Assignment-3 (Unconstrained Optimization)

Vishwa Mohan Tiwari

Wednesday 11th October, 2023

1 3.1 : Optimizer Development

The development of the unconstrained gradient-based optimization algorithm involves two major stages. The first stage is to develop an algorithm that provides the search direction and the second stage involves having an algorithm that provides the step size. From Engineering Design Optimization (EDO) book, two algorithm for search directions were chosen i.e, Steepest descent and BFGS (Quasi-Newton). Also the `bracketing` and `backtracking` line search algorithms were used to get the stepsize.

1.1 Steepest descent

The algorithm for this search direction finding method is taken from EDO algorithm 4.5. As the gradient points to towards the direction of steepest increase, so -∇f gives us the direction of steepest descent which is used to reach the minimum over multiple steps.

1.2 BFGS (Quasi-Newton)

This method has a higher order of convergence as compared to Steepest descent. Newton's method have a significant disadvantage since they require Hessian. Quasi-Newton method overcomes this by using first order information to approximate the Hessian by using two successive derivatives. For coding this method algorithm 4.7 of EDO is followed.

1.3 Line search: Back-tracking

This step finding algorithm relies on the *sufficient decrease condition* Eq. 1 and backtracking the size of step in the search direction. Algorithm 4.2 from EDO is followed for coding this.

$$\phi(\alpha) \leq \phi(0) + \mu_1 \alpha \phi'(0) \tag{1}$$

1.4 Line search: Bracketing

This is a two step algorithm, first phase involves finding an interval in which we are certain to find a point that satisfies *Strong Wolfe* (1 and 2) and second phase involves finding the point in the interval provided by the first (bracketing) phase. Quadratic interpolation based pinpointing method is implemented here using Algorithm 4.3.

$$|\phi'(\alpha)| \leq \mu_2|\phi'(0)| \tag{2}$$

The code implementation submitted to gradescope.

2 3.2

Since we have two method for getting the search directions and two methods for getting the step size. We have four algorithms that can be used by the optimizer. We know that *Quasi-Newton* methods are faster compared to the first order methods such as *Steepest descent*. Also quadratic interpolation based pinpointing method can be faster that backtracking. But to consolidate our hypothesis, some experiments were conducted on **Slanted quadratic** and **Bean functions**

Algorithm	No. of Major Iteration/No. of func. call for Implement
Steepest descent with backtrack	27/[111]
Steepest descent with bracketing	23/[68]
BFGS with backtrack	7/[22]
BFGS with bracketing	8/[28]

Table 1: For Slanted Quadratic number of iterations taken by different algorithms vs Scipy BFGS

Based of the results in Table 1 and Table 2, the BFGS algorithm with bracketing is selected.

Algorithm	No. of Major Iteration/No. of func. call for Implement
Steepest descent with backtrack	19/[95]
Steepest descent with bracketing	18/[51]
BFGS with backtrack	25/[78]
BFGS with bracketing	27/[84]

Table 2: For Bean function number of iterations taken by different algorithms vs Scipy BFGS

3 3.3

We use `scipy.optimize` software's `minimize` module to solve the two test problems namely, slanted quadratic and 2-D Rosenbrock and compare them with our four optimizations algorithms.
For the slanted quadratic function many values of the hyper parameters were tested and finally certain fixed values were used. The tolerance of converges for the results in Table 3 is

$\tau = 10^{-6}$ and the initial guess is $x_0 = [0, 3]$. The parameters for `backtracking` line search is $\rho = 0.8$ and for `bracketing` line search is $\mu_2 = 0.9$, $\sigma = 2$ and $\mu_1 = 10^{-5}$

Algorithm	No. of Iteration/No. of func. call for Implement	No. of Iteration/No. of func. call for Scipy BFGS
Steepest descent with back-track	27/[111]	8/[24]
Steepest descent with bracketing	23/[68]	8/[24]
BFGS with backtrack	7/[22]	8/[24]
BFGS with bracketing	8/[28]	8/[24]

Table 3: For Slanted Quadratic number of iterations and function calls taken by different algorithms vs Scipy BFGS

For the 2D rosenbrock, the tolerence of converges for the results in Table 4 is $\tau = 10^{-5}$ and the initial guess is $x_0 = [0, 0]$. The parameters for `backtracing` line search is $rho = 0.4$ and for `bracketing` line search is $\mu_2 = 0.7$, $\sigma = 2$ and $\mu_1 = 10^{-5}$

Algorithm	No. of Iteration/No. of func. call for Implement	No. of Iteration/No. of func. call for Scipy BFGS
Steepest descent with back-track	9316/[18695]	25/[75]
Steepest descent with bracketing	4786/[9886]	25/[75]
BFGS with backtrack	562/[1695]	25/[75]
BFGS with bracketing	36/[147]	25/[75]

Table 4: For Rosenbrock 2-D number of iterations and function calls taken by different algorithms vs Scipy BFGS

From the results in Table 3 and 4 we infer that `scipy`'s `BFGS` is a superior algorithm than our implementation of all the algorithms. Cost of our implementation of BFGS is close to that of `scipy`, but some hyper parameter tuning can be done. The steepest descent is defeated by the BFGS algorithm by a long margin while solving Rosenbrock function. Overall we find out that the BFGS implementation with `bracketing` is the best algorithm we have .

4 3.4

Since we assume that the exact solution of objective functions might not be known, so we plot the $|\nabla f|_\infty$ against the number of iterations.
We obtain rate of convergence greater than one but less than two for the slant quadratic and 2D Rosenbrock with BFGS from semi-log plots of Fig. 1 and 2. This is because quasi-newton method uses the curvature information but the Hessian is just an approximation of the true Hessian, hence we don't get order 2 convergence.

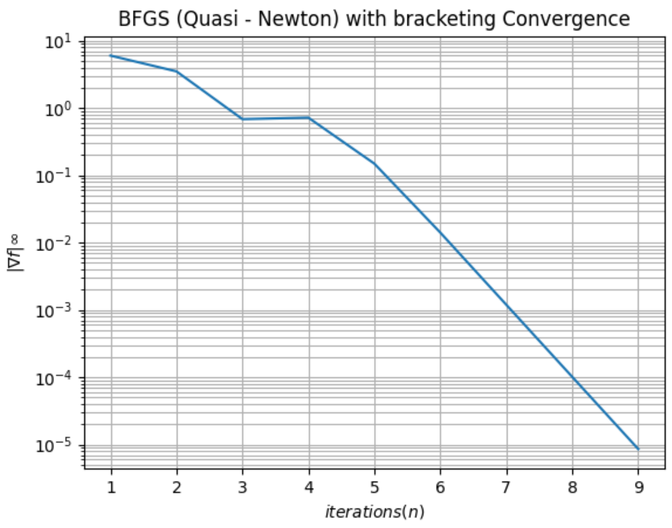

Figure 1: Slanted quadratic convergence plot, the rate convergence = 1.08

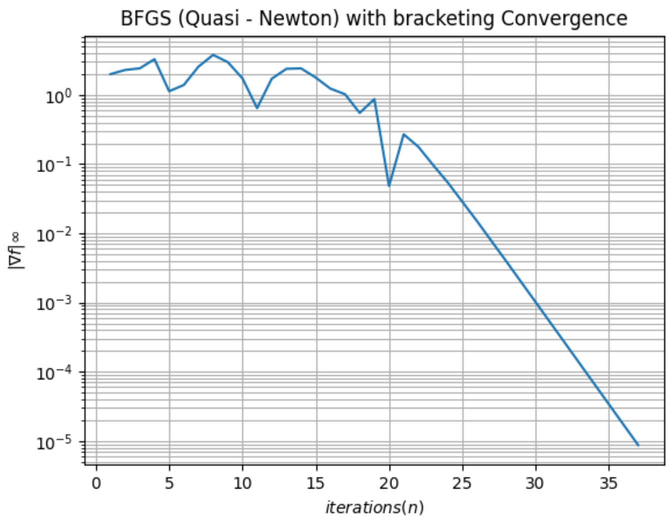

Figure 2: Rosenbrock convergence plot, the rate convergence = 1.0008

The formula used for calculating the convergence rate is give by Eq. 3

$$p = \frac{\log(||\frac{x_{k+1}-x_k}{x_k-x_{k-1}}||)}{\log(||\frac{x_k-x_{k-1}}{x_{k-1}-x_{k-2}}||)} \tag{3}$$

5 3.5

The optimization path of the BFGS optimizer with `bracketing` is shown in Fig.3 and 4 for slant quadratic and 2-D rosenbrock respectively.

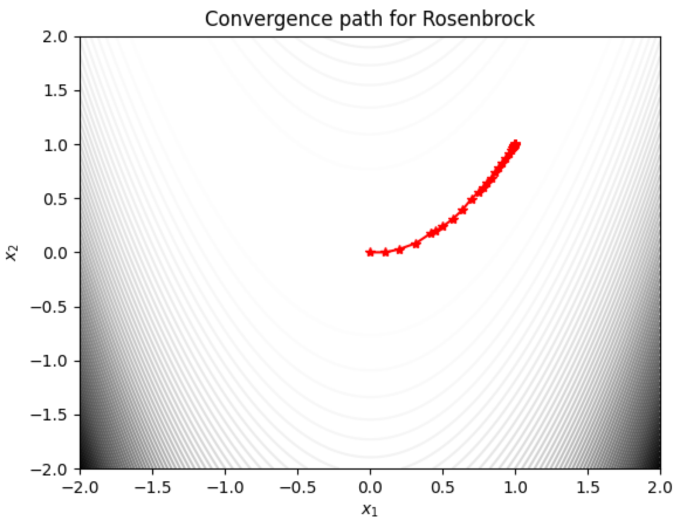

Figure 3: Path taken by BFGS optimizer (shown in red) to convergence for 2-D Rosenbrock

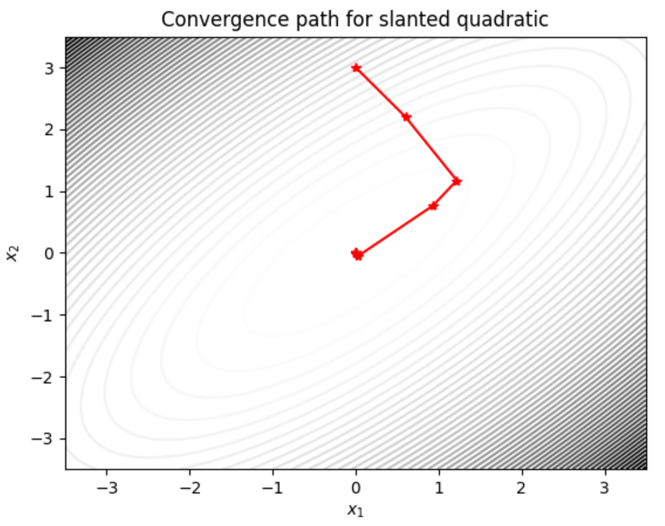

Figure 4: Path taken by BFGS optimizer (shown in red) to convergence for slanted quadratic

It is observed that unlike steepest descent where we have near orthogonal path of the optimizer when the curvatures are drastically different, here we see smooth paths both in the case of slanted quadratic and 2-D rosenbrock.

6 3.6

The computational cost metric used here is number of function calls and the number of iterations the optimizer. In terms of function evaluations, it is observed that the implemented

BFGS performs better than `scipy` as the dimensionality of the problem increases. The semi-log plot is used for showing the variation of function evaluation with dimensionality because the number of function evaluations in the two cases differ by two order of magnitudes shown in Fig. 5. In terms iterations as the metric for computational cost we observe that `scipy` takes more number of iterations compared to implemented BFGS as the dimensionality increases shown in Fig. 9.

This could be explained using following arguments:

1. The implemented algorithm used analytical gradients whereas the `scipy` algorithm doesn't have analytical gradients explicitly.
2. There is no reset used for implemented BFGS algorithm, but the `scipy` uses certain conditions that might require more function invaluations.
3. The manually tuned hyper-parameters in case of implemented BFGS help it to converge faster than `scipy`.

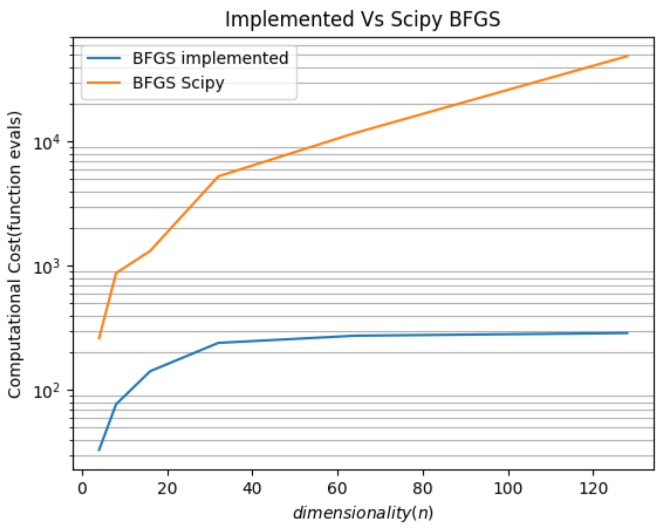

Figure 5: Semi- log plot of number of function evaluations against dimensionality for BFGS implemented and Scipy

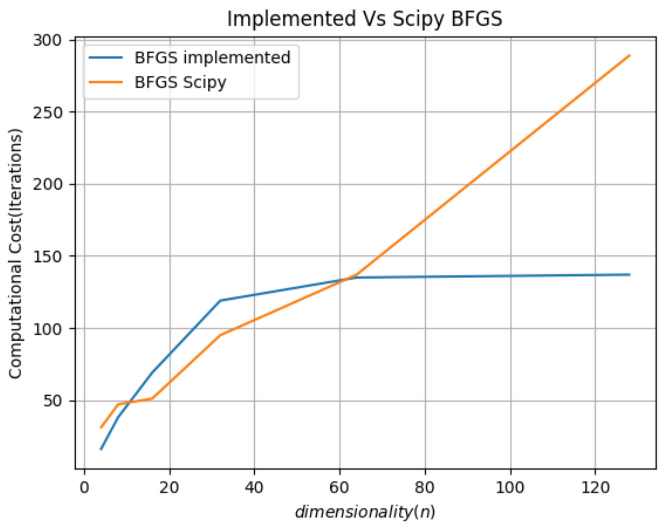

Figure 6: Plot of number of iterations against dimensionality for BFGS implemented and Scipy

7 3.7

There following challenges that were involved:

1. Tuning the hyper parameters was the most time consuming part of the whole development process. I spent days struggling with convergence.
2. Auto-tuning is definitely an improvement that will make the algorithm robust.
3. The line search does act as a globalization strategy, but the current implement has a initialization dependence. A better globalization strategy could be used to tackle this issue.

8 3.8

9 Appendix

We show the convergence plot of n = 8,32,128 cases

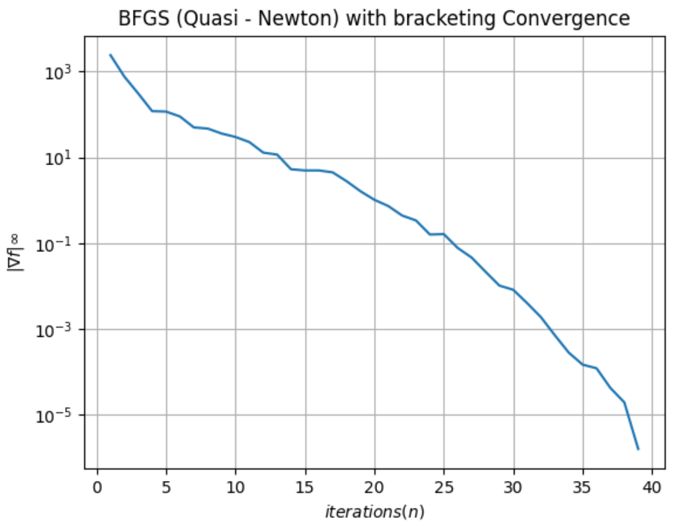

Figure 7: Convergence of 8D Rosenbrock

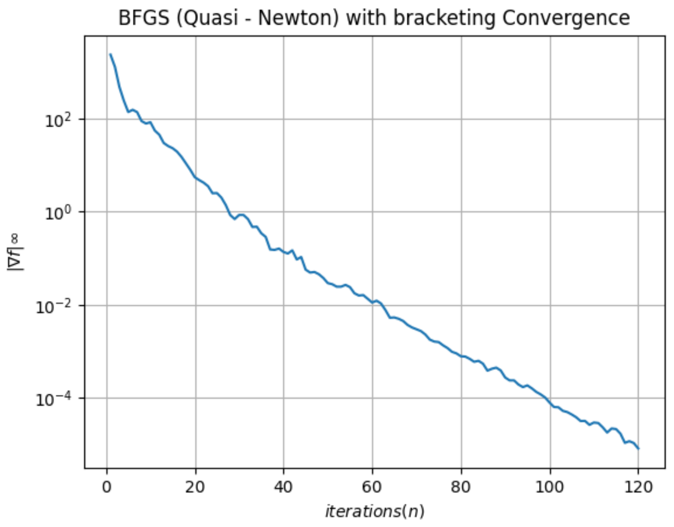

Figure 8: Convergence of 32D Rosenbrock

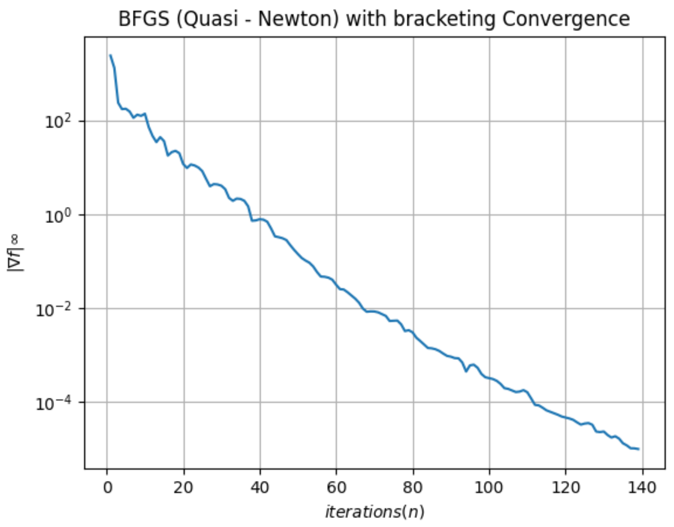

Figure 9: Convergence of 128D Rosenbrock