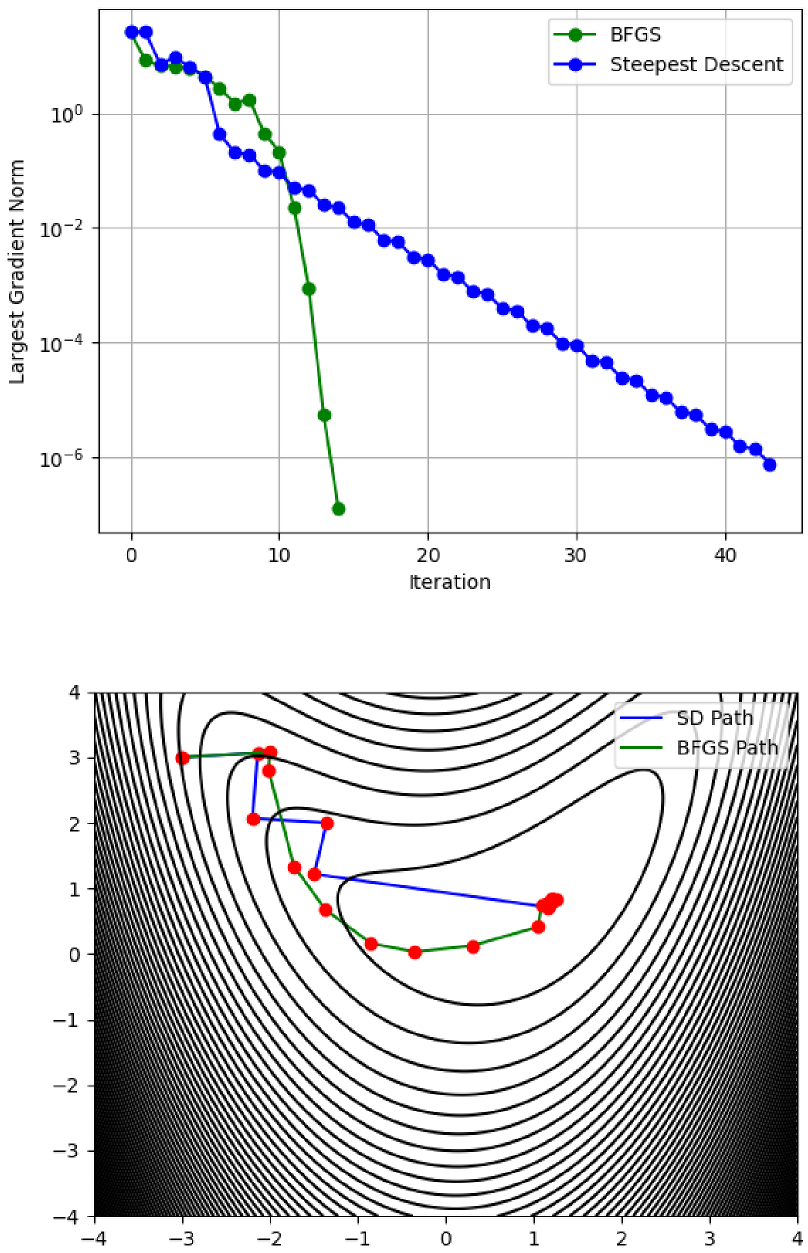


Figure 1: [TOP]Convergence Comparison between BFGS and Steepest Descent, and [BOTTOM] Optimization path of BFGS and Steepest Descent on the Bean function

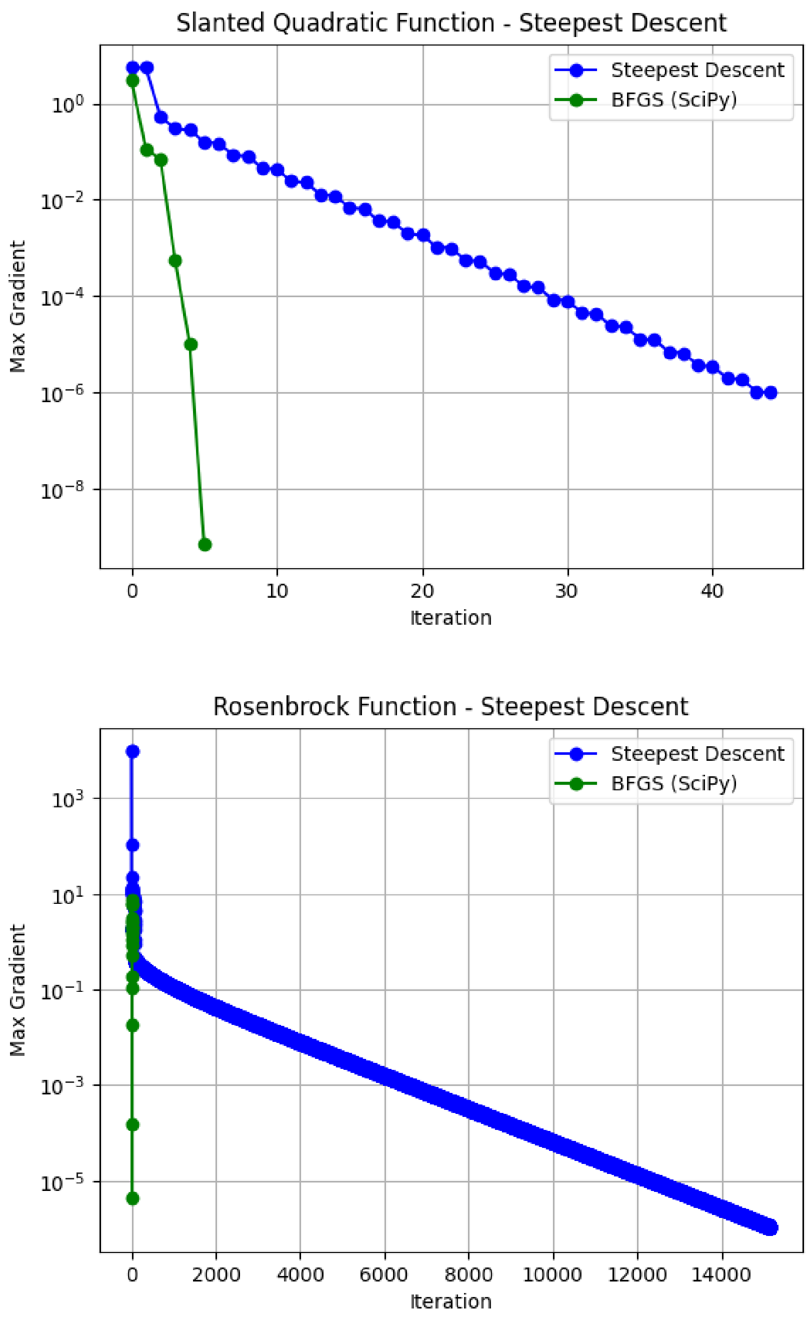
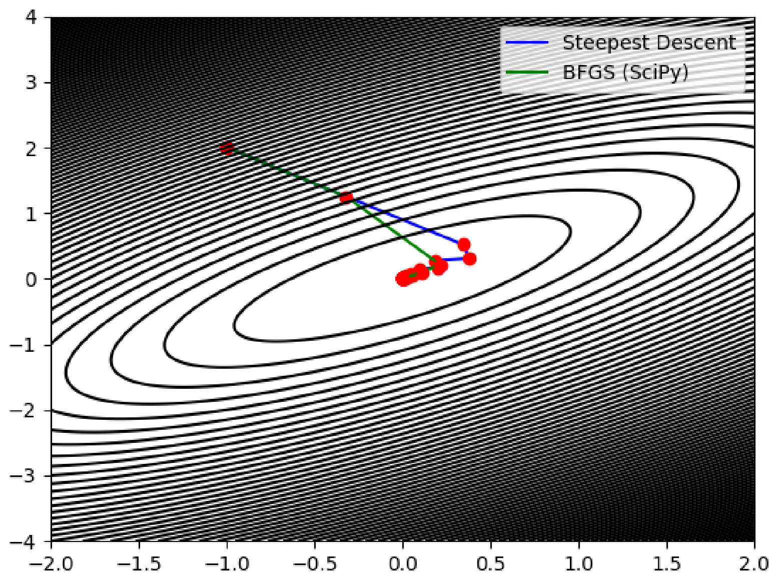


Figure 2: Convergence Plot for Slanted Quadratic and Rosenbrock Function



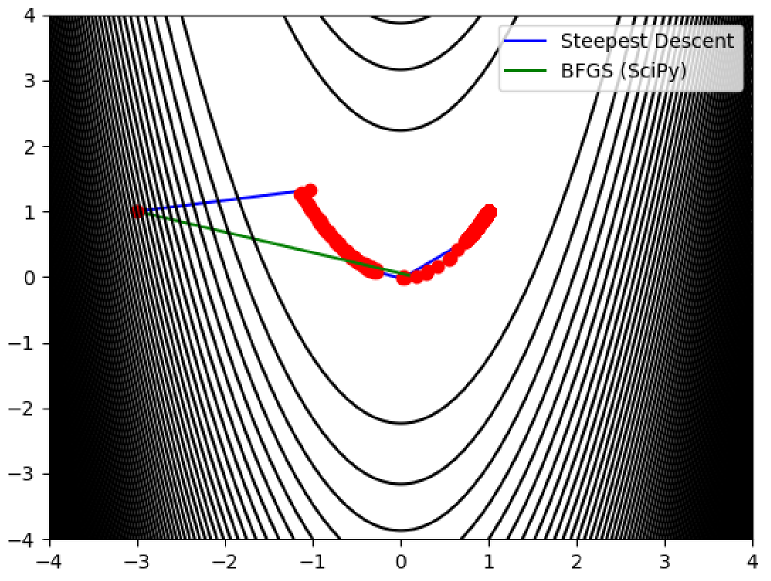


Figure 3: Optimization path for both the slanted quadratic function and the Rosenbrock function respectively.

4 N-Dimension Rosenbrock

As the last part of this assignment, we investigate into the N-dimension Rosenbrock. The equation for N-Dimensional Rosenbrock is given as:

$$f(x) = \sum_{i=1}^{n-1} \left(100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right) \tag{6}$$

while its gradient, upon differentiating with respect to x_i, we obtain the following:

$$\frac{\partial f(x_i)}{\partial x_i} \begin{cases} -400x_1(x_2 - x_1^2) - 2(1 - x_1) & x_i = 1 \\ 200(x_i - x_{i-1}^2) - 400x_i(x_{i+1} - x_j^2) - 2(1 - x_i) & 2 \leq i \leq n-1 \\ 200(x_n - x_{n-1}^2) & j = n \end{cases} \tag{7}$$

We now have all the necessary information to compute the optimal point within the N-Dimensional Rosenbrock. From analytical answers, we know that the global minimum of Rosenbrock always exists at:

$$\begin{aligned} x^* &= [1, 1, 1, 1...1] \\ f^* &= 0.0 \end{aligned} \tag{8}$$

Therefore, we vary the dimension from $N = 2, 4, 8...64$, as higher dimensions are unreasonable (method wise and computationally wise) to compute for Steepest Descent. The quantity we are interested in this analysis in computational expense, which we will monitor using the number of **iterations**. Figure 4 presents the Dimensions vs. Iterations result. For the `SciPy` BFGS algorithm, we observe a gradual increase in iterations as dimension of the problem increases. This trend is true regardless of different starting point.

For steepest descent, however, We observe a rather non-proportional relationship between dimensions on the computational cost. For $N = 2$, we see a high starting iteration for $x_0 = [0, 0, 0...., 0]$ and $x_0 = [-4, -4, -4, ..., -4]$, but a lower iteration for $x_0 = [2, 2, 2...., 2]$. This makes sense as $x_0 = [0, 0]$ is already a tricky starting point for 2D Rosenbrock due to the low gradient in the area. For $x_0 = -4$, the point is far away from the valley and therefore also requires multiple steps to converge.

As dimension increase however, from 2 to 8 and 16, we observe a decrease in computational cost. When dimension increase to 32 and 64, the cost begins to increase steadily. The behavior observed suggests that in around $N = 4 - 16$ dimensions, the steepest descent path might be less obstructed by the narrow valley effect seen in 2D. Further increase in dimensions likely adds complexity to the optimization landscape. As dimension increases, the volume of the search space increases exponentially with each dimension added. This makes it increasingly harder for optimization algorithms to find the minimum, hence the steadily increasing computational cost. This could be illustrated rather clearly in Figure 5, where we see sort of a "search phase" in the function when the maximum absolute gradient $||y||_\infty$ is constant for the first 8000 iterations.

5 Lesson Learned

The main challenge faced during this project is the Line Search algorithm. It appears that my bracketing and pinpointing algorithms are wrong as their answers did not match with the textbook answers at 4.3 exactly. Therefore, majority of the effort on this project was spent on debugging the line search algorithm.

Another challenge is debugging the Direction and Line Search algorithm. This is an area of active confusion as my BFGS algorithm is still bugged. However, it is rather difficult to differentiate whether a bug came from the line search algorithm or the search direction algorithm. Since now that we know the line search algorithm works since the method of Steepest Descent has been working reliably, we could trace back the bug down to the BFGS update.

Main lesson learned in this project include coding styles and debugging methods. The previous bracketing line search algorithm was not extremely modular. Therefore, a chunk of effort within this homework was devoted in rewriting the code from HW2. Upon completion, however, the new line search algorithm could be readily placed in different functions without having to provide too

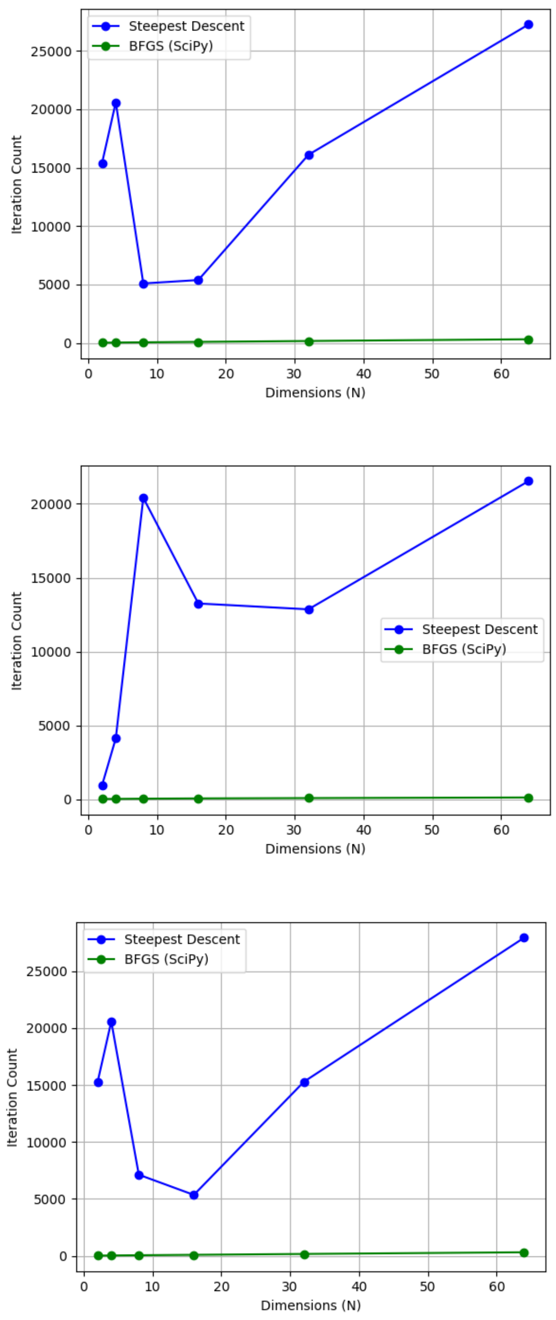
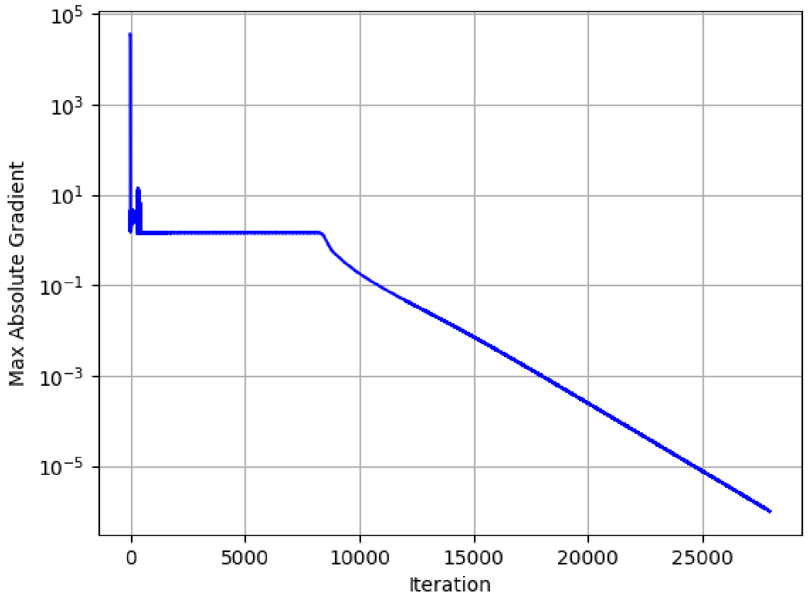


Figure 4: Dimension vs. Iteration with respect to different starting points. TOP: $x_0 = [0, 0, 0..., 0]$. MIDDLE: $x_0 = [2, 2, 2..., 2]$. BOTTOM: $x_0 = [-4, -4, -4..., -4]$



[H]

Figure 5: Convergence Plot for $N = 64$

much input. The most reliable debugging method that I have learned in this project is plotting. Since a lot of the problem is 2D, we have the luxury of visualizing the search direction and line search algorithm in action. I used this method to find an issue that existed with my gradients.

For the future, it would definitely be beneficial for me if I could fix the BFGS algorithm as it is evident from this write-up that SciPy BFGS is many, many orders of magnitude faster than the steepest descent method. Much of my effort within the next two weeks will have to be placed on fixing the BFGS code. It is suspected that the current code has some linear algebra issues when calculating the inverse Hessian.