Numerical Optimization - Handin 4

This paper seeks to investigate the implementation of `BFGS` together with the algorithm for a line search which fulfills the strict Wolfe conditions. The algorithm is evaluated in terms of correctness, performance and is compared to previous implementations of other optimizers performed on the given objective functions.

1 Theory

1.1 The hunt for the negative eigenvalue

With $B_k = I$ we get the following update:

$$B_{k+1} = I + \frac{1}{y^\intercal s - s^\intercal s}(y-s)(y-s)^\intercal = QIQ^\intercal + \frac{1}{y^\intercal s - s^\intercal s}QDQ^\intercal = Q(I + \frac{D}{y^\intercal s - s^\intercal s})Q^\intercal = QKQ^\intercal$$
$$\text{where} \quad Q = \left[\ y-s \ \middle|\ \mathbf{o}_2 \ \middle|\ \mathbf{o}_3 \ \middle|\ \dots \ \middle|\ \mathbf{o}_n \ \right] \quad \text{for } \mathbf{o}_i \text{ such that } \ (y-s)^\intercal \mathbf{o}_i = \mathbf{o}_i^\intercal \mathbf{o}_j = 0 \ \ (i,j) \in \{2..n\}^2, \ i \neq j$$
$$\text{and} \quad D = \text{diag}(||y-s||^2, 0, 0, \dots, 0)$$
$$\implies K = \text{diag}(1 + \frac{||y-s||^2}{y^\intercal s - s^\intercal s}, 1, 1, \dots, 1)$$

As K contains the eigenvalues of B_{k+1}, we must find conditions on s and y such that the first eigenvalue is negative. We already have that the numerator is non-negative, so the denominator must be negative. This gives us the second necessary condition, $s^\intercal s > y^\intercal s$. We also want the fraction to evaluate to less than minus one for the sum to be negative:

$$\begin{aligned} & ||y-s||^2/(y^\intercal s - s^\intercal s) < -1 \\ \iff & ||y-s||^2 > s^\intercal s - y^\intercal s \ \text{ (using negative denominator)} \\ \iff & (y-s)^\intercal (y-s) = y^\intercal y - 2s^\intercal y + s^\intercal s > s^\intercal s - y^\intercal s \\ \iff & y^\intercal y > s^\intercal y \end{aligned}$$

To find somewhere this holds: For $x = [-1.7, 1.3]^\intercal$, $p = [0.7, 0.7]^\intercal$, $\alpha = 3.4$ we have $y^\intercal y = 11.471 > s^\intercal y = 8.069 < s^\intercal s = 11.329$. For $c_1 = 0.1$ and $c_2 = 0.9$ this step fulfills the Wolfe conditions, as illustrated in figure 1 too.

1.2 Super-linear convergence

We wish to show that the gradient norm on a quadractic function, f, converges to zero Q-superlinearly when using the step; $x_{k+1} = x_k - B_k^{-1}\nabla f(x_k)$. First we show that the new gradient corresponds to a linear transform by multiplying the previous gradient with a matrix R_k and that $\lim_{k\to\infty} R_k = 0$

$$\begin{aligned} \nabla f(x_{k+1}) = \nabla f(x_k + p_k) &= Q(x_k - B_k^{-1}\nabla f(x_k)) + g \\ &= Qx_k + g + QB_k^{-1}\nabla f(x_k) \\ &= I\nabla f(x_k) - QB_k^{-1}\nabla f(x_k) \\ &= (I - QB_k^{-1})\nabla f(x_k) \\ &= R_k \nabla f(x_k) \qquad ; \qquad R_k := (I - QB_k^{-1}) \end{aligned}$$
$$B_k \xrightarrow{k\to\infty} Q \implies R_k \xrightarrow{k\to\infty} I - QQ^{-1} = \mathbf{0}$$

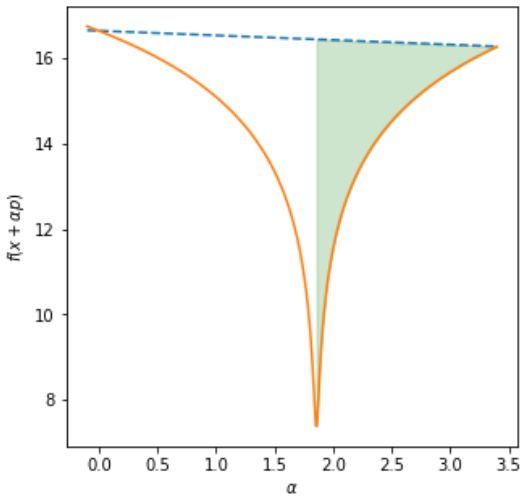

Figure 1: Illustration of the function values of f_3 along direction p, going α along p. The c_1 and c_2 conditions are plotted as well.

Now we look at the ratio between the norm of the next gradient and the current gradient and show that this ratio is upper bounded by a sequence which goes to zero as k goes to infinity.

$$\lambda_k^{max} \overset{\text{def}}{=} \max |\text{eigenvalues}(R_k)|$$

$$\frac{||\nabla f(x_{k+1})||}{||\nabla f(x_k)||} = \frac{||R_k \nabla f(x_k)||}{||\nabla f(x_k)||} \leq \lambda_k^{max} \frac{||\nabla f(x_k)||}{||\nabla f(x_k)||} = \lambda_k^{max}$$

$$R_k \xrightarrow{k\to\infty} \mathbf{0} \implies \lambda_k^{max} \xrightarrow{k\to\infty} 0$$

According to the definition of Q-superlinear convergence in the notes the elements of the sequence used to upper bound the ratio must be in $(0, 1)$ but we can simply start at a k where all following elements in the sequence will be in $(0, 1)$ since the sequence goes to zero. Note that we here used the eigenvalues of R_k in the bound which perhaps requires R_k to be symmetric. More generally the bound could be made with the largest singular value with the same result and without this assumption on R_k.

2 Correctness

Refer to figure 2 for the following discussion. Each column has the same settings of c_1 and c_2, only a_{init} varies.
Let's take a look at the left-most column first. c_1 and c_2 have been chosen to give almost no restrictions, basically all that is required is that $f(x_k) > f(x_{k+1})$. For a big initial step length 100 the optimum is overshot, and we return a point almost at the complete opposite side of the valley. Any further and the function value would've been too high, so this is where we land. In the lower graph of the first column we immediately accept the short step 0.01, but make very little progress. The results are as expected.
Now for the middle column: Here we have a moderate value for c_1, but still a quite relaxed c_2. c_1 limits how far we can go, and as can be seen for both step lengths, we undershoot the optimum, as it gets excluded by our choice of c_1. For the short step length, we note that the lower choice of c_2 results in more required progress, when comparing to $c_2 = 0.99$.
Now for the right-most column; We have a low value for c_1, so we could theoretically step as far as in the leftmost column. But c_2 is set the lowest yet - only a small region around a local optimum is acceptable. Because we take the intersection of the requirements by c_1 and c_2, a_{init} does not play a large role, and the plots end up looking similar.

We argue for correctness of our algorithm by looking at the middle and bottom segment of figure 4, where we see convergence plots for both `BFGS` and `scipy BFGS` for different dimensions. We see that our implementation of

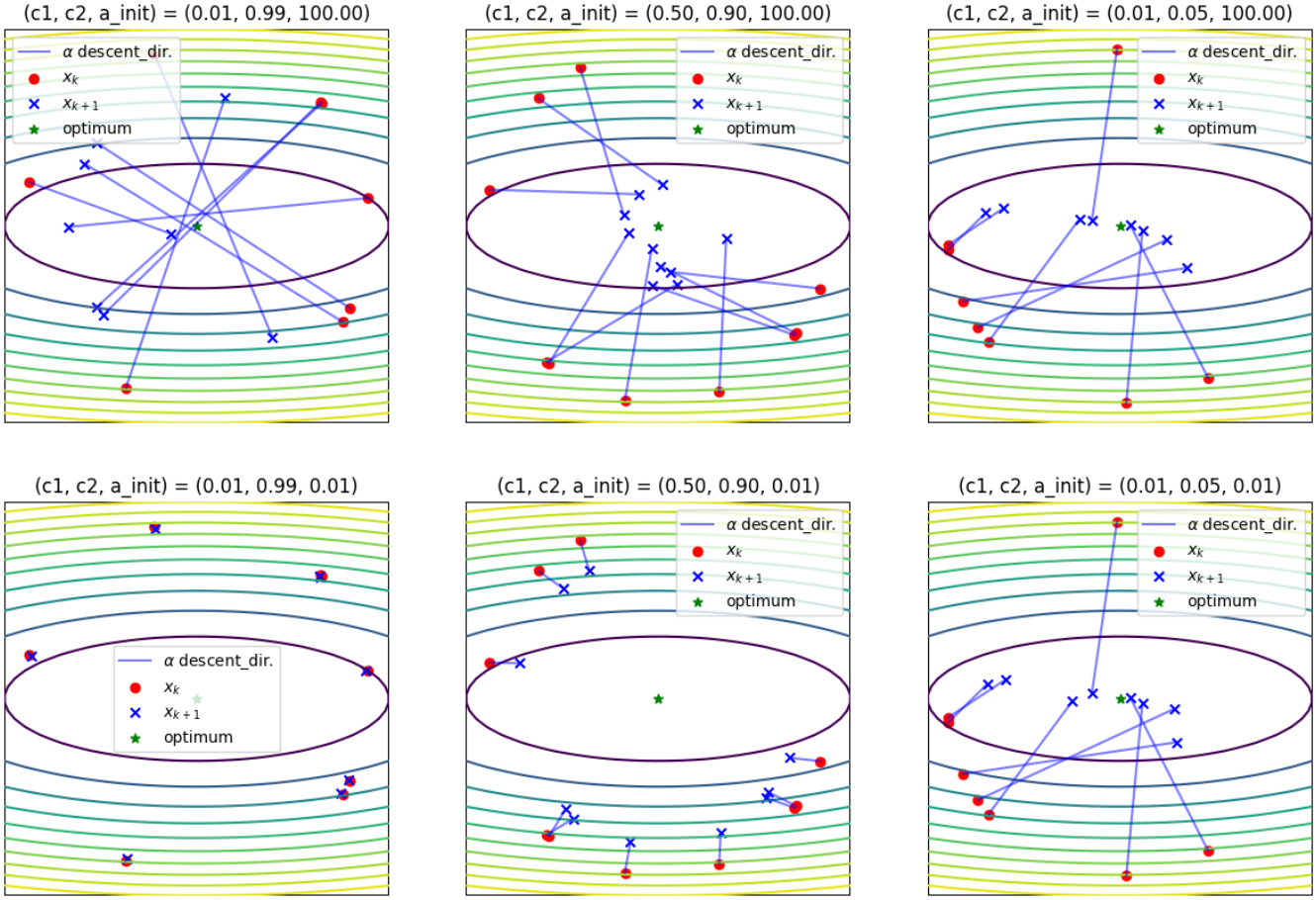

Figure 2: Step lengths returned by `wolfe-strict` on $f_1(\cdot, \alpha = 10)$, with different settings of c_1, c_2 and a_{init}.

BFGS is very similar and follows the same trends as scipy BFGS. Though they are not equally fast, as can be seen on the x-axis. Here, we see that our implementation of BFGS reaches convergence in fewer steps for all of the four functions, suggesting that our implementation is faster. Though the speed of convergence is not the same, the plot should still suffice to show that our implementation is correct, and perhaps even better than scipy BFGS.

3 Results

Evaluation of the performance of BFGS on test functions

To evaluate the performance of BFGS on the test functions, have a look at figure 4 and the previous discussion on comparison with scipy BFGS. Suprisingly, our implementation performs better than scipy BFGS in terms of steps and time taken until convergence, however, by looking at figure 3 we see that the other optimizers from previous assignments are still superior in terms of steps taken until convergence. BFGS always takes more steps than newton, but runs faster (seconds) when the function we optimize is a higher order than 2, which might be because we don't need the exact newton step, but a fast approximation suffices. Approx Newton (CG) is generally the strongest optimizer, but can have high variance in the time it takes to finish. The only drawback is that it requires access to the hessian, which BFGS circumvents elegantly without losing much performance.

Comparison to Newton original and approx Newton

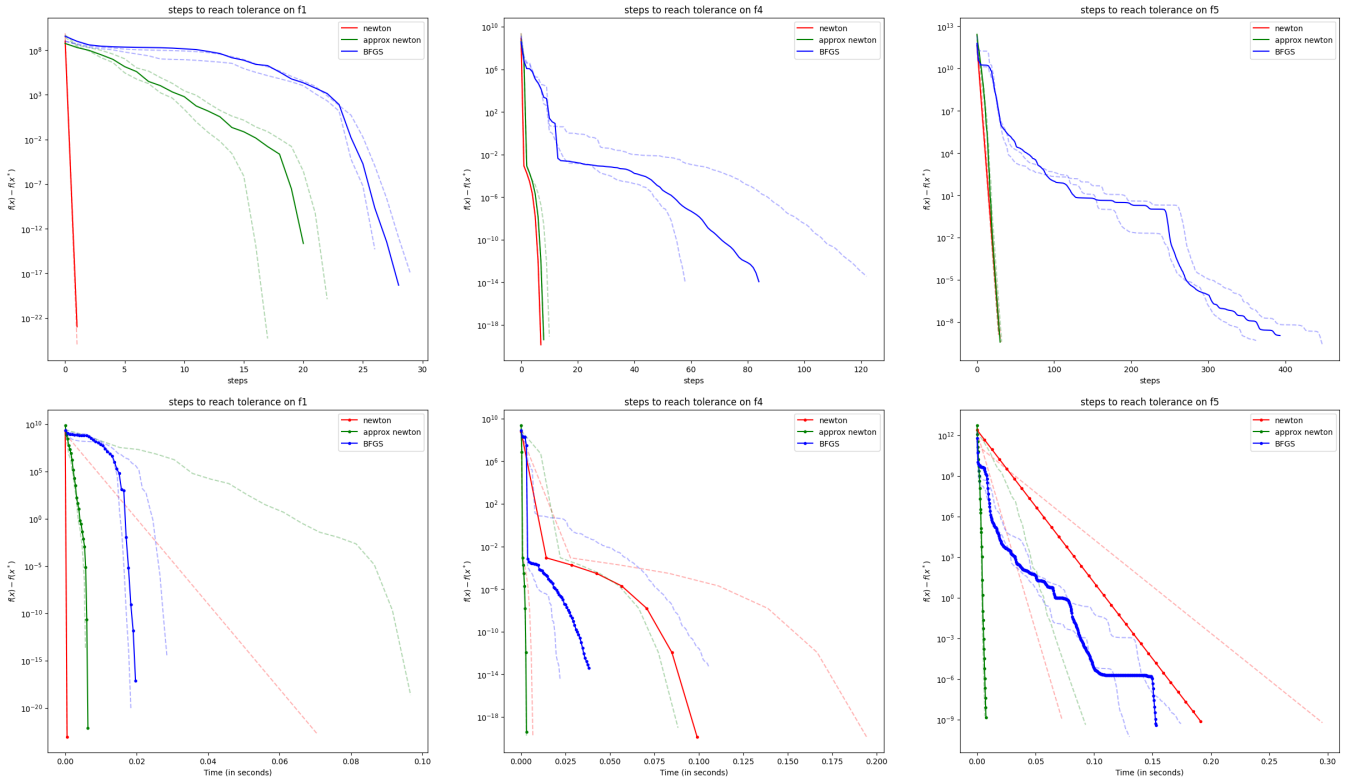

Figure 3: Comparion of BFGS, Original Newton and Approx Newton measured in steps until convergence (upper) and time until convergence (lower) for $d = 20$, $c_1 = 0.01$ and $c_2 = 0.9$ (Plotted for slowest and fastest runs (Transparent lines) and the median run (Thick line)).

In figure 3 we see a comparison of BFGS against original newton and approx newton, where in the upper segment of the figure, we see them against each other in terms of steps taken until convergence, and in the lower segment we see them against teach other in terms of time taken until convergence. We see that in terms of steps taken, BFGS

is not competitive on any of the functions. In terms of time taken until convergence, we see that it is faster than `newton` on f_1 and f_5 suggesting that if time is a factor and computational resources is not, then `BFGS` is better to use than `original newton`. However, it cannot compete with `approx newton` in any of the cases. Do note, that we did not do a comparison on functions f_2 and f_3 as the requested dimensionality was preferred to be $>= 20$, and the banana function does not support this. Likewise, the hessian matrix had to be positive (semi-)definite for `approx newton` to work, and only f_1, f_4 and f_5 were included in the previous assignment hence we also only compare to these, which is the reasoning for our choice of functions we compare with.

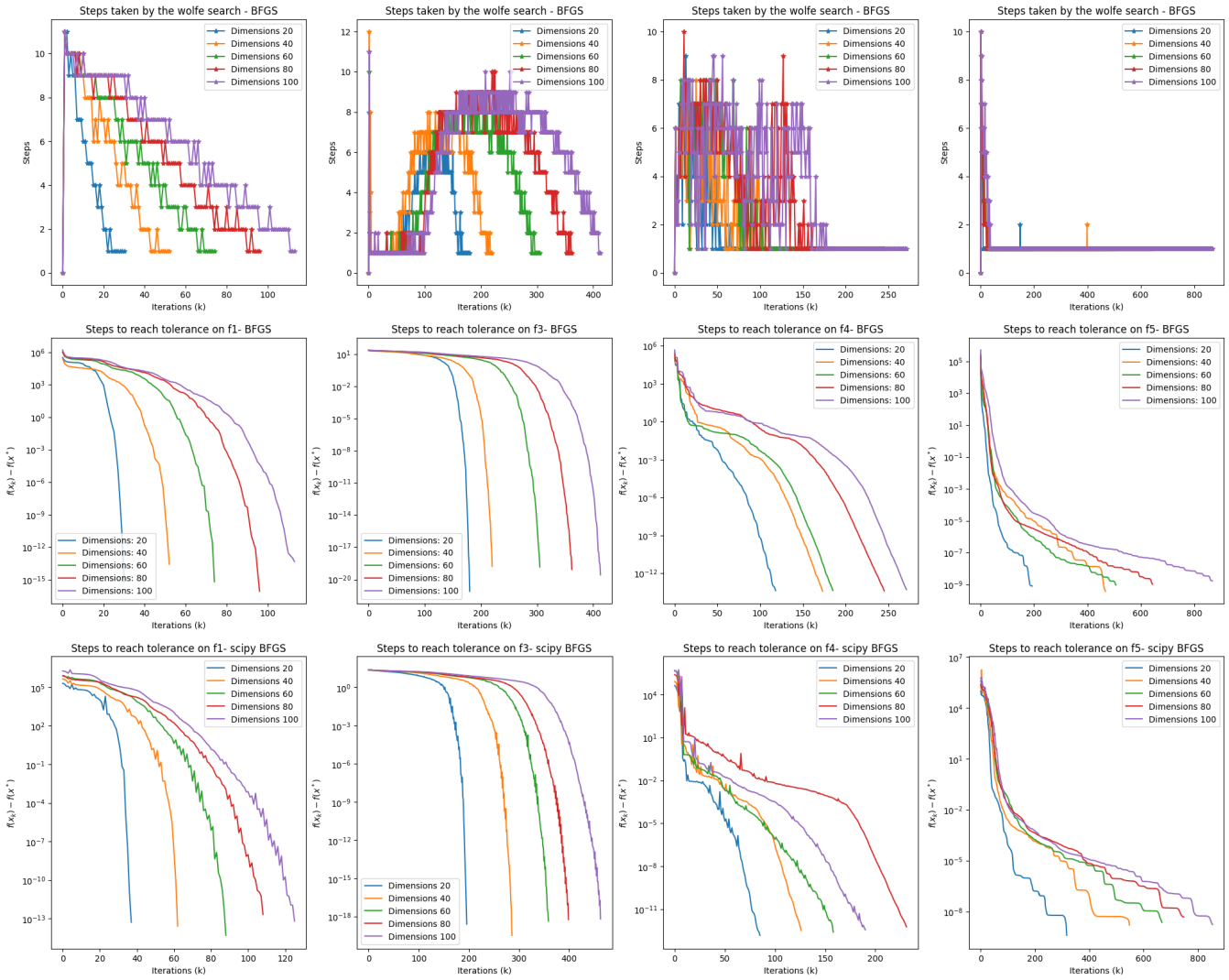

Figure 4: This figure contains three different segments. The top consists of the steps taken by the wolfe search for a different number of dimensions. The middle consists of convergence plots for different dimensions for our implementation of `BFGS`. The bottom consists of convergence plots for different dimensions of `scipy BFGS`.