Evaluating performance of nonlinear optimization methods

Christian Segercrantz, Tuukka Mattlar
Aalto University School of Science, Department of Mathematics and Systems Analysis
{*christian.segercrantz, tuukka.mattlar*}*@aalto.fi*

1 Background

This project evaluates different nonlinear optimization algorithms and the performance of these with respect to the specific characteristics of each method. The presented solutions were primarily obtained using a personal computer running Windows 10 build 19043.1319. The central processing unit (CPU) is a Intel Core i7-6700K, which has 4 cores with 2 threads each, overclocked to a clock speed at 4.00GHz. The computer has 16GB of rapid access memory (RAM) clocked at 2667 MHz. In Section 2.4, an alternative run is computed on the Aalto CS's JupyterLab server. The detailed information of this server is not available but it is seemingly less powerful as the PC used for the computations primarily.

The four methods discussed in this project are the Gradient method with (the heavy-ball method) and without momentum, Newton's method and Broyden-Fletcher-Goldfarb-Shanno (BFGS) method. All methods are implemented and studied with either a exact or inexact line search methods where the bisection method is used for exact cases and Armijo's rule for inexact ones. Finally, the methods are evaluated using performance profiles.

2 Applications

2.1 Task 1: Gradient method with and without momentum

The gradient methods are studied with the following function 1 with an initial point $x_0 = \begin{bmatrix}-10 & 10\end{bmatrix}^\top$.

$$f_1(x_1, x_2) = 2(0.5x_1^2 + 4x_2^2) - 0.5(x_1x_2) \tag{1}$$

To find out if function 1 is convex, we study the eigenvalues of the Hessian for the function. As $H(f(x_1, x_2)) = \begin{bmatrix} 2 & -0.5 \\ -0.5 & 16 \end{bmatrix}$, the eigenvalues obtained are $\lambda_1 \approx 16.0, \lambda_2 \approx 2.0$, meaning that the function is convex under Theorem 12 on week 3. This indicates that any found local minima is also a global minima as per Theorem 2 from lecture 4.

The gradient decent method uses first-order derivative information in order to find the optimal point. However, the gradient, the first-order derivative information, can sometimes behave erratically in the function making the algorithm bounce back and fourth. This can ultimately even lead to divergence. In order to prevent this jumpy behaviour caused by, for example, flat spots or other irregularities we add some history to the step-direction. This can be seen as an analogy from physics, we give the algorithm momentum to keep moving in the direction it has already been going. The momentum, i.e. the history information, is obtained by adding some of the previous direction to the new direction. By doing so, we decrease the risk of having the method take to steps that do not take us closer to the optimal point. Thus instead of making the direction $d_k = -\nabla f(x_k)$ we make it $d_k = -(1-w)\nabla f(x_k) + wd_{k-1}$ where $w \in [0, 1)$ is the weight hyperparameter.

Figure 1 displays how many iterations are needed for the convergence of the function 1 with respect to the weight w.

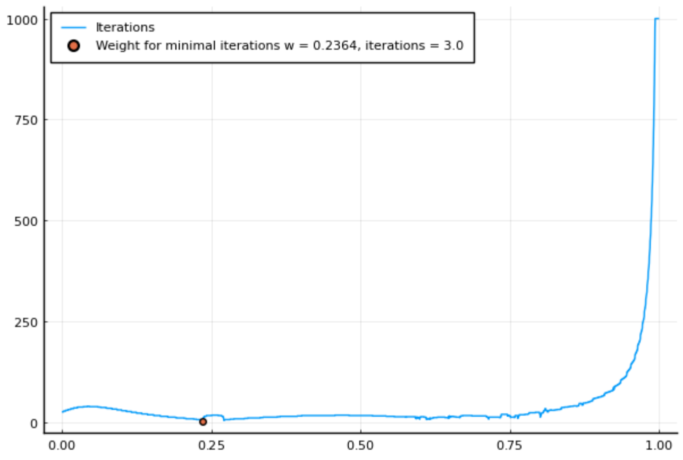
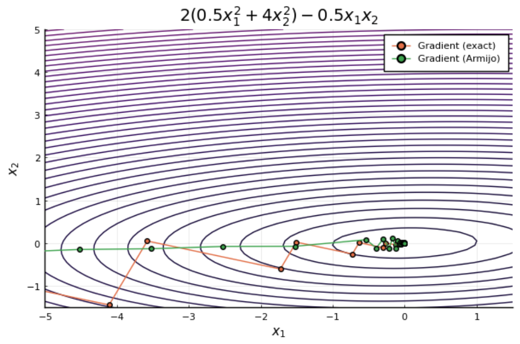


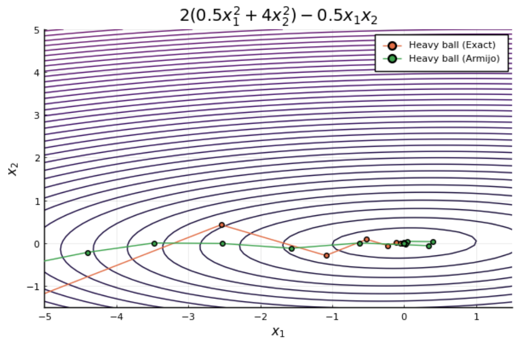
Figure 1: Iterations needed for the algorithm to converge as a function of the weight on Function 1. The first minima are marked with a orange dot.

Figure 1 illustrates the optimal weight between d_{k-1} and the computed gradient, where 0 is basically a gradient method and 1 constantly follows the d_k. The result shows how there is little difference between values within the range of approx 0.2 and 0.75. However, as $w \to 1$ the number of iterations increases rapidly as the method no longer gets new information on each iteration. The most optimal value is ≈ 0.24 which only requires 3 iterations for finding the optimal solution. The reason behind why a lower weight, or in other words larger momentum, works better in our case is because of the non-symmetry of the function. The Figures 2 show that the decent of the function value for the two variables differ. This means that the momentum will keep the algorithm to continue down the slope.

The convergence of the two methods with both inexact and exact cases are presented in Figures 2a and 2b.



(a) The gradient method.



(b) The heavy ball method

Figure 2: Visual presentation of the convergence of the gradient method with and without momentum on Equation 1.

The figure 2 shows that all methods for Function 1 converge to the same point that is a local optimum but also by convexity, the global optimum. The pure gradient method seem to 'zig-zag' more along the way whereas the heavy-ball follows a more direct path to the optima. The gradient method seems to require plenty of iterations close to optima whereas the heavy-ball method achieves the optima with less iterations closer to optima. One more observation is that the inexact heavy ball first misses the optima before turning around and coming back for it.

Table 1: Key performance indicators for the methods on Function 1.

	$f_1(\bar{x})$	x value	Time	Iterations
Gradient (exact)	0	$\begin{bmatrix}0 & 0\end{bmatrix}^\top$	199μs	25
Gradient (Armijo)	0	$\begin{bmatrix}0 & 0\end{bmatrix}^\top$	153μs	50
Gradient with momentum (exact)	0	$\begin{bmatrix}0 & 0\end{bmatrix}^\top$	134μs	16
Gradient with momentum (Armijo)	0	$\begin{bmatrix}0 & 0\end{bmatrix}^\top$	98μs	37

The most efficient method in terms of clock time is the heavy-ball Armijo. Moreover, one can notice that the heavy-ball method is generally faster than the gradient method, and the Armijio-versions are faster than those of exact version. In terms of iterations, the exact versions result in approximately half of the iterations of those by the inexact armijios'. The heavy-ball method requires generally less iterations than the gradient method.

The reasoning behind the differences can be divided in two, inexact vs exact and the difference between Gradient and heavy-ball gradient methods. First of all, as the inexact Armijio versions do not optimize the step size on each iteration, the result is less computational time but more iterations. The reason for the inexact heavy-ball method first missing the optima is also related to this, since without optimized step size, the optima can be missed.

The heavy-ball method successfully reduces zig-zagging in comparison to the gradient method, as can quite clearly be seen in figures 2a and 2b. This can also help in decreasing the number of iterations as happened in our case. This results from the fact that the previous direction d_{k-1} is used to stabilize the path. However, it is likely that such balancing could in some cases cause harm if the initial direction d_1 was not facing approximately towards the optima. In fact, figure 1 indicates that the method first misses the optimum in the case of Armijio-version and only then converges back closer to the optima.

2.2 Task 2: Newton's and BFGS method

The Newton's method and its variant BFGS methods are studied with the following exponential function 2 with the initial point $x_0 = \begin{bmatrix} -2.5 & -3.5 \end{bmatrix}^\top$

$$f_2(x_1, x_2) = e^{x_1+2x_2-0.1} + e^{x_1-2x_2-0.1} + e^{-x_1-0.2} \tag{2}$$

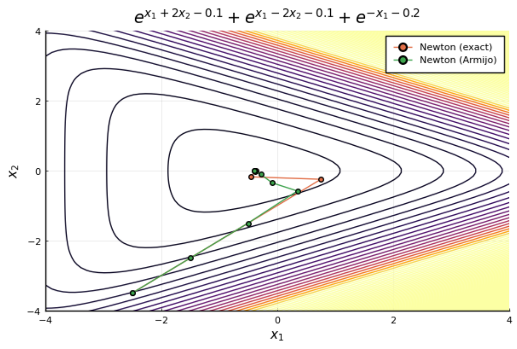
The Hessian of Function 2 is

$$\begin{bmatrix} e^{x_1-2x_2-0.1} + e^{x_1+2x_2-0.1} + e^{-x_1-0.2} & 2e^{x_1+2x_2-0.1} - 2e^{x_1-2x_2-0.1} \\ 2e^{x_1+2x_2-0.1} - 2e^{x_1-2x_2-0.1} & 4e^{x_1-2x_2-0.1} + 4e^{x_1+2x_2-0.1} \end{bmatrix}, \tag{3}$$

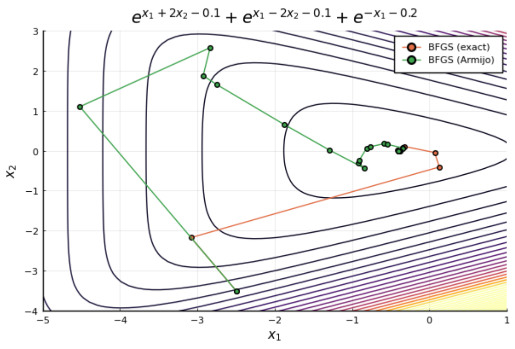
which is positive semi-definite for all value and the function convex.

The Newton's method utilizes both the gradient of the point of interest as well as the second order information obtained by calculating the hessian of each point of interest. This provides the method with more profound knowledge on the behaviour of the function, and thus the direction. The BFGS method is a quasi-newton method meaning that it utilizes approximations to decrease the computational requirements while using the idea of the Newton's method. In BFGS, the inverse of the Hessian is approximated with local curvature information. This means that no there is no need to perform the computationally heavy task of computing the Hessian nor its inverse during the iterations.

The convergence profiles of the two methods for function 2 are presented in figures 3.



(a) Newton's method.



(b) The BFGS method

Figure 3: Visual presentation of the convergence of Newton's and the BFGS on Equation 2.

The Figures 3a and 3b show that all functions converge to the same point. Moreover, as the function 2 is by exponential nature a convex function, the results is also a global oprimum. The Newton's method quite neatly approaches the optima whereas the BFGS method takes rather complex paths along the way. Especially the inexact version of BFGS gets relatively far away from the optima before eventually finding it.

Table 2: Key performance indicators for the methods on Function 2.

	$f_2(\bar{x})$	x value	Time	Iterations
Newton's (exact)	2.43	$\begin{bmatrix}-0.4 & 0.0\end{bmatrix}^\top$	48μs	4
Newton's (Armijo)	2.43	$\begin{bmatrix}-0.4 & 0.0\end{bmatrix}^\top$	28μs	7
BFGS (exact)	2.43	$\begin{bmatrix}-0.4 & 0.0\end{bmatrix}^\top$	64μs	6
BFGS (Armijo)	2.43	$\begin{bmatrix}-0.4 & 0.0\end{bmatrix}^\top$	75μs	19

In terms of performance, the table 2 indicates that in terms of iterations, the newton method is generally faster than the BFGS method. The required computational times taken are also interestingly less for Newton's methods. The inexact BFGS seems to provide no benefit in any way.

The results are rather surprising since the BFGS method is supposed to outperform Newton's method due to the approximation that decreases the need for computation. However, as our case only limits the variable dimensions in the length of two in both dimensions, the computational power required also decreases making the difference relatively unnoticeable. Moreover, there is potential, that the system used has more efficient base algorithms for the hessian and inverse operations more heavily used by the Newton's method. This is likely to be another explanation together with the rather limited dimension lengths.

2.3 Task 3: Exact variants of the four methods

The exact variants of each method are studied with the following polynomial Function 4 with the initial point $x_0 = \begin{bmatrix}-0.5 & -2\end{bmatrix}^\top$. In this section, all the previously introduced methods are utilized using exact line search. For the heavy-ball method, a weight of 0.2 is used

$$f_3(x_1, x_2) = (x_1^2 + x_2 - 10)^2 + (x_1 + x_2^2 - 15)^2 \tag{4}$$

The Hessian of Function 4 is

$$\begin{bmatrix} 4(x_1^2 + x_2 - 10) + 8x_1^2 + 2 & 4x_1 + 4x_2 \\ 4x_1 + 4x_2 & 4(x_1 + x_2^2 - 15) + 8x_2^2 + 2 \end{bmatrix}, \tag{5}$$

which we can easily see that is not positive semi-definite for all values of x and y and thus not convex. We can confirm this from Figure 4 as we can see the multiple local minimas. The local minimas all have the same objective value, which means that all represent a global optima as well.

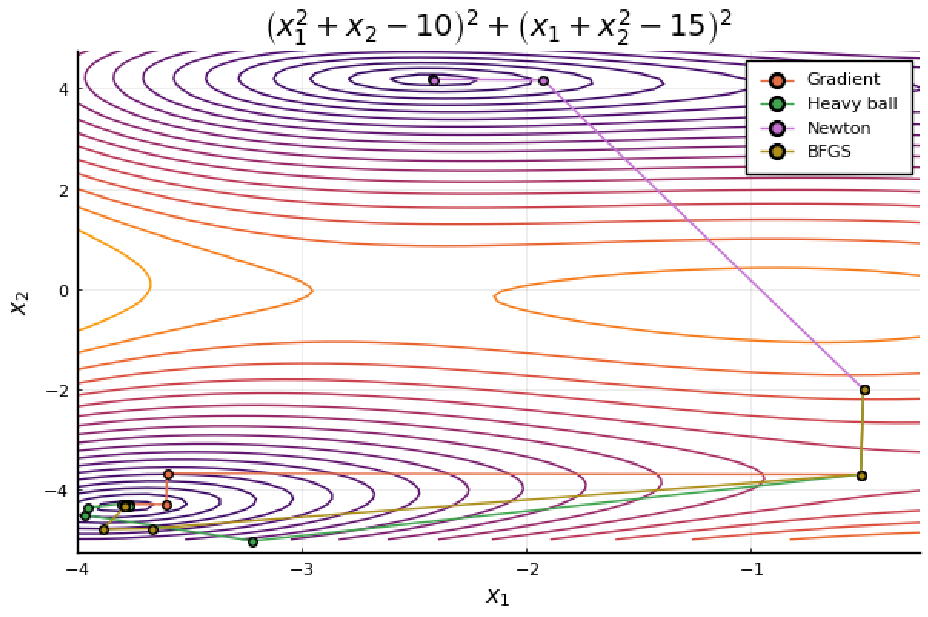


Figure 4: Visual convergence presentation of the exact variants of the four methods following the function 4.

Figure 4 shows that not all methods converge to the same point. Newton's method converges to a different point than the other points. However, both points accessed are local optima. What is interesting is that the path to the optima found by the Newton's method crosses a local maximum.

Table 3: Key performance indicators for the methods on Function 4.

	$f_3(\bar{x})$	x value	Time	Iterations
Gradient	0	$\begin{bmatrix}-3.79 & -4.33\end{bmatrix}^\top$	91μs	11
Gradient with momentum	0	$\begin{bmatrix}-3.79 & -4.33\end{bmatrix}^\top$	158μs	18
Newton's	0	$\begin{bmatrix}-2.41 & 4.17\end{bmatrix}^\top$	35μs	3
BFGS	0	$\begin{bmatrix}-3.79 & -4.33\end{bmatrix}^\top$	60μs	6

Table 3 shows how differently the methods perform. The newton's method is clearly the fastest and most efficient in terms of iterations. Heavy-ball gradient method is clearly the slowest and requires most iterations.

The results are quite interesting as the Newton's method should in theory be the slowest as it computes a hessian during each iteration. Moreover, as the BFGS method should be a faster version of the Newton's method, given the approximation of the hessian, the average time per iteration of Newton's and BFGS methods is surprisingly similar. This is likely due to the fairly small and simple $\mathbb{R}^{2x2}$ Hessians in this case. Howeve, one can notice that the average computational time pro iteration is the highest for Newton's method which is aligned with the theory and is likely, that for larger matrices would illustrate the limitation of Newton's method.

The difference in the Gradient and the heavy-ball gradient methods is interesting as in the first task the heavy-ball method helped in lowering the number of iterations and computational time. In fact, Figure 4 shows how the path of both gradient method as well as the heavy-ball gradient methods is first similar but after the first step, the heavy-ball method selects a direction that is less different to the d_1 as the d chosen by the Gradient method. Later the heavy-ball method misses the optima and has to turn around for a new approach increasing the number of required iterations.

Figure 5 displays, similarly to Figure 1, the optimal weights of the gradient method with momentum. We can again see that a smaller weight outperforms iteration wise larger weights. From earlier plots we can again see that the functions have steeper decent on one axis than the other, hence why a greater "momentum" performs better.

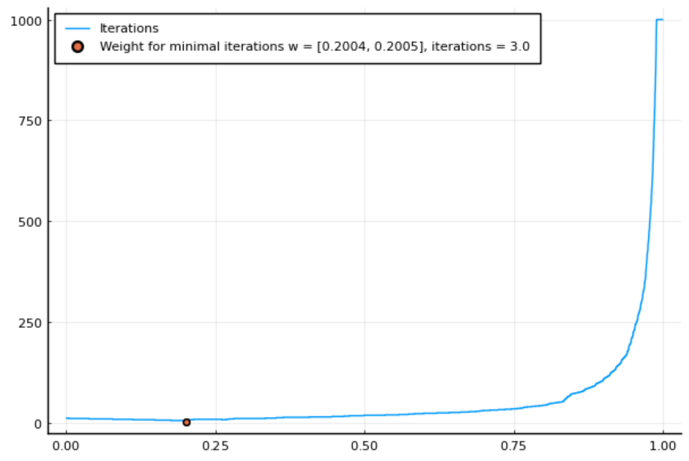


Figure 5: Iterations needed for the algorithm to converge as a function of the weight on Function 4. The first minima are marked with a orange dot.

2.4 Task 4: Performance profiles

In order to evaluate the different methods we will be using performance profiles. Each of the four methods presented above with both line search variants were ran 1000 times on the following function 6, with two differently conditioned matrices A.

$$f_4(x_1, x_2) = \frac{1}{2}x^\top Ax - b^\top x, \quad x \in \mathbb{R}^{100}, \tag{6}$$

where A is a positive semi-definite matrix and b a vector. Both A and b were randomly generated for each iteration, however the randomness was controlled with seed. Moreover, two conditions were used for the random generation of A: one with moderate condition numbers and one with larger condition numbers. This affects the range in eigenvalues that further affects the stability of results for any matrix operations. For each iteration, the time taken to minimize Function 6 was measured. In the case that no solution was found, or a non-optimal solution was found, was the time set to infinity.

In order to compute the performance ratios we find the t_{min} for each instance of Function 6. We then divide all times t of that instance with t_{min} to get the ratio τ. Those iterations that have not converged are removed. The ratios τ are then sorted in increasing order. We then sum all the ratios that are less than or equal to a specific τ to get the performance profile ρ_M for each method. Lastly, ρ_M is plotted as a function of τ in order to obtain the performance profile plots.

The following figures indicate the performance of each method. Methods that reach high P-values with lower τ-values outperform those reaching similar values later.

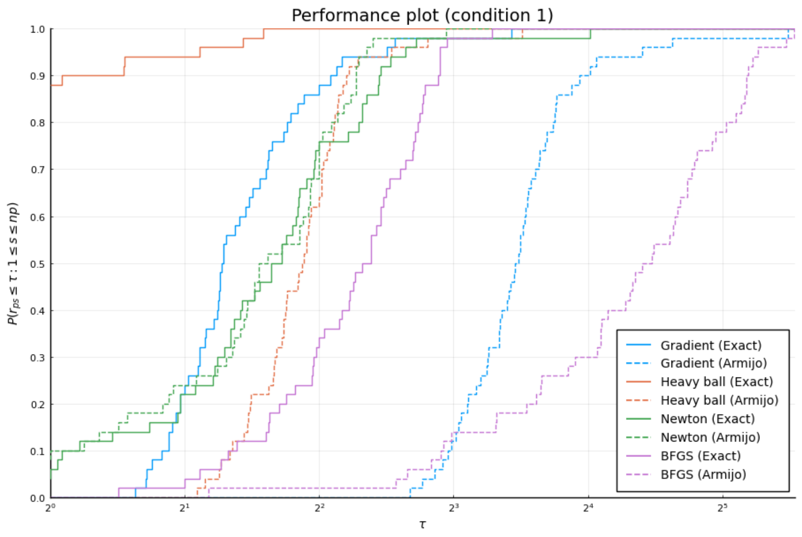


Figure 6: Performance profiles of the algorithms discussed using moderate condition numbers for matrices A.

Figure 6 indicate that for the first condition, the heavy-ball variant of the gradient method managed to perform best in most cases. We can see that in general, the gradient based methods outperformed the second-order derivative methods. This is likely due to the conditioning. Since conditioning is calculated as $\kappa = \frac{max_{1=1,\ldots,n}\lambda_i}{min_{1=1,\ldots,n}\lambda_i}$, the larger the conditioning the more the level curves of the function are stretched out. Thus, the gradient gives good information for the direction to quickly converge to the optima when the conditioning is small. As the conditioning grows does the gradient based methods require more iterations and display a lot of "zig-zagin".

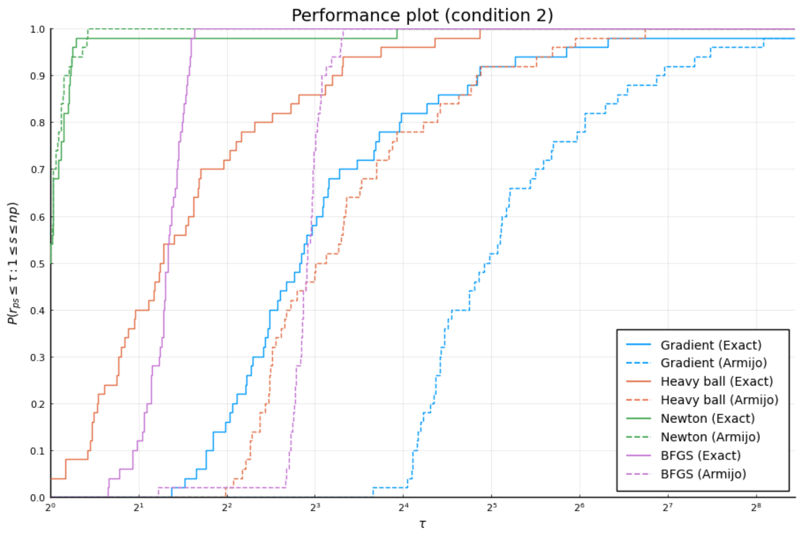


Figure 7: Performance profiles of the algorithms discussed using larger condition numbers for matrices A.

The figure 7, however, shows a whole different distribution. It seems like the exact BFGS-method was nearly always the fastest as it neatly follows the y-axis almost until the 100% line.

Viewing the larger picture we can see that this time the second-order derivative functions outperforms the solely gradient based ones. Since the conditioning is larger, i.e. the level curves more stretched out, the second-order derivative gives information needed in order to the quickly find the optima. In fact, based on exercise 5.1 and some additional algebra, we know that using Newton's method will converge in one iteration for Function 6 each time. Hence, it's a matter of how quickly the computer can calculate the inverse Hessian versus how good information the gradient provides.

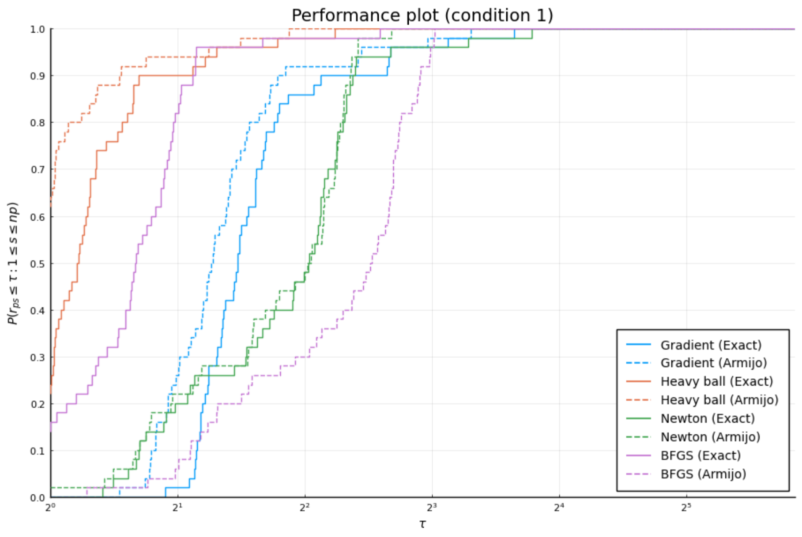


Figure 8: Performance profiles of the algorithms discussed using moderate condition numbers for matrices A on the alternative setup discussed in Section 1.

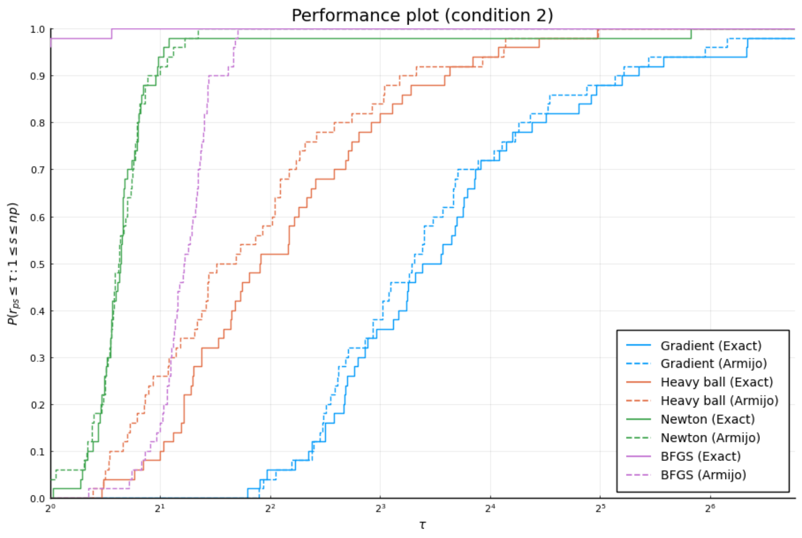


Figure 9: Performance profiles of the algorithms discussed using larger condition numbers for matrices A on the alternative setup discussed in Section 1.

Figures 8 and 9 display the performance profiles on the alternative setup, the Aalto CS's JupyterLab server. We can see that the performance profiles of have clear differences. The most significant one seems to be that the methods using the exact line search clearly outperform others on the local setup. We can also see that Newton's method outperforms the other methods on the larger condition numbers on the local setup while the BFGS method does this on the server. This follows from Newton's method, as it is in theory the computationally most expensive method. This may be caused by differences in the setups hardware and software, how well linear algebra is calculated and how well the central processing units are utilized.

3 Discussion and conclusions

This project has studied four different algorithms, the gradient method, the gradient method with momentum, Newton's method and Broyden-Fletcher-Goldfarb-Shanno (BFGS) method as well as two variants of line search for this: exact bisection and Armijo's rule based line search. The algorithms were first build and tested, and then compared to each other using Performance profile plots.

We could see that both the gradient based methods were very sensitive to the conditioning of the function applied on. The smaller the conditioning, the worse will the gradient based methods work. Using momentum, especially with a tuned weight hyperparameter, one can counteract the impact of the conditioning on the algorithm.

The second-order derivative based algorithms were faster when applied on functions with smaller conditioning. However, when the condition was larger, the gradient based methods were faster in general than Newton's and BFGS method. This is a result of the heavy computations performed by the methods, calculating or approximating the inverse Hessian. The hessian based methods were faster then the gradient based methods when the latter needed a lot of iterations to converge, i.e. when the conditioning was small.

On functions with multiple optimas we noticed that different methods can converge to different points.

We noticed that software and hardware differences of setups can have a large impact on the effectiveness of algorithms. When a system is able to make good use of linear algebra, the resources needed for second-order derivative computations are reduced, thus making these methods more effective relatively. However, since the computed functions were at their largest $\mathbb{R}^{100\times 100}$ we could see a different result on problems of larger magnitude. Further tests should be performed to conclude how the methods work on very large problems.

Additionally, the choice of setup had a large impact on the effectiveness of the line search methods speed. The server's, which the authors believe to be less powerful based on empirical evidence, gap between the line search algorithms. Since the time needed to calculate the exact line search algorithm probably is more hardware dependent than the inexact, is the result that they are closer in performance for the server than the local setup. In order to draw more conclusions, the authors would need to investigate the true differences in the systems and do additional tests and measurements.