AE588 Assignment 6 - akshatdy

6.1

I chose to implement the Nelder-Mead algorithm because there was an easy to follow example in the book, and it is much easier to visualise than the other algorithms.

Nelder-mead works by first creating a simplex, which is a collection of n+1 points in the design space, where n is the number of design variables. This simplex is then modified in every iteration of the optimization to bring the simplex closer to the optimum. The simplex can be modified in the following ways:

- Reflection: Calculate the centroid of all points except the worst one. Then "reflect" the worst point through the centroid and evaluate the function at this reflected point.
- Expansion: If the reflected point is the best point so far, then try to extend the simplex in this direction, creating an expanded point, and evaluate the function there.
- Contraction (Inside/Outside): If neither reflection nor expansion improves the worst point, attempt to perform a contraction to reject the worst point, either inward or outward, and evaluate the function at the new point.
- Shrink: If contraction still doesn't work, then we shrink the whole simplex toward the best point.

The algorithm stops when either

- Number of iterations has reached a limit
- The distance between the points of the simplex is too small
- The standard deviation of the function values in the simplex is too small

6.2

General notes:

- In my implementation of Nelder-Mead, I have adjusted the maximum iterations allowable to ensure that the optimizer always exits only when convergence critera for Δ_f and Δ_x are met.
- The convergence criteria for SciPy based optimizers have been left at their defaults.
- When using a gradient based method, I provide the function and gradient separately to SciPy. Hence, when calculating the total function calls (fev), I have added both the function and jacobian evaluations for the SciPy BFGS cases with analytical gradients.

6.2.a

Results for my Nelder-Mead implementation:

fev	x^*	$f(x)^*$
118	[1.2134114738702 0.8241223344287]	0.0919438164113

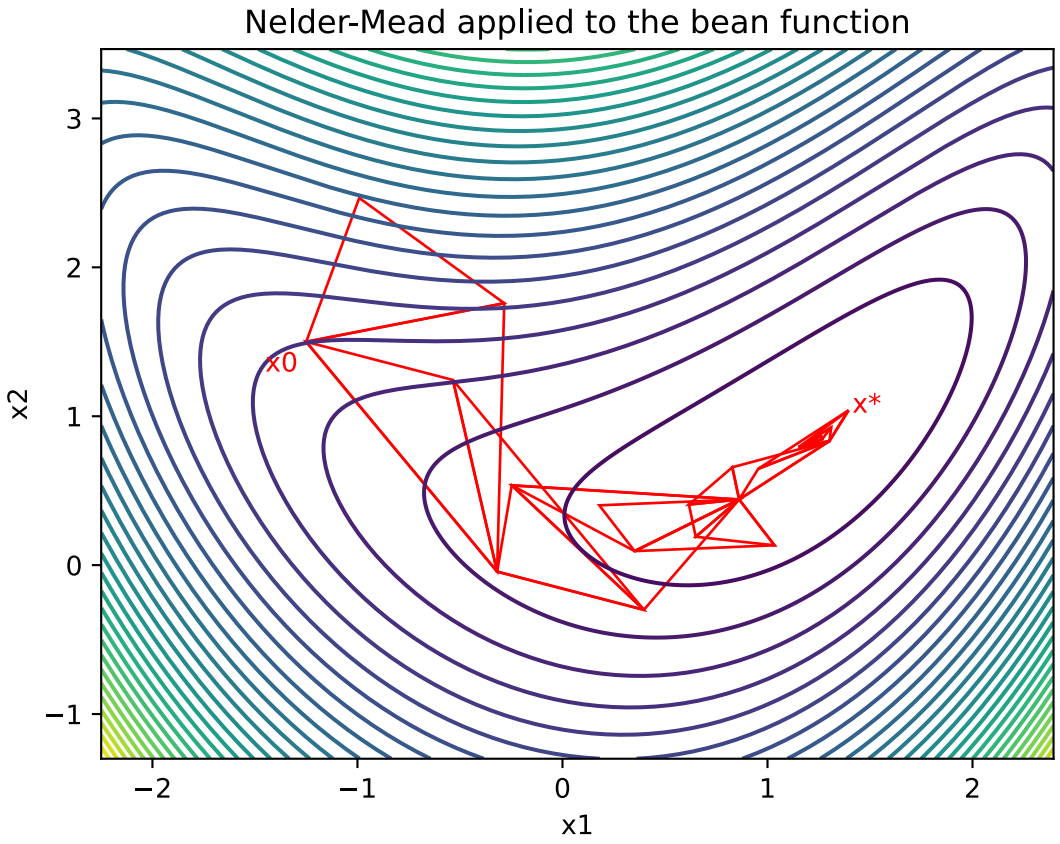


Figure 1: Optimization path for Nelder-Mead on the Bean function

6.2.b

Results comparing the optimization of a noisy bean function between:

- my implementation of Nelder-Mead
  - This was given enough iterations so that it never stops unless the other convergence critera are met
- Scipy BFGS and analytical gradient that is consistent with the noisy bean function
  - left at default settings so it may stop without meeting convergence criteria
  - function evaluations and jacobian evaluations are added to get a full picture of the total "function" calls since those are provided separately

The noise was varied from 10^{-1} to 10^{-9}, but only certain important results are displayed here so the table can look nice

The results in the table are based on the difference between

- a baseline optimum given by BFGS with analytical gradients on smooth bean
- the optimum given by Nelder-Mead and BFGS on a noisy bean

The noise as asked in the question has been bolded for visibility

noise	1e-09	1e-07	**1e-04**	1e-03	1e-01
NM x_1^*	2.0163130543e-07	1.7823436629e-06	**1.1398095279e-03**	1.1137357362e-02	1.4233977454e-01
BFGS x_1^*	2.1221663982e-08	1.7827848480e-06	**1.1387659005e-03**	1.1137250698e-02	3.0250005917e-01
NM x_2^*	1.0921469717e-07	3.1169619389e-06	**3.2864916927e-04**	1.3311632814e-02	2.1351179809e-01
BFGS x_2^*	3.0562184006e-08	3.0765705131e-06	**3.2676560626e-04**	1.3310648521e-02	4.3373146544e-01
NM $f(x)^*$	8.9171718620e-10	6.0122489765e-08	**1.0421326057e-04**	1.2019839886e-04	3.4746256407e-02
BFGS $f(x)^*$	8.9175596074e-10	6.0122437681e-08	**1.0421325524e-04**	1.2019881706e-04	1.2929027499e-01
NM fev	118	116	**119**	152	165
BFGS fev	22	22	**24**	128	260

We can see that in most cases with low noise, BFGS beats or matches the precision of Nelder-Mead, with fewer function evaluations. However, as the noise increases, around 10^{-3}, BFGS starts to struggle, with more function evaluations than Nelder-Mead. Eventually at 10^{-1}, the result from BFGS is a lot worse than the result from Nelder-Mead with more function evaluations.

This might happen because as the noise increases, the gradient that BFGS uses gets more and more affected by the noise as it starts to converge towards the minimum, and is unable to proceed in the correct direction.

This behaviour can be seen if we trace the optimization path of Nelder-Mead with BFGS at an acceptable noise level, like 10^{-4} and again at 10^{-3}, where you can see that it starts zig-zagging a lot more towards the end trying to find the optimum

From this, we can conclude that Nelder-Mead is better suited to situations where the noise is quite high proportional to the function and gradient value near the optimum.

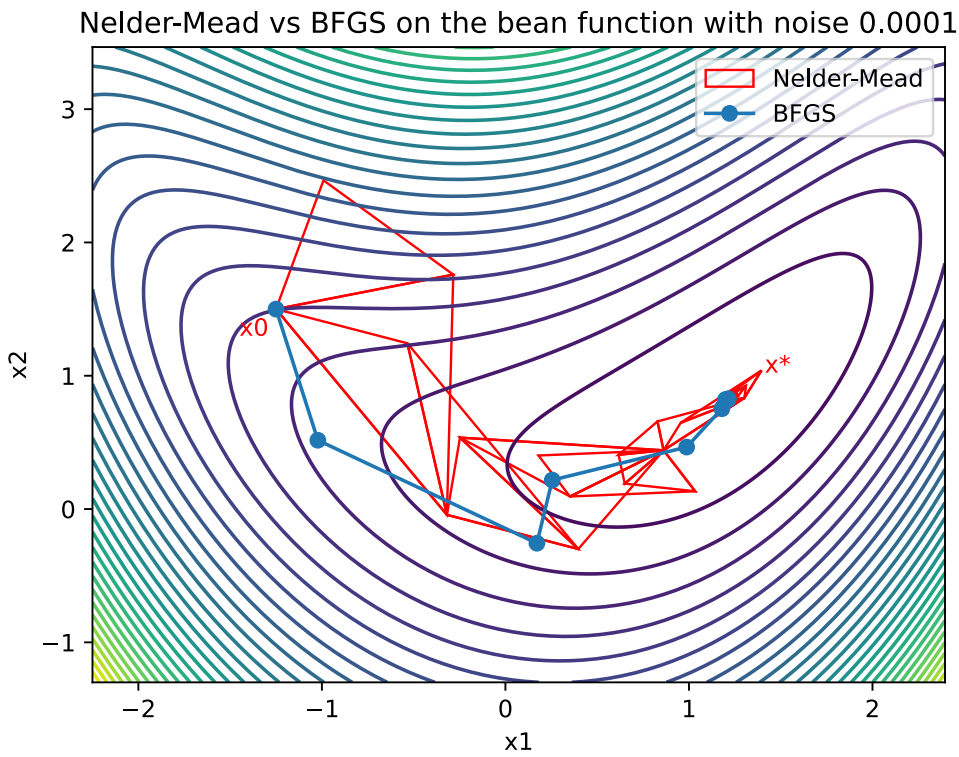


Figure 2: Noise of 1e-4

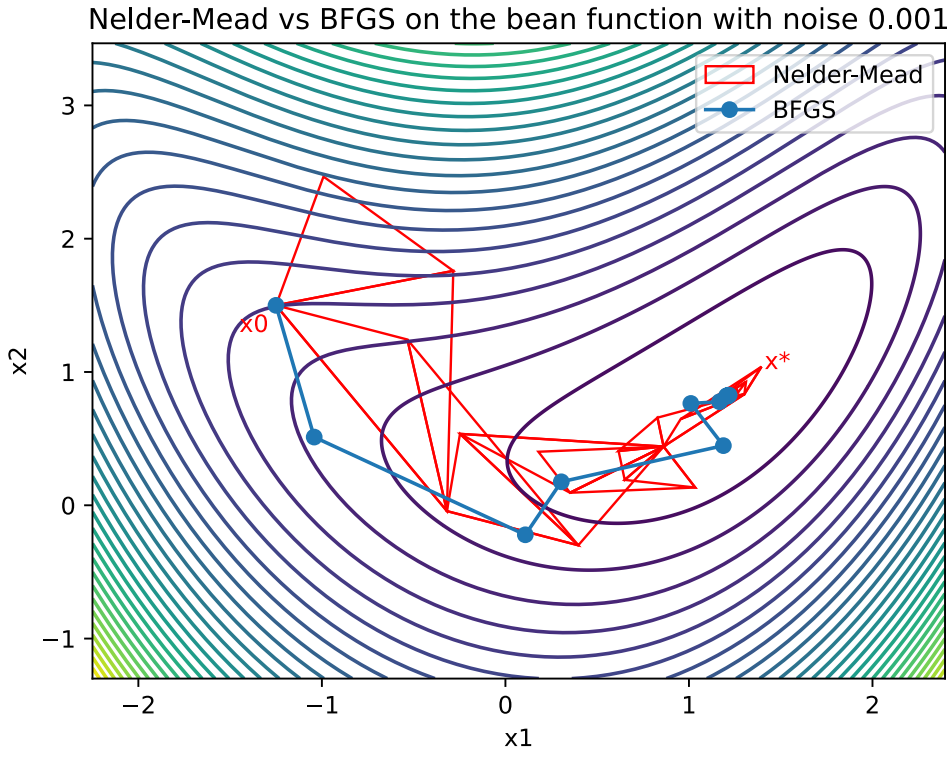


Figure 3: Noise of 1e-3

6.2.c

Results comparing the optimization of a bean function with checkerboard steps between:

- my implementation of Nelder-Mead
  - This was given enough iterations so that it never stops unless the other convergence critera are met
- Scipy BFGS and analytical gradient that is **same as** the smooth bean function
  - left at default settings so it may stop without meeting convergence criteria
  - function evaluations and jacobian evaluations are added to get a full picture of the total "function" calls since those are provided separately

The step was varied from 5 - 4.5 in steps of 0.5, but only certain important results are displayed here so the table can look nice

The results in the table are based on the difference between

- a baseline optimum given by BFGS with analytical gradients on smooth bean
- the optimum given by Nelder-Mead and BFGS on a bean with checkerboard steps

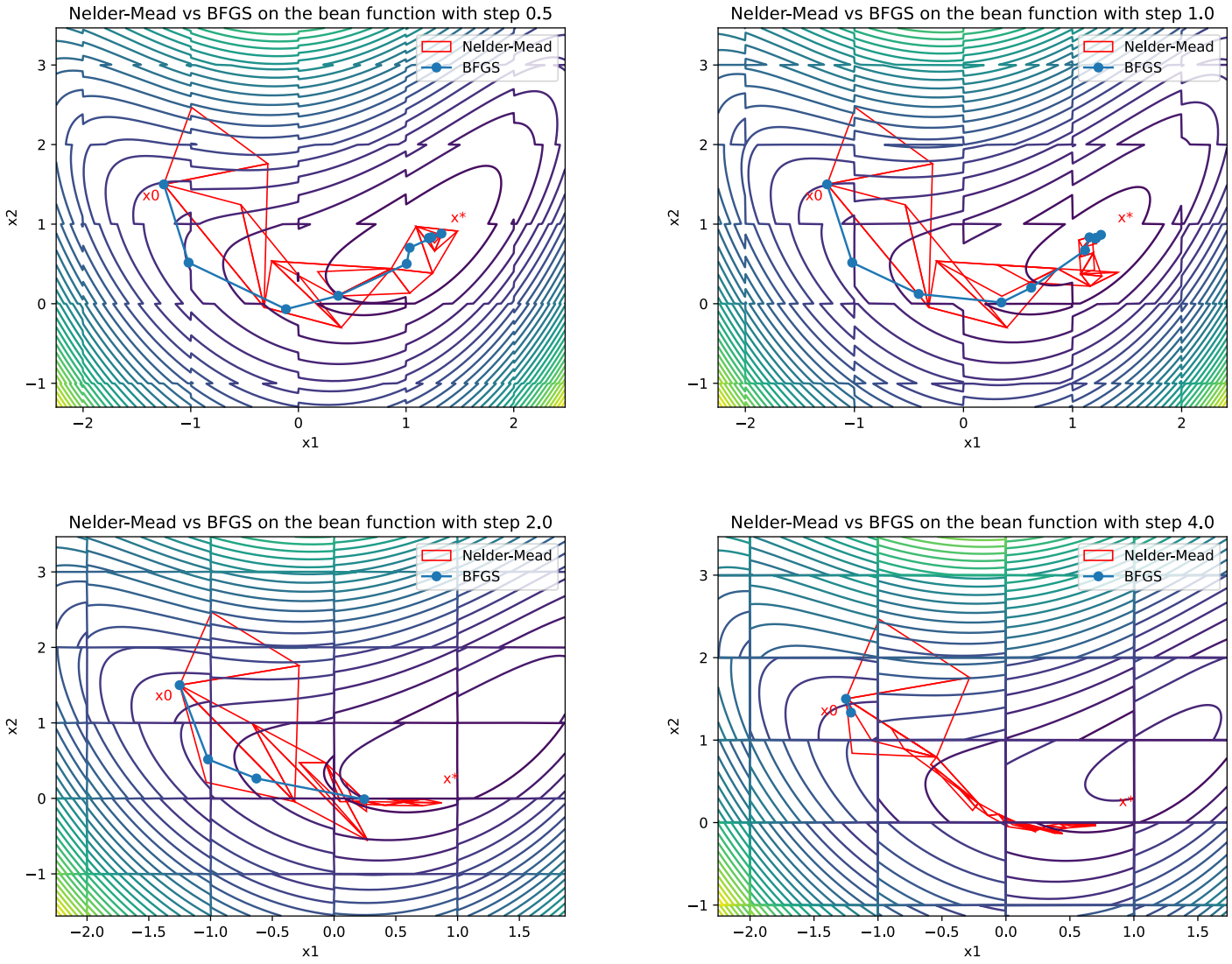
The step as mentioned in the book has been bolded for visibility

step	0.5	1	2	**4**
NM x_1^*	1.9525311101e-07	3.7481327375e-08	5.3108398495e-01	**5.3108513572e-01**
BFGS x_1^*	5.2850222865e-08	1.4886810451e-08	9.6900960373e-01	**2.4245849311e+00**
NM x_2^*	5.3761213215e-07	5.8140068204e-08	8.2412244364e-01	**8.2412244364e-01**
BFGS x_2^*	3.5659314945e-08	2.2196733696e-07	8.3577030009e-01	**5.0985975251e-01**
NM $f(x)^*$	8.2503448517e-14	3.4486302702e-14	1.1173500938e+00	**1.1173500938e+00**
BFGS $f(x)^*$	1.5626389072e-14	1.0322298571e-13	1.5058626297e+00	**5.6301164658e+00**
NM fev	114	123	231	**266**
BFGS fev	24	24	144	**125**
BFGS conv	True	True	False	**False**

We can see that in cases with a large step like 4, Nelder-Mead beats the precision of BFGS by a huge margin. However, as the step decreases, around 1 and 2, BFGS becomes comparable. Eventually at 0.5, the result from BFGS is better than the result from Nelder-Mead. Throughout all step sizes, BFGS consistently has fewer function evaluations than Nelder-Mead, however the function calls rise sharply for sizes 2 and 4.

This might happen because as the step size increases, the line search that BFGS uses has a higher chance of getting confused by the step if the function value after the step is a lot higher than before it. This might cause BFGS to get stuck and not be able to converge. This can be seen in the status of the result returned by the SciPy optimizer, as it is not able to converge for step sizes 2 and 4.

This behaviour can be seen if we trace the optimization path of Nelder-Mead with BFGS at all the step sizes, where you can see that it starts getting stuck near the checkerboard edges as the step size increases, and is not able to even proceed beyond the first few iterations for a step size of 4.



From this, we can conclude that Nelder-Mead is better suited to situations where the the function is extremely discontinuous.

6.2.d

The same function was optimized with both BFGS and Nelder-Mead

Method	conv	fev	x^*	$f(x)^*$
NM	True	319	-1.17968e-12 8.52064e-13 1.47292e-07	2.90550e-12
BFGS	False	112	-4.21251e-01 2.54760e-02 -1.46982e-02	4.72419e-01

Nelder-mead works beautifully here, as opposed to BFGS. We can see that BFGS failed to converge, as seen by the "conv" column. This makes sense, as the optimim point is at [0,0,0], where the gradient is undefined due to the absolute value in the function, and causes a divide by 0 for BFGS. This makes the problem ill-conditioned and prevents BFGS from getting close to the optimum and fail to converge.

From this, we can conclude that Nelder-Mead is better suited to situations where the function has a discontinuous gradient, and is not C2 continuous, which is a requirement for BFGS. If we want to make it work with BFGS, we will have to modify the function to be continuous near the optimum, using methods similar to Kreisselmeier–Steinhauser (KS) aggregation for constrained optimization.

6.3

General Notes:

- When using BFGS with analytical gradient, I provide the function and gradient separately to SciPy. Hence, when calculating the total function calls (fev), I have added both the function and jacobian evaluations.
- My implementation of Nelder-Mead has an iteration limit of 10000, which ends up being ~12000 to ~13000 function evaluations, after which it reports that it failed to converge.
- The value of x mentioned in the table is the norm of (x returned - optimal x[1, 1, ...]) so that higher dimensions can be reasonably compared.
  - **Note:** This same subtraction is not needed for $f(x)$ since the optimal value is just 0.

n=128 is the highest number I was able to reasonably manage. The results for n=[2, 4, 8, 16, 32, 64, 128] for all the methods are in the table.

Note: In my python code, I have limited the dimensions to 64 so that it doesnt time out the autograder

dims	2	4	8	16	32	64	128
NM x	2.736e-07	3.181e-07	4.253e-08	3.461e+00	5.100e+00	7.670e+00	1.119e+01
SciPy NM x	1.151e-05	2.855e-05	1.156e+00	3.966e+00	5.547e+00	7.942e+00	1.129e+01
BFGS FD x	1.193e-05	1.376e-05	1.515e-05	1.543e-05	1.552e-05	1.547e-05	1.560e-05
BFGS AG x	1.951e-06	3.398e-09	3.844e-08	1.211e-07	1.923e-08	9.878e-08	2.930e-08
NM $f(x)$	2.724e-14	4.136e-14	6.693e-14	1.115e+01	2.556e+01	5.868e+01	1.255e+02
SciPy NM $f(x)$	3.686e-10	5.808e-10	6.150e-01	1.487e+01	3.014e+01	6.231e+01	1.267e+02
BFGS FD $f(x)$	2.844e-11	4.675e-11	5.833e-11	6.099e-11	6.380e-11	6.791e-11	7.746e-11
BFGS AG $f(x)$	7.717e-13	2.497e-16	5.526e-15	3.731e-14	4.000e-15	9.603e-14	2.907e-13
NM fev	164	512	2183	13477	12312	11560	11789
SciPy NM fev	146	422	1600	3200	6400	12800	25600
BFGS FD fev	72	225	1029	1870	7359	26910	105780
BFGS AG fev	48	90	138	220	442	842	1646
NM conv	**True**	**True**	**True**	*False*	*False*	*False*	*False*
SciPy NM conv	**True**	**True**	*False*	*False*	*False*	*False*	*False*
BFGS FD conv	**True**	**True**	*False*	**True**	**True**	**True**	**True**
BFGS AG conv	**True**	**True**	**True**	**True**	**True**	**True**	**True**

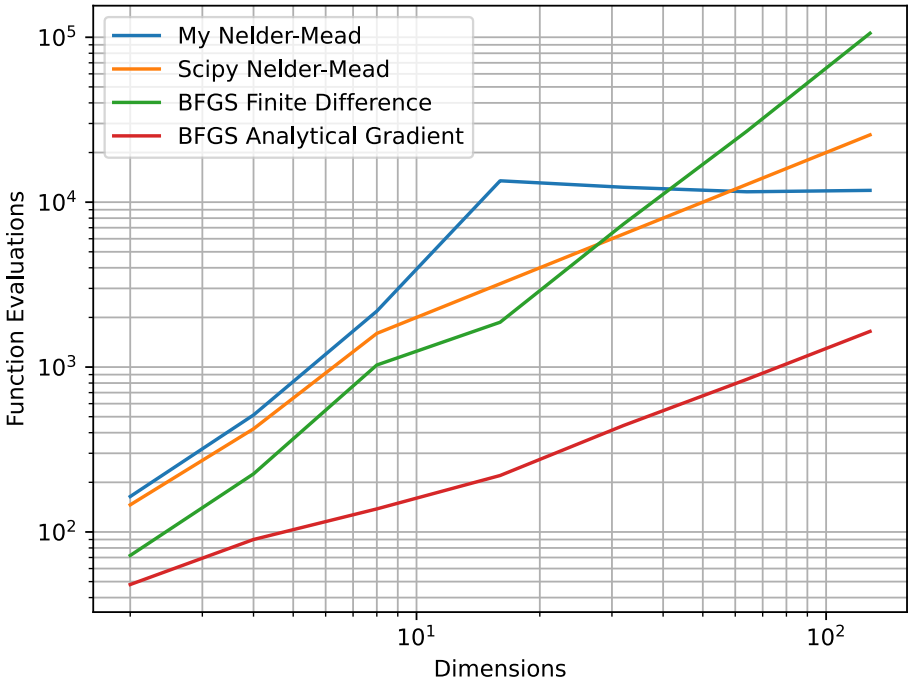


Figure 1: Function evaluations vs dimensions for all methods

Convergence

All algorithms manage to converge for n=[2, 4]. Nelder-Mead is here is limited by the number of fev allowed, with my implementation able to give precise results till n=8, while SciPy's implementation is limited to 200*n fevs. There is an outliar for BFGS with finite difference at n=8, where SciPy reports that it did not converge, but it still gives precise enough results (10^{-5} for x and 10^{-11} for $f(x)$). We can ignore this since the result is still precise enough. For n=8 and beyond, only the BFGS methods manage to converge.

Precision

For n=[2, 4] my Nelder-Mead outperforms SciPy's Nelder-Mead, which could just be due to a less strict convergence criteria used by SciPy. My Nelder-Mead is also consistent in terms of its precision ($10^{-7}/10^{-8}$ for x and 10^{-14} for $f(x)$) while it is able to converge without breaking the maximum iteration limit. Just for n=2, my Nelder-Mead has the most precise solution, followed by BFGS with analytical gradient, then BFGS with finite difference, and SciPy's Nelder-Mead performs the worst. For n >= 4, BFGS with analytical gradient outperforms the rest consistently with high precision. The precision of BFGS with finite difference is consistent (10^{-5} for x and 10^{-11} for $f(x)$) for all the dimensions tested, albeit lower than with analytical gradient, only becoming slightly less precise as the dimensions increase but still staying within an order of magnitude.

Compute Performance

BFGS with analytical gradient is at least 3x better than the Nelder-Meads in the case of n=[2, 4] while the Nelder-Meads still work. For higher n, we can only compare it to BFGS with finite difference, which it outperforms by a huge margin of 32x for n=64 and 62x for n=128. In cases where the Nelder-Meads are able to converge, BFGS with finite difference is still better than both the implementations by a factor os 2x. Both the Nelder-Meads perform similarly, the SciPy implementation outperforming my own by up to 20%. However we need to take note that my implementation gives a more precise optimum, which could account for the extra function evaluations as the convergence criteria for precision might be lower for SciPy's implementation.

Conclusion

Considering the trends discussed, for problems with a lot of dimensions, BFGS is the most viable solution as it works while performing the fewest function evaluations. Even if analytical gradients are not available, just using finite difference still performs well enough to be better than Nelder-Mead.