Assignment-6 (Gradient-free Optimizers)

Vishwa Mohan Tiwari

Saturday 2nd December, 2023

1 Implementing Gradient free optimizer: Nelder-Mead

1.1 Nelder-Mead

Nelder-Mead is one of most popular optimizers for gradient -free optimization. It uses **simplex method** of Nelder-Mead, and it's a deterministic, direct search based optimizer. The algorithm implemented in this project was based of Engineering Design Optimization (EDO) book, *algorithm 7.1.* This algorithm uses simplex which is a geometric figure defined by a set of '$n+1'$ points in the design space of n variables. Major steps of the algorithm are given out below:

1. We start by giving the algorithm an initial design. The first step involves generating '$n+1'$ design points based on the initial guess. This is done by adding steps to initial design to generate 'n' new points.

$$x^{(i)} = x^{(0)} + s^{(i)} \tag{1}$$

   Here each $x^{(i)}$ is a design and $s^{(i)}$ taken such that all sides of the simplex have the same length 'l'.

2. Upon obtaining these designs we evaluate the objective function at these points and sort the simplex points so that $x^{(0)}$ is the best design and $x^{(n)}$ is the worst.

3. This algorithm further performs five main options on the simplex to create a new one. Expect for shrinking all other operations produce new points. Each iteration aims to improve the worst point to form a better simplex.

$$x = x_c + \alpha(x_c - x^{(n)}) \tag{2}$$

$$x_c = \frac{1}{n}\sum_{i=0}^{n-1} x^{(i)} \tag{3}$$

   The operations are: *reflection, expansion, outside contraction, inside contraction and shrinking.* We start with reflection by putting $\alpha = 1$ in equation 2, where x_c the centroid formed by all points expect the worst point $x^{(n)}$. Based on whether the reflected point was good enough or not we go ahead and perform operations such as

expansion($\alpha = 2$), outside contraction($\alpha = 1/2$), inside contraction($\alpha = -1/2$) and shrinking (equation 4).

$$x^{(i)} = x^{(0)} + \gamma(x^{(i)} - x^{(0)}), \gamma = 0.5 \tag{4}$$

4. To check convergence we'll be checking the size of the simplex, and the difference between f_{best} and f_{worst}.

```
def Neld_Mead(x0,tau_x=10**(-1), tau_f=10**(-6)):
    """
    Inputs -
    x0 : Starting design (array of dim values)
    tau_x : simplex size tolerances
    tau_f : Function value of s.d tolerances

    Outputs -
    x_star : Optimal point
    """
    X = np.array(x0.reshape(-1,1))
    n = np.shape(x0)[0]
    for j in range(1,n+1):
        temp = np.zeros((n))
        for dim in range(n):

            temp[dim] = x0[dim] + s(j, dim+1, n)
        X = np.hstack((X, temp.reshape(-1,1)))

    ## Starting the while loop
    delta_x = 1
    delta_f = 1
    count = 0
    while delta_f> tau_f:
        count +=1
        ### sorting the designs ####
        sorted_X, fun = sort_desi(X) #user defined funciton that sorts
   designs
        xc = np.mean(sorted_X[:,:n],axis=1) #centroid
        xn = sorted_X[:,-1]  # last design after sorting
        xr = xc + (xc - xn)    #reflection
        fxr = f(xr)
        fxn = f(xn)
        if (fxr < fun[0]):
            xe = xc + 2*(xc - xn) #expansion
            print("expansion done!")
            .....
        elif (fxr <= fun[-2]):
            xn = xr
            sorted_X[:,-1] = xr
        else :
            if fxr > fxn:
                xic = xc - 0.5*(xc - xn) # inside contraction
                    .....
                else :
```

```
                    ......
            else:
                xoc = xc + 0.5*(xc - xn) # outside contraction
                if f(xoc) < fxr :
                    xn = xoc
                    sorted_X[:,-1] = xoc
                else :
                    for j in range(1,n+1):
                        sorted_X[:,j] = sorted_X[:,0] + 0.5*(sorted_X[:,j]
    - sorted_X[:,0])


        delta_f = abs(f(sorted_X[:,-1])- fun[0]) # difference between best
    and worst
        X = sorted_X
        print("count {}".format(count))
    return X,sorted_X, X[:,0]
```

Listing 1: Nelder- Mead code structure

2 Studies on the Implemented Optimizer

2.1 1. a

We use our gradient-free 'Nelder-Mead' optimizer on Example 7.1 of EDO book. The initial guess is given as $x_0 = [-1.2, 1.5]$ and the objective function of our interest is bean function. The Figure 1 shows the various simplex formed in the path of convergence to the optimal design/point for the case of Nelder-mead optimizer. Further we see the path of gradient based 'BFGS' optimizer. The implemented gradient-free optimizer took '140' and gradient -based BFGS took '30' function calls.

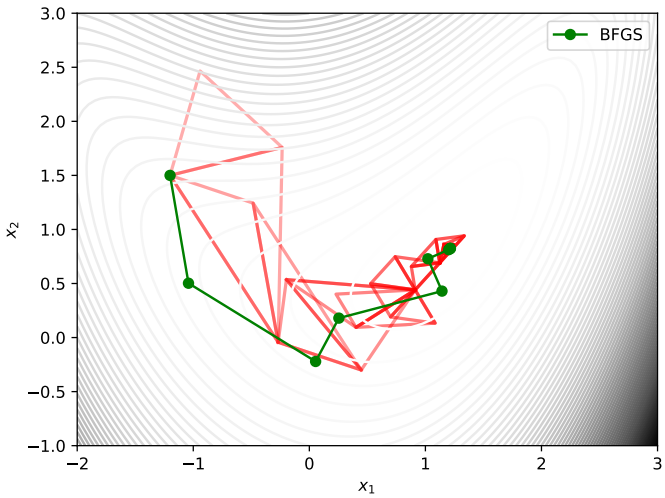

Figure 1: Path of convergence for the bean function for Nelder-Mead and BFGS

2.2 1. b

In this section we introduce Gaussian noise to our objective function and corresponding gradients, after which we further analyse the performance of the optimizers.

Figure 2 shows the convergence path of the gradient-free and gradient based for bean function with gaussian noise of magnitude 10^{-4}.

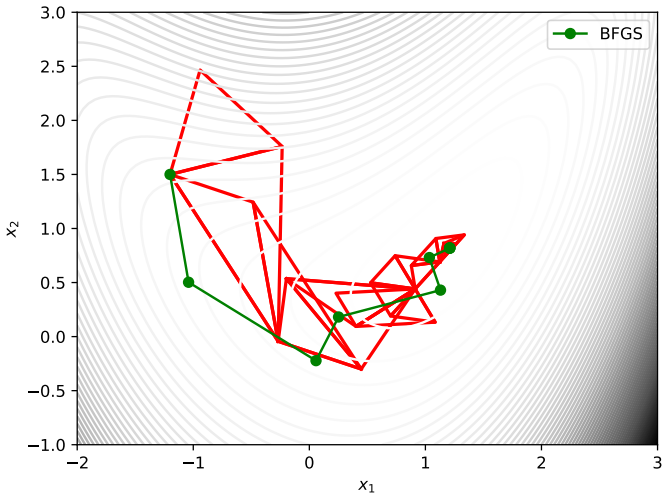

Figure 2: Path of convergence for the bean function for Nelder-Mead and BFGS with noise of amplitude 10^{-4}

The gradient free optimizer used 560 evaluations for converging to the optimal value which is higher than what it took for the same function without noise which was '140' function evaluations. Whereas for the gradient based we don't see much change, as it takes 22 iterations. These results might be hard to reproduce since no specific seed was used while generating the randomness for the gaussian noise.

2.2.1 Extra Credit

We can further analyse the impact of the magnitude of noise on the performance of the optimizers by varying the magnitude of the noise and then observing the function evaluations taken to converge.

Figure 3 shows that gradient- free takes tremendously high number of evaluations compared to the gradient-based. Also the number of function evaluations increase with the increase in magnitude of noise for the gradient free. But for gradient-based there is only a small/ steady increase in the number of evaluations.

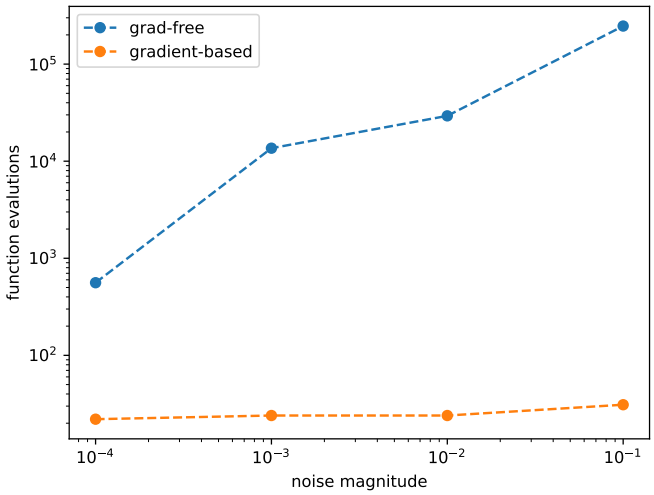

Figure 3: Impact of noise on the optimizer performance in terms of the function evaluations.

Another key observation here is that as the magnitude of noise increases the optimizers converge to a optimal value far from that of the function without any noise. Especially for the case of gradient-based optimizer where the noise is added in the gradient as well. For the case when the noise magnitude was 10^{-1} it converged to (1.14048988, 0.70523222) compared to the gradient free which converged to (1.21340165 0.82411128). So here, the accuracy of the gradient-based optimization is getting impacted more as compared to the gradient free.

2.3 1. c

In this section we introduce dis-continuity in the objective function by adding ceiling of the product of sines ($4\lceil \sin(\pi x_1)\sin(\pi x_2)\rceil$) to Jone function as demonstrated in example 7.9 in EDO.

While calculating the gradients we can ignore the ceil term which is added to the Jone function, because this doesn't impact gradient when $\sin(\pi x_1)\sin(\pi x_2)$ not an integer. The function is basically getting shifting in the z direction if the design variables x_1 and x_2 are in x and y direction respectively. The new discontinuous function gradient is not defined only when the function inside ceil is an integer. For most of the case it may not impact the gradient based optimizers drastically.

Figure 4 shows the convergence of the 'Nelder-mead' and 'BFGS' starting from the $x0 = [-1.2.1.5]$ with step magnitude of 4. Figure 5 shows a convergence form smooth Jones functions.

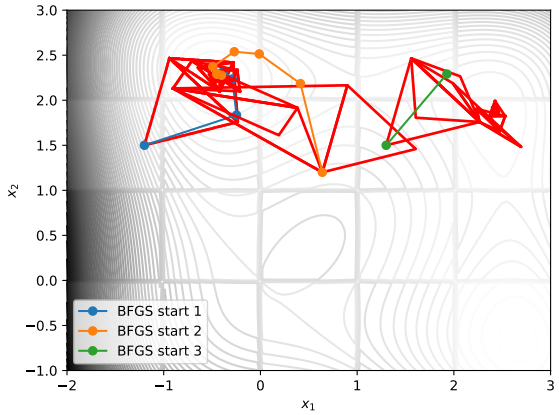

Figure 4: Nelder-Mead and BFGS multi-start for step magnitude 4.

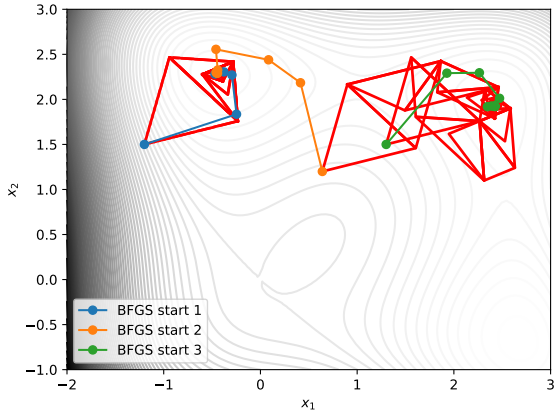

Figure 5: Nelder-Mead and BFGS multi-start for step magnitude 0.

From these Figures we observe that introduction of step can impact the optimal value to which the gradient based and gradient free optimizers converge. It can we be seen in start 2 case where Nelder-Mead converges to a different optimal value in Figure 7 and 5.

2.3.1 Extra Credit

While changing the amplitude of the steps we observe the following :

1. The point of convergence of the gradient-free optimizer is impacted as it solely depends on the function value. Figure 6 and 4 show that start 2 'Nelder-mead' converge to different optimal value for step magnitude of 3 and 4

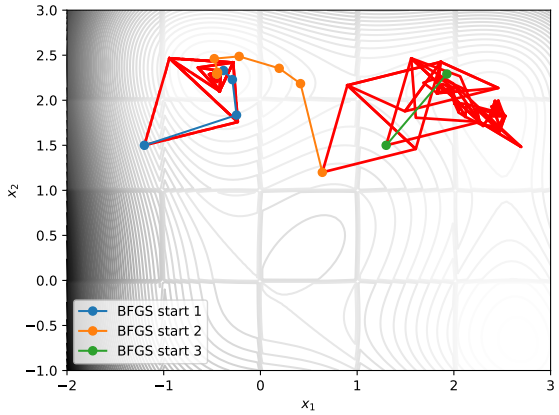

Figure 6: Nelder-Mead and BFGS multi-start for step magnitude 3.

2. The gradient-based optimizer can get stuck at the point where the function jumps because of the ceil function as shown in Figure 6 (step magnitude = 3, BFGS start 3). But as the magnitude of step is decreased the chances of these optimizers to get stuck at the boundary (where gradients become not defined) decreases as shown in Figure 7 (step magnitude = 0.4)

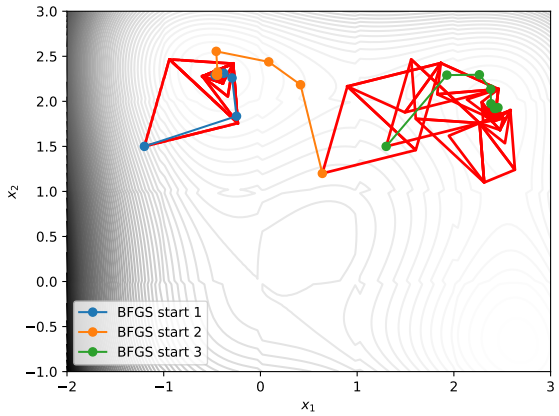

Figure 7: Nelder-Mead and BFGS multi-start for step magnitude 0.4.

2.4 1. d

Since the given function is always positive, it's minimum has to be zero which occurs at (0,0,0). We use gradient-based BFGS, scipy and implemented Nelder-Mead to minimize this. To reach a simplex size tolerance of 10^{-5} the scipy Nelder-Mead takes 317, implemented Nelder-Mead takes 689 and the gradient-based BFGS takes 408 function evaluations.

We observe that these optimizers struggle to converge as not all design variables come close to the optimal at once as shown in Table 1

Optimizer	No. funcs eval	optimal value
Implemented Nelder-Mead	689	$(-3.47e^{-12}, -5.24e^{-11}, 1.59e^{-06})$
Scipy Nelder Mead	317	$(7.84e^{-01}, 4.56e^{-10}, 3.13e^{-01})$
Scipy BFGS	408	$(-5.87e^{-09}, -1.174e^{-08}, 6.12e^{-02})$

Table 1: The table with function convergence data for the given function

The presence of discontinuities due to absolute value can be a big reason for this difficulty in convergence of this function. Further this can also be reason for poor performances of BFGS, where it under performs compared to the scipy's gradient-free optimizer.

3 Effect of increased dimensionality

The Figure 9 shows the trend of the number of function evaluation needed as the number of design variables increase.

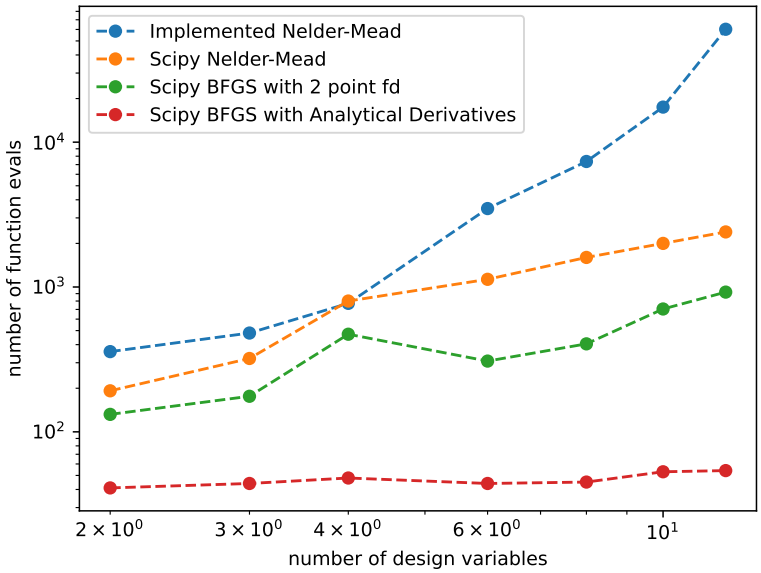

Figure 8: Cost of optimization for increasing with number of design variables in the n-dimensional Rosenbrock function

We observe from the Table 2 that Implemented Nelder-Mead has higher slope, followed by scipys Nelder-Mead, BFGS with 2-point (stencil) finite difference and BFGS with Analytical derivatives.

Optimizer	Slope
Implemented Nelder-Mead	2.415
Scipy Nelder-Mead	1.45119
Scipy-BFGS with finite-difference	1.19459
Scipy BFGS with Analytical	0.153

Table 2: Slopes for different Optimizers

Further we get issues in convergence for finite difference based BFGS, due to lack of precision in the gradient found. Which leads to the optimizer converging to a slightly wrong optimal.

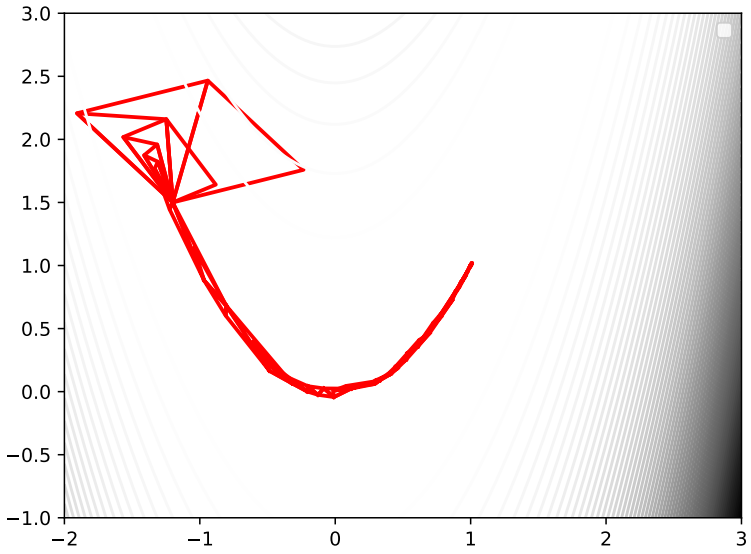

Figure 9: 2D Rosenbrock convergence path for Nelder-Mead

The Implemented Nelder- Mead becomes even more expensive as we go for higher dimensions. For 2D case Nelder-Mead took 358 function evaluations compared to 123 of BFGS.