GRADIENT DESCENT VS NEWTON'S METHOD

THE ALGORITHMIC SHOWDOWN

Reid Falconer[a]

[a]*Barcelona Graduate School of Economics, Barcelona, Spain*

Keywords: Line Search, Gradient Descent, Newton's Method

1. Outline

Line search methods are used widely for numerical optimisation. This project seeks to determine how two line search methods, namely gradient descent and Newton's method, perform relative to each other and to further assess the differences in the execution of first order and second order optimisation algorithms.

Let $\boldsymbol{x} \in \mathbb{R}^n$ and consider the function

$$f(\boldsymbol{x}) = \frac{1}{2}(\boldsymbol{x} - \boldsymbol{m})^\top A(\boldsymbol{x} - \boldsymbol{m}) - \sum_{i=1}^{n} \log\left(x_i^2\right) \tag{1}$$

where $\boldsymbol{m} \in \mathbb{R}^{\mathbf{n}}$ is a fixed vector and $A \in \mathbb{S}_{t+}^n$ is a fixed positive definite matrix.

We will primarily focus on the two-dimensional case ($n = 2$). Where $\boldsymbol{m}$ is fixed at $\boldsymbol{m} = (0.5, 0)$ and we will study the convergence of the two algorithmic methods mentioned above depending on $\rho \in (-1, 1)$ in

$$A = \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}$$

More specifically, in this analysis we will implement the *gradient descent* and *newton's method* for the problem of minimising function (1). Furthermore, we will also explore

Email address: reid.falconer@bracelonagse.eu (Reid Falconer)

how the performance of the two algorithms scales with n equal to 50, 100, 200, 500 and 1000.

1.1. Gradient Descent

Gradient descent is a simple method for numerical optimisation, similar to some of the methods you can use with the 'optim()' function in R, but here we will explicitly program it. It is an alternative to analytical solutions that cannot always be employed. For example, data might be too big, or matrix inversion may be too costly.

More precisely, gradient descent is an iterative method that starts with an initial guess ($\boldsymbol{x}^{(0)}$) and updates the parameters in each iteration by the amount of the gradient multiplied by a learning rate, α_t

$$\boldsymbol{x}^{(t+1)} = \boldsymbol{x}^{(t)} - \alpha_t \nabla f\left(\boldsymbol{x}^{(t)}\right) \tag{2}$$

where $\boldsymbol{x}$ is a vector of parameters and $\bigtriangledown f(\boldsymbol{x}^{(t)})$ is the gradient of the objective function $f(\boldsymbol{x})$, that is, a derivative of our objective function with respect to the parameters.

Additionally, given that the learning rate can substantially modify the behaviour of the algorithm (if it is too big the algorithm may not converge), we also implement basic backtracking, where we start with $\alpha_t = 1$ and check whether $f\left(\boldsymbol{x}^{(t+1)}\right) < f\left(\boldsymbol{x}^{(t)} - \alpha_t \nabla f\left(\boldsymbol{x}^{(t)}\right)\right)$. If not, $\alpha_t \leftarrow \alpha_t/2$ and repeat.

1.2. Newton's Method

The Newton-Raphson method, or Newton method, is also a powerful technique for solving equations numerically. It is very similar to gradient descent however, Newton's method requires the function $f(\boldsymbol{x})$ to be twice differentiable. This is because Newton's method updates the parameters in each iteration by the amount of the gradient ($\nabla f(\boldsymbol{x}^{(t)})$) multiplied by the inverse of the Hessian matrix ($(\nabla^2 f(\boldsymbol{x}^{(t)})^{-1})$, which is a square matrix of second-order partial derivatives of the function (1).

Intuitively, the Hessian describes the local curvature of the loss function, which allows us to perform a more efficient update. In particular, multiplying by the inverse Hessian leads

the optimisation to take more aggressive steps in the directions of shallow curvature and shorter steps in directions of steep curvature. Note, crucially, the absence of any learning rate hyper-parameters in the update formula, which the proponents of these methods cite this as a large advantage over first-order methods. Therefore,

$$\boldsymbol{x}^{(t+1)} = \boldsymbol{x}^{(t)} - (\nabla^2 f(\boldsymbol{x}^{(t)}))^{-1} \nabla f\left(\boldsymbol{x}^{(t)}\right) \tag{3}$$

However, to make the two algorithms more comparable, we modify Newton's method to include a learning rate,γ_t, to enable us to implement basic backtracking. Consequently,

$$\boldsymbol{x}^{(t+1)} = \boldsymbol{x}^{(t)} - \gamma_t (\nabla^2 f(\boldsymbol{x}^{(t)}))^{-1} \nabla f\left(\boldsymbol{x}^{(t)}\right) \tag{4}$$

where we start with $\gamma_t = 1$. We then check whether $f\left(\boldsymbol{x}^{(t+1)}\right) < f\left(\boldsymbol{x}^{(t)} - \gamma_t (\nabla^2 f(\boldsymbol{x}^{(t)}))^{-1} \nabla f\left(\boldsymbol{x}^{(t)}\right)\right)$. If not, $\gamma_t \leftarrow \gamma_t/2$ and repeat.

2. Gradient Descent vs Newton's Method

2.1. Contour Plots

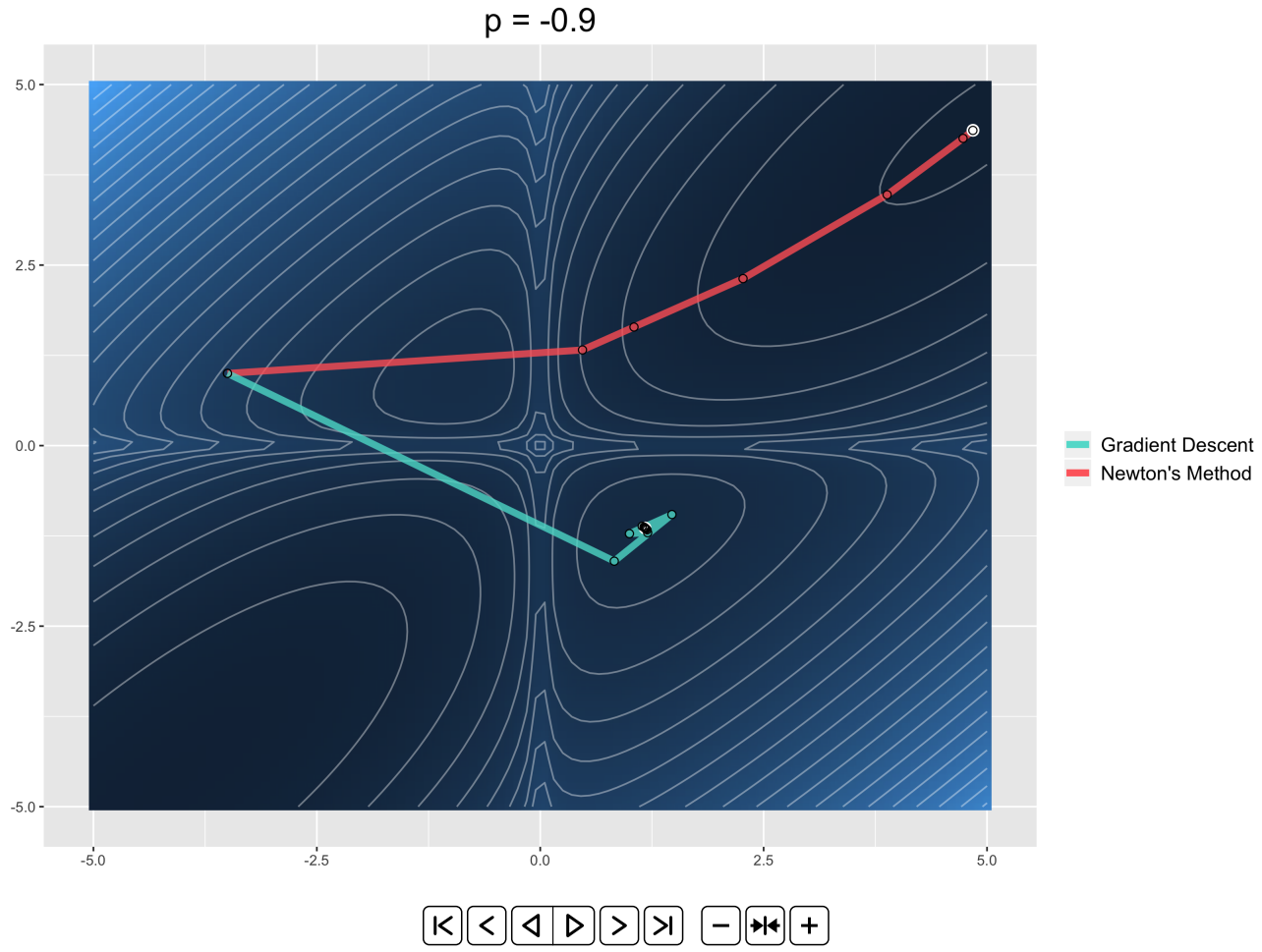
We begin the algorithmic showdown by focusing on the two-dimensional case ($n = 2$) where $\boldsymbol{m} = (0.5; 0)$. Firstly, we illustrate the contour plots with the sample paths of both algorithms, while scrolling through $\rho = -0.9$ to 0.9 in increments of 0.1.

The ellipses in Figure 1 are the contours of the quadratic function (1), and the red and blue lines are the trajectories taken by Newton's method and gradient descent respectively, which were both initialised at $\boldsymbol{x}^{(0)} = [-3.5; 1]^\top$ for every ρ. Furthermore, the black circles in the figure (joined by the trajectory lines) mark the successive values of $\boldsymbol{x}$ that the Newtons method and gradient descent go through (i.e. the iterations).

Interestingly, it can be seen that the two algorithms find different local minima for specific values of ρ, even when initialised at the same point. Therefore, clearly, the algorithms cannot distinguish between a local minimum and a global minimum since they both approximate the function as a quadratic, which can only have one minimum. Moreover,

it appears that Newton's method converges to a local minimum in significantly fewer iterations when compared to gradient descent (this will be confirmed in the next section) although they both take steps of roughly the same length.

Figure 1: Gradient Descent vs Newton's Method, contour plots with sample paths, scrolling through $\rho = -0.9 : 0.9$



Notes: Starting point initialised at $(-3.5, 1)$

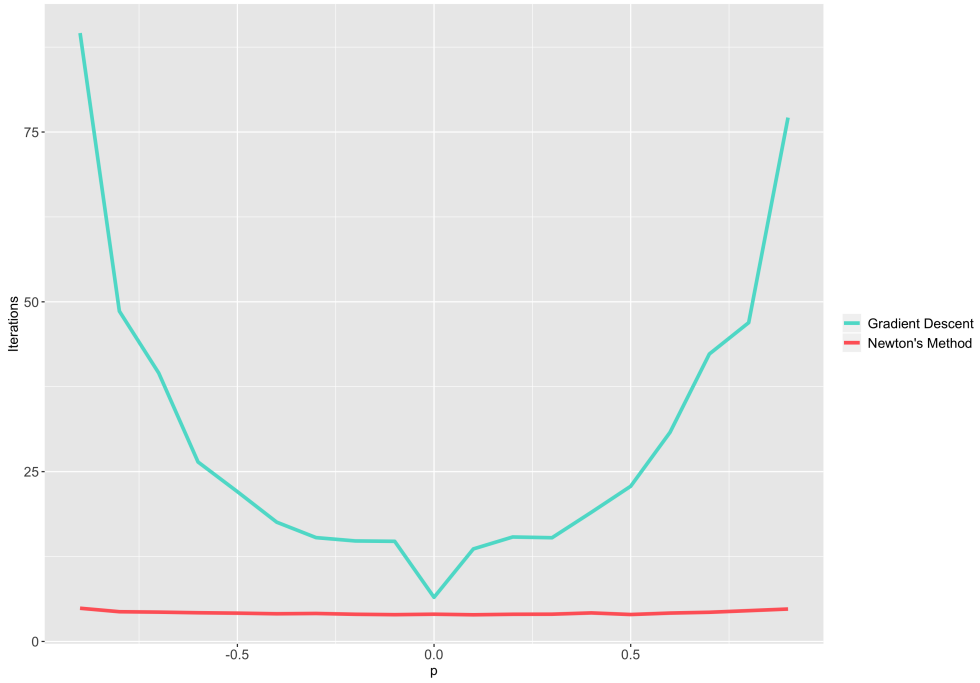
2.2. Iterations and Runtime ($n = 2$)

The second part of the showdown involves assessing the average number of iterations[1] required by each algorithm to converge to a local minimum, while scrolling through $\rho = -0.9$ to 0.9. Figure 2 highlights this result and confirms that Newton's method requires

[1]The average number of iterations are calculated by averaging the number iterations of 500 simulations at every ρ

substantially fewer iterations to converge, compared to gradient descent, across all values of ρ. Gradient descent requires 89 and 77 iterations on average at $\rho = -0.9$ and $\rho = 0.9$ respectively, and only needs approximately 6 iterations when $\rho = 0$. Conversely, Newton's method needs only 4 iterations , on average, across all values of ρ.

Figure 2: Average number of iterations over ρ

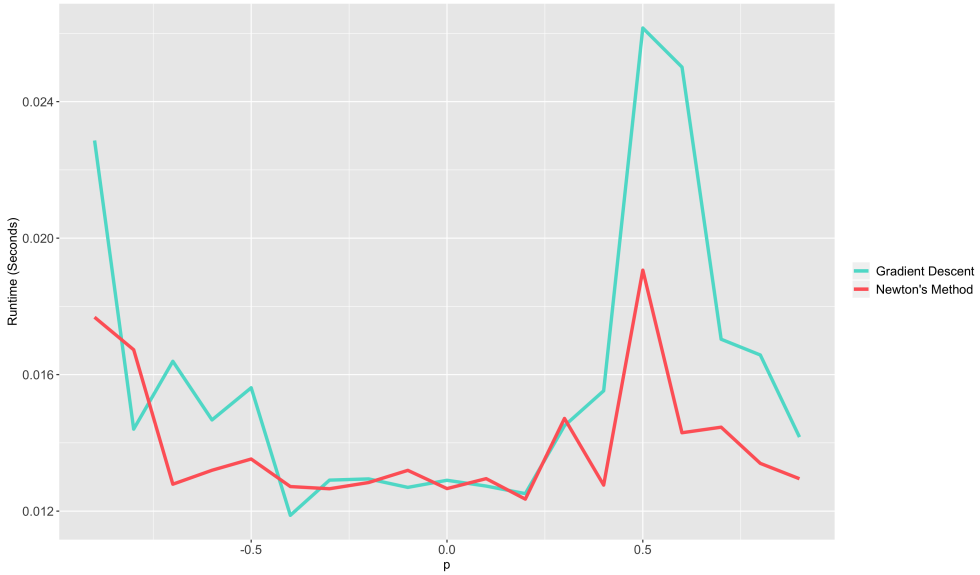


Notes: The average number of iterations are calculated by averaging the number iterations of 500 simulations at every ρ

Nevertheless, when assessing the average runtime[2] of the algorithms while scrolling through $\rho = -0.9$ to 0.9 it can be seen (see Figure 3) that there are no significant differences between the two methods.

[2]The average runtimes are calculated by averaging the runtimes of 500 simulations at every ρ

Figure 3: Average runtime (seconds) over ρ



Notes: The average runtimes are calculated by averaging the runtimes of 500 simulations at every ρ

3. Higher Dimensions

Optimising functions of this kind has recently become important in Bayesian model selection procedures involving high-dimensional regression problems. In that case, n will be much larger, say 50, 100, 200, 500 or 1000. Therefore, in this part of the showdown, we will choose random values for $\boldsymbol{m}$ and A (see function (1))[3] and study (by simulations) how the performance of gradient descent and Newton's method scales with n.

3.1. Iterations and Runtime ($n = 50, 100, 200, 500$ and 1000)

The final part of this showdown assess the average number of iterations[4] and the average runtime[5] of the algorithms while scaling n.

Figure 4 and Table 1 depict the average number of iterations required by each algorithm to converge to a local minimum, for different dimensions (n). Newton's method scales well

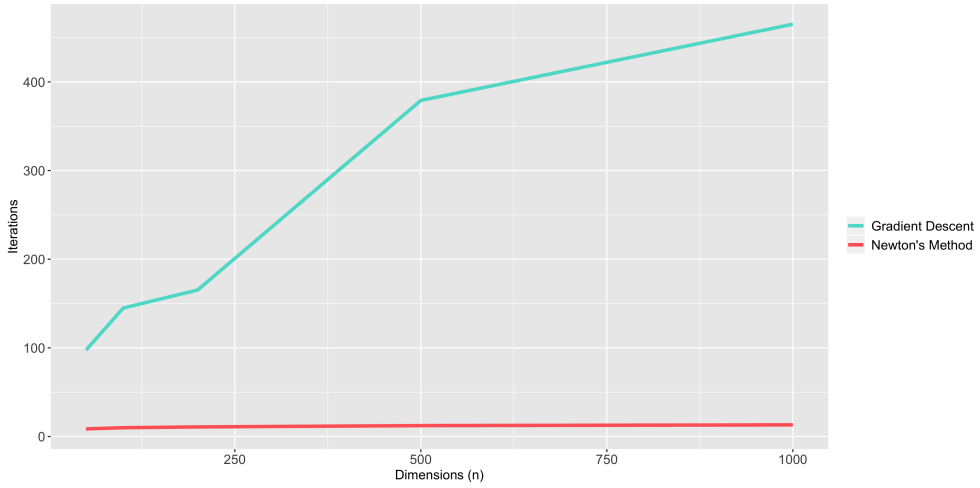
[3]Where $\boldsymbol{m}$ is drawn from a uniform distribution $\boldsymbol{m} \in \{-1, 1\}$) with n observations and A is drawn from a normal distribution with a mean equal to 0 and standard deviation equal to 1 with $n * n$ observations.

[4]The average number of iterations are calculated by averaging the number iterations of 100 simulations for dimensions n

[5]The average runtimes are computed by averaging the runtimes of 100 simulations for dimensions (n)

with n, as it only requires between 8 and 13 iterations, on average, across all dimensions to converge. Conversely, gradient descent scales poorly and requires significantly more iterations at every dimension of n. Furthermore, unlike Newton's method where the iterations remain fairly constant over n, gradient descent requires more iterations as n increases.

Figure 4: Average number of iterations over dimensions (n)



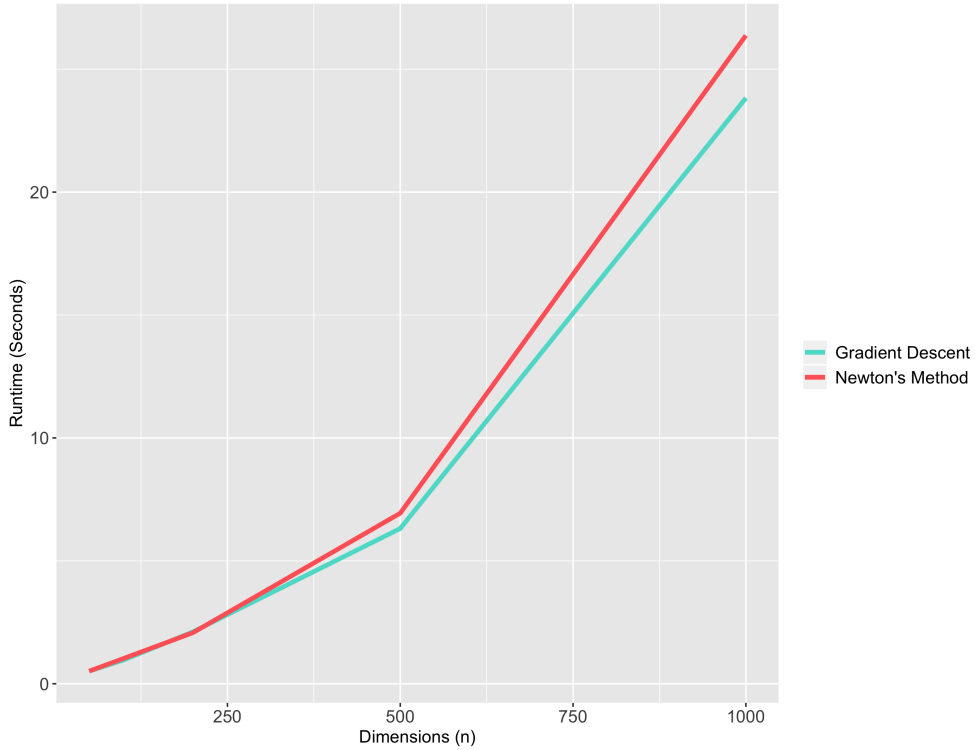
Notes: The average number of iterations are calculated by averaging the number of iterations of 100 simulations for dimensions n

Table 1: Average number iterations over dimensions (n)

Dimensions	**50**	**100**	**200**	**500**	**1000**
Gradient Descent	97.51	144.89	165.24	379.1	465.15
Newton's Method	8.57	9.89	10.78	12.26	13.18

Nonetheless, when evaluating the average runtime of the algorithms over n dimensions (see Figure 5 and Table 2), both methods show a similar pattern and do not differ considerably. In fact, at higher dimensions (i.e. $n = 500$ and $n = 1000$) gradient descent appears to be faster (on average). Thus, even though Newton's method requires fewer iterations, there are no differences in the runtime compared to gradient descent (if anything, Newton's method is slower at higher dimensions). This is likely due to the fact that Newton's method requires the Hessian to be derived and implemented, which requires $\mathcal{O}(n^2)$ operations and inverted which requires $\mathcal{O}(n^3)$ operations.

Figure 5: Average runtime (seconds) over dimensions (n)



Notes: The average runtimes are calculated by averaging runtimes of 100 simulations for dimensions n

Table 2: Average runtime (seconds) over dimensions (n)

Dimensions	**50**	**100**	**200**	**500**	**1000**
Gradient Descent	0.51	0.97	2.11	6.32	23.83
Newton's Method	0.52	1.03	2.08	6.94	26.37