AE588 HW6

Q1:

I implemented Nelder–Mead algorithm.

- First, point of the simplex were computed using Eq 7.2 and 7.1, which is then stored in X, where each element contains a full set of design variables. The first element is the initial guess and rest are computed using Eq 7.1 and 7.2. I created a simplex function which outputs the design space of n variable X and its corresponding function value.

$$x^{(i)} = x^{(0)} + s^{(i)}$$

$$s_j^{(i)} = \begin{cases} \frac{l}{n\sqrt{2}}\left(\sqrt{n+1}-1\right) + \frac{l}{\sqrt{2}}, & \text{if } j = i \\ \frac{l}{n\sqrt{2}}\left(\sqrt{n+1}-1\right), & \text{if } j \neq i\,. \end{cases}$$

- For convergence criteria I implemented both the size of simplex and standard deviation of function value.
- To order the list of simplex points in respect to its function value from low to high, I implemented a function “sort”. It takes in X and F (vector containing function value, so $X[0] = x^{(0)} \;\; => \;\; F[0] = f^{(0)}$). It then uses these vectors to form a dictionary where the key is the function value and values are design variable. Then using sort module in dictionary, the dictionary is sorted and sorted X and F vector are extracted from the dictionary, which is then returned. However, it is possible that two key values are same (happened in Rosenbrock higher dimension), then it will be overwritten, and we will lost a set of design variable. Thus, to prevent this “sort” function was implemented in the following way:

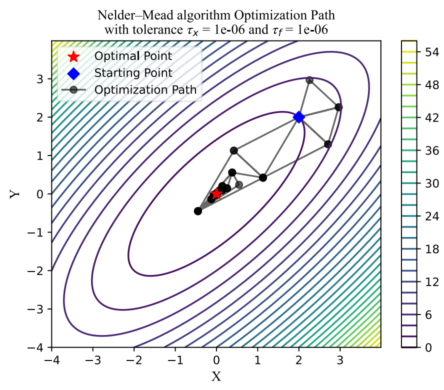
```python
def sort(X_vec, F_vec):
    # Create a list of tuples (function value, corresponding point)
    simplex_tuples = list(zip(F_vec, X_vec))
    # Sort the list of tuples based on function values
    simplex_tuples.sort(key=lambda x: x[0])
    # Recreate X_vec and F_vec using the sorted list of tuples
    X_vec = np.array([point for _, point in simplex_tuples])
    F_vec = np.array([value for value, _ in simplex_tuples])
    return simplex_tuples, X_vec, F_vec
```

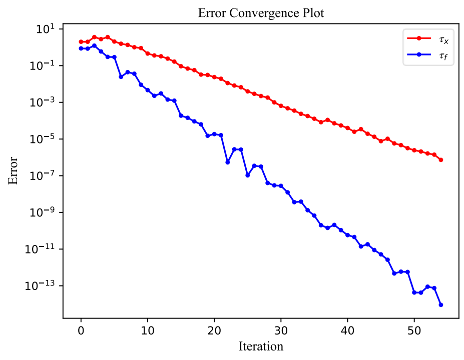
- Now, we compute the centroid x_c excluding the worst point. Then we generate a new point through equation:

$$x = x_c + \alpha\left(x_c - x^{(n)}\right)$$

- Then, implemented the logic given in algorithm 7.1. Moreover, at the end of each iteration error needs to be recalculated for new X design space.

To test the implementation, it was applied on quadratic slanted function with starting guess [2, 2]

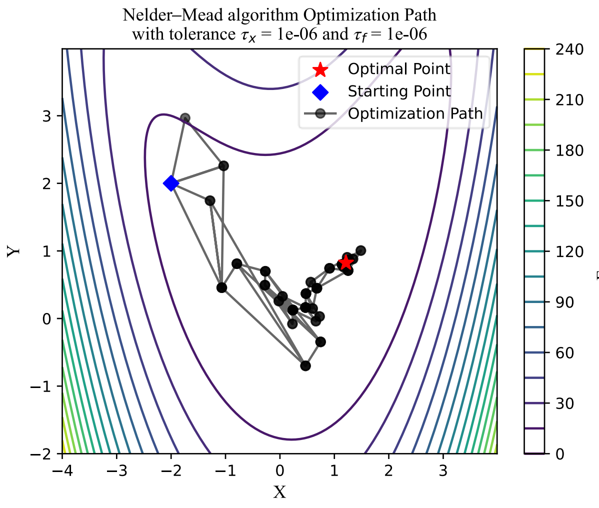


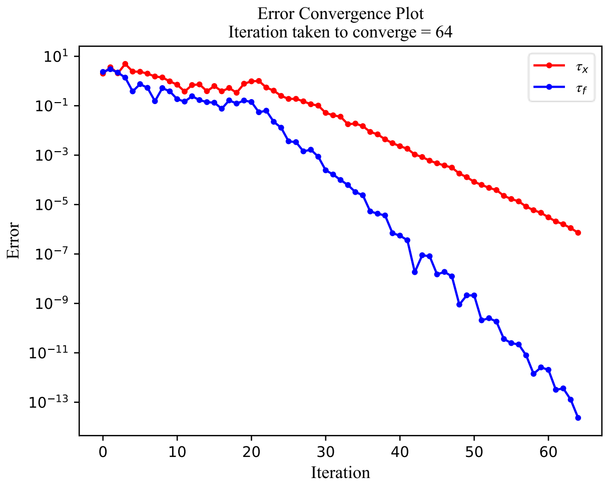


Q2.

****Throughout the question, analytical derivatives were provided for SciPy-BFGS.**

Implementing Nelder–Mead on bean function with starting point [-2, -2]:





Method	x^*		$f(x^*)$	Iteration
Nedler-Mead	1.21341156	0.82412267	0.09194381641128956	64
SciPy - BFGS	1.21341166	0.82412262	0.09194381641122203	14

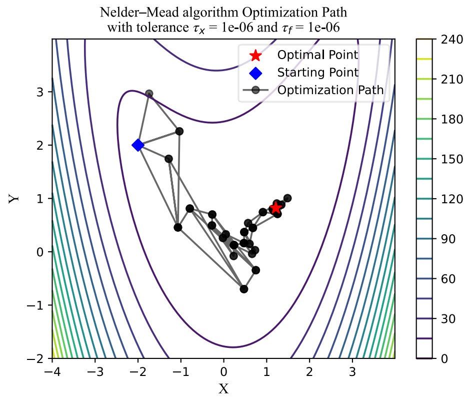
As, it is evident both gradient-free and gradient based optimization method converges to the same point.

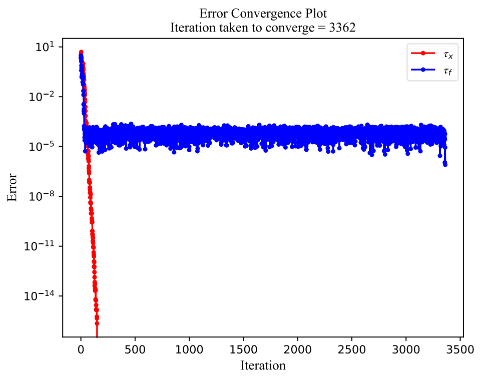
Introducing Noise to bean function:

```
# Generate random noise using a Gaussian distribution
noise = np.random.normal(loc=0, scale=magnitude)
```

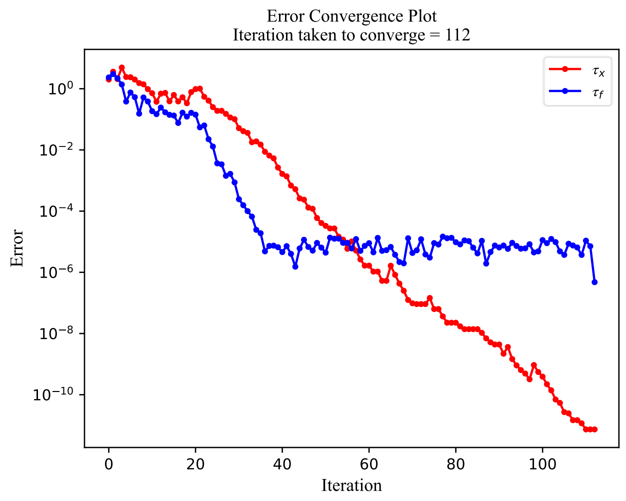
Where the magnitude was **10^{-2}, 10^{-3}, 10^{-4}** and **10^{-5}**. This noise was added to bean function and its derivatives.

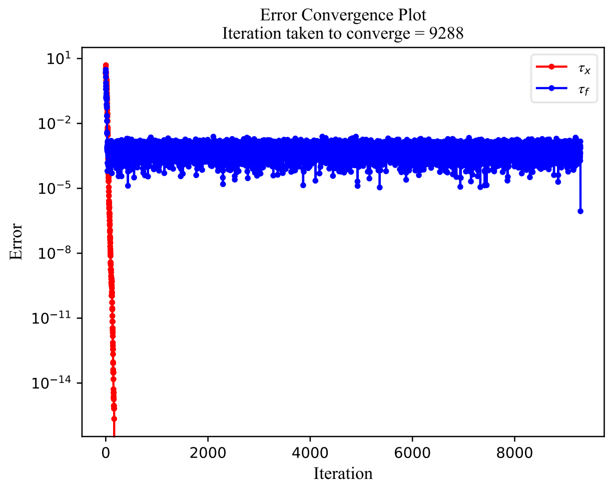
Adding random noise to beam function with a **magnitude of 10^{-4}** using a gaussian distribution:





Adding random noise to beam function with a **magnitude of 10^{-5} (left)** and **magnitude of 10^{-3} (right)** using a gaussian distribution:





Magnitude	Nelder-Mead			SciPy - BFGS		
	x^*	$f(x^*)$	Iteration	x^*	$f(x^*)$	Iteration
10^{-2}	[1.21202883, 0.83973779]	0.0927872629863325	52888	[1.20997798, 0.81792273]	0.0920002893332581 7	12
10^{-3}	[1.2087567, 0.8251257]	0.09205087592456947	9288	[1.21311685, 0.82320967]	0.09194533815204417	11
10^{-4}	[1.21545642, 0.82891037]	0.09198080634579422	3362	[1.21329117, 0.82399511]	0.09194384533280049	10
10^{-5}	[1.21354067, 0.82432068]	0.0919438727984465	112	[1.21341506, 0.82412912]	0.09194381647426471	12
Actual Result	x^* = [1.21341166, 0.82412262]			$f(x^*)$ = 0.09194381641122203		

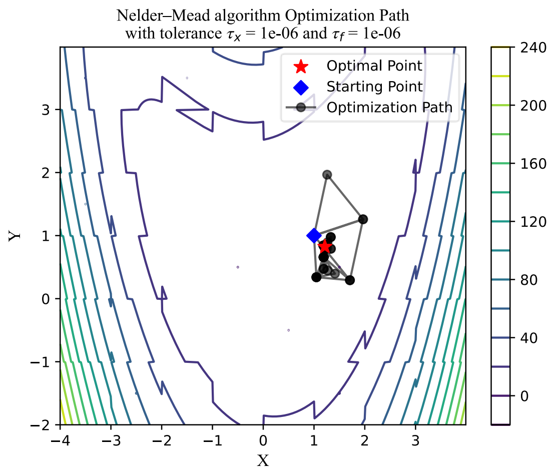
It is evident that the minimum found becomes worse as the noise magnitude is increases for both Nelder-Mead and SciPy-gradient based optimizer. However, gradient bases optimizer performs considerably better than gradient free optimizer as it is closer to minima.

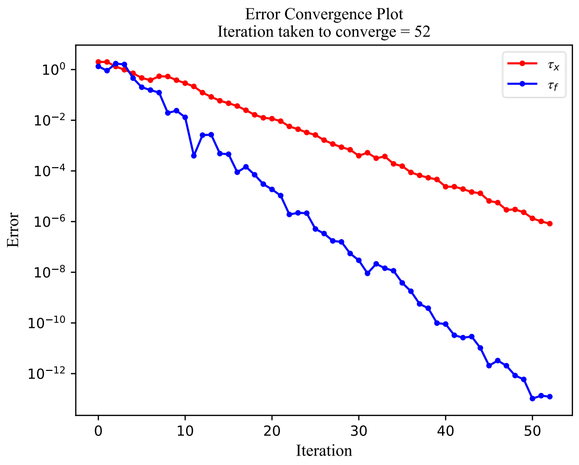
Moreover, the iteration taken to converge to convergence criteria for gradient-free method significantly increases as noise magnitude is increases. As seen from the error graph, the convergence criteria standard deviation of function value seems to hover around the set noise magnitude for most of the computation time. However, there was no difference in number of iterations it took for SciPy, maybe because of different convergence criteria.

Adding checkerboard steps:
Making the bean function multimodal discontinuous function by:

$$\Delta f = 4[\sin \pi x_1 \sin \pi x_2]$$

Thus, Δf was added to bean function by use of *np.celling()*. The plots below are for magnitude = 4.





Magnitude	Nelder-Mead			SciPy - BFGS		
	x^*	$f(x^*)$	Iteration	x^*	$f(x^*)$	Iteration
4	[1.21341152, 0.82412252]	0.09194381641126620	52	[1.21341041, 0.82412157]	0.091943816414044423	10
3	[1.2134117, 0.82412278]	0.091943816411127855	52	[1.2134117, 0.82412262]	0.091943816411122908	10
2	[1.21341154, 0.82412256]	0.091943816411125488	54	[1.21341152, 0.82412281]	0.091943816411152242	6
1	[1.2134117, 0.82412276]	0.091943816411126126	47	[1.2134116, 0.82412252]	0.091943816411123585	7
0.5	[1.2134118 0.82412265]	0.091943816411127877	47	[1.21341168, 0.82412262]	0.091943816411122293	7
Smooth Case	[1.21341156, 0.82412267]	0.09194381641128956	64	[1.21341166, 0.82412262]	0.091943816411122203	14

*** For BFGS, tolerance needed to lower from default and not all starting guess successfully converged.*

For gradient-free method, the minimum function value found was almost the same accurate up to 10^{-12}. However, the optimum point slightly different for each magnitude. Surprisingly the iteration it took was less than for the smooth case. Similar trend can be observed from gradient-based method. Thus, we can conclude that change in magnitude has no significant effect on convergence for both gradient-free and gradient based method. Also, both methods were able to converge to basically the same optimum point with multimodal discontinuous function, given different starting point.

Now minimizing for the function:

$$f(x_1, x_2, x_3) = |x_1| + 2|x_2| + {x_3}^2$$

Method	x^*	$f(x^*)$	Iterations
Nelder-Mead	[-1.17923849e-12, 8.52170073e-13, 1.47291612e-07]	2.9052734526858524e-12	165
SciPy - BFGS	[-2.18369873e-10, -7.45060667e-09, -3.46930226e-06]	1.513161927453196e-08	11

*** SciPy prematurely stops due to error "Desired error not necessarily achieved due to precision loss."*

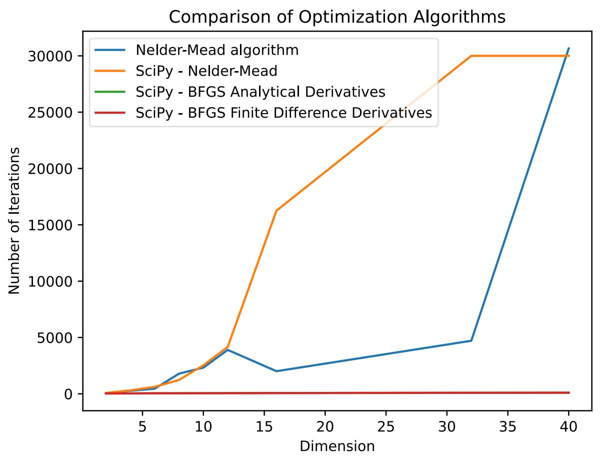
Gradient-Free method outperformed the gradient based method as the optimum result should be $x^* = [0, 0, 0]$ *and* $f^* = 0$. However, iteration taken by Nelder-Mead method to converge is significantly higher than BFGS, but BFGS stopped prematurely due precision loss.

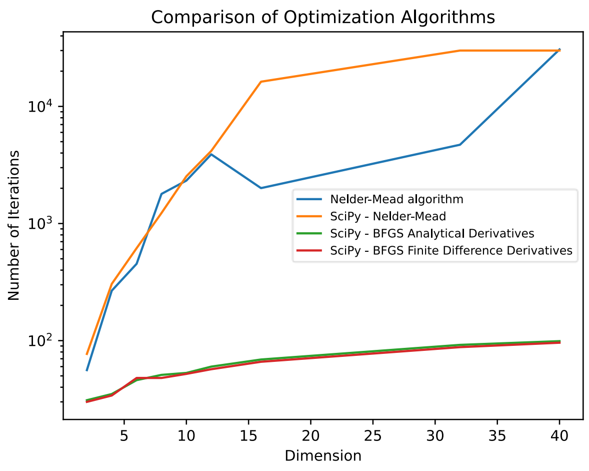
Q3:

*****Dimension represents the number of design variable.***

Implementing on N-D Rosenbrock function with starting guess $x_0 = [2]_n$, where n is the number of dimensions of Rosenbrock function:

Number of Iteration:

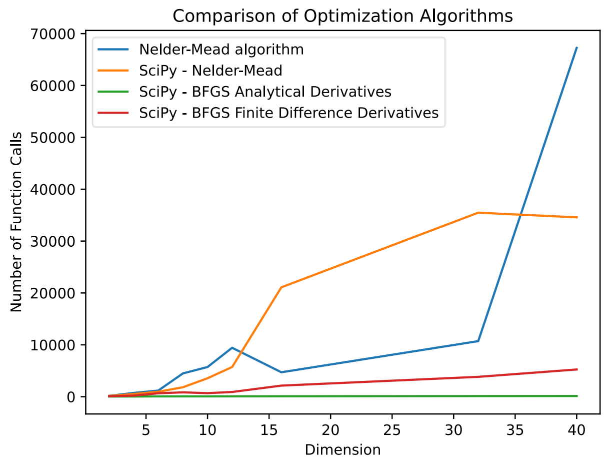


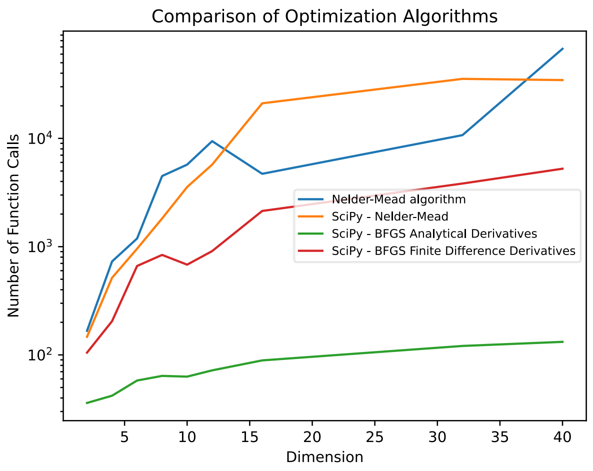


It is evident from the plots that BFGS method with analytical derivatives or Finite difference derivatives is faster than Nelder-Mead algorithm for any dimension of Rosenbrock function. My Nelder-Mead function was more or less had similar iteration to SciPy's Nelder-Mead. However, after 12th dimension it diverges, and my function takes significantly less iteration than SciPy's. However, after 32nd dimension the number of iterations taken drastically increases and is greater than SciPy's for higher dimension.

Nevertheless, it is evident that rate of increase in number of iterations taken to converge for gradient-based method is significantly lower than gradient-based method. Moreover, gradient-free method failed to converge for higher dimension as it exceeds the max iteration of 100,000.

Number of Function Calls:

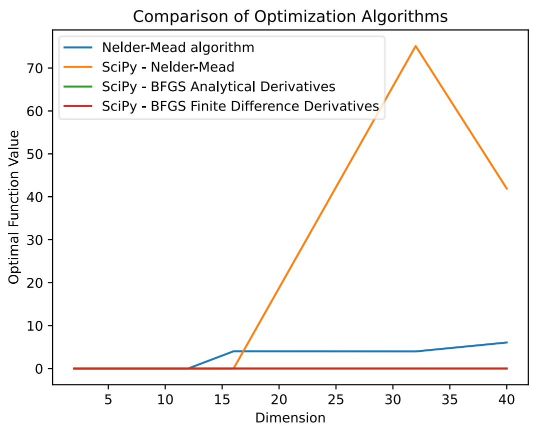


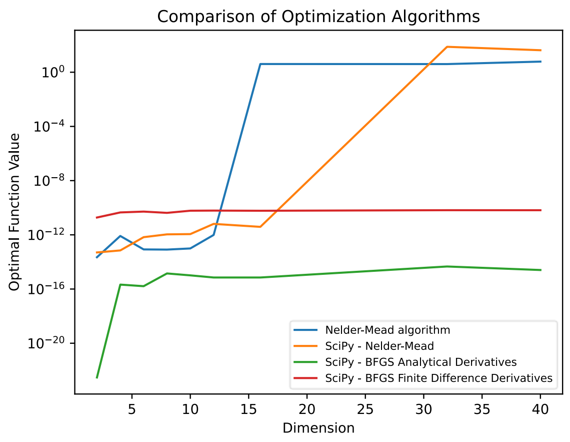


As expected, Nelder-Mead algorithm has significantly more function calls as the number of design variable increases than BFGS. BFGS analytical derivative considerably outperforms the BFGS using finite difference for derivative as expected. BFGS using finite difference for derivative is almost like mine Nelder-Mead Algorithm.

Comparing the rate of change of function calls before 16 design variables, as above it gradient free method converge result are inaccurate. The rate of increase for gradient-based method is about 0.4 and gradient-free method is 2.7. Thus, gradient-free method is only viable for problem with less design variable, however for problem with greater design variable gradient-based significantly outperforms gradient-free methods.

Optimal Function Value:

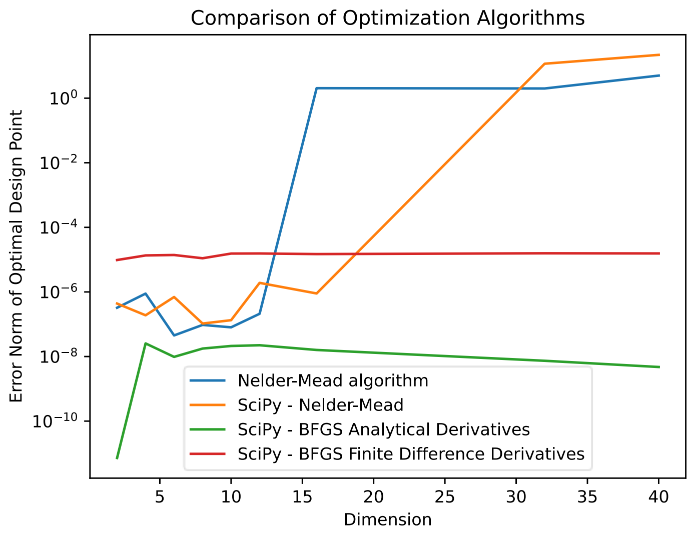




It is evident from the graph that as the number of design variables increases, the optimal solution differs significantly for Nelder-Mead Method. We know the optimal function value is zero, hence the BFGS with analytical derivatives is almost zero and very close to machine precision. However, BFGS using finite difference is considerably worse than analytical derivative, with error magnitude difference of 10^{-6}. Moreover, Nelder-Mead method performs better than it when the number of design variable is less than 16. After which Nelder-Mead Method performs extremely bad. But SciPy's Nelder-Mead method performs significantly worse than mine function where its function values go up to 80, whereas for mine max it goes is 3.2 and seems to however between 2.6 and 3.2.

Error Norm can be computed as:

$$error\ norm = \left\| x^{*} - x^{actuall} \right\|$$
$$x^{actuall} = [1]_n$$



We observe a similar trend, where Nelder-Mead methods performs better than BFGS using Finite Difference but worse than analytical derivatives. However, the error increases drastically after 16 design variables. But, for my function even when the function value was bad at around 2.5 or 3.2, the design variables were only a few decimal places off.