Newton Direction in Descent Methods for Optimization

Caleb Maddry

Abstract

Newton's method is faster than gradient descent and displays quadratic convergence compared to gradient descent's linear convergence rate. I implemented Newton's method and tested it on different functions (quadratic, Rosenbrock, and Himmelblau's) to observe the method's behavior. The resulting algorithm converges quadratically. To bypass the inversion of the Hessian matrix - a relatively slow process, I implemented different quasi-Newton methods: Lazy-Newton, the Broyden-Fletcher-Goldfarb-Shanno algorithm, the DFP algorithm, and the symmetric rank-1 update method. Finally, I discussed Trust Region methods as an alternative to line search methods.

1 Introduction

Optimization problems are everywhere. Public policy, economics, science, engineering, and hundreds of other fields solve optimization problems to gain insights into the real world [1] [2] [7]. The idea is to choose the "best value" given an input set of parameters. For example, if we want to buy a car, we might take different criteria into consideration. Do we need a lot of seats? Do we want to drive fast? Will I take this car up I-70 to go skiing? These are parameters that we would use when deciding what car to buy.

You could think about these parameters as inputs to a function. Then, the "best choice" would be the choice that minimizes this function. In our case, we will be examining unconstrained, smooth optimization. We will assume there are no constraints on the variables, and assume the function is smooth. There are a number of ways to minimize a function. First, we will consider the gradient descent method that we learned about in class.

1.1 Gradient descent

In class, we discussed an optimization method called "gradient descent" or "steepest descent". Given its name, it is quite clear what this method does. Each iteration takes a step in the direction of steepest descent. In other words, we are stepping in the direction opposite to the gradient ∇f. This process is repeated until the minimum has been found.

Derivation

In optimization, our goal, given a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, is to find the minimum of f:

$$\min_x f(x)$$

In the extension of this project we'll examine the *trust region strategy*, but first we will look at *line search methods*. Iterative line search methods take in iterative step x_{k+1}. Each step takes the following form:

$$x_{k+1} = x_k + \alpha_k d_k$$

Where d_k is the descent direction. As the name implies, a decent direction is a direction where a step in this direction causes a decrease in the function value. In other words, $f(x_k) > f(x_k + \alpha_k d_k)$, where α_k is the size of the step. We can restate this condition in the following form:

$$\nabla f(x_k) \cdot d_k < 0 \quad \text{or} \quad \nabla f(x_k)^T d_k < 0$$

An obvious first choice for d_k, and the direction used in gradient descent, is simply: $d_k = -\nabla f(x_k)$ and therefore we have the following iteration:

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

This iteration is used in the gradient descent optimization method. We are taking a ”step” into the direction of steepest descent. It would be similar to descending a mountain in the direction of the steepest path. The size of the step taken is determined by the quantity α. If the step is too large, we will simply overshoot the minimum. If the step is too small, the method will converge slowly. To determine α, I use a backtracking line search.

1.1.1 BackTracking Line Search

There are a variety of different ways to choose an appropriate value for α. A line search seeks to find α_k by minimizing the function $f(x_k)$ along the direction d_k. That is, minimizing the following function: $f(x_k + \alpha_k d_k)$. That is,

$$\min f(x_k + \alpha_k d_k) \quad \text{where} \quad \alpha > 0$$

An exact solution to the minimization problem would be costly and isn’t necessary here. Instead, an inexact solution is approximated using a backtracking line search. This method involves an initial step $\alpha_k = 1$, and decreases the step until a *sufficient decrease condition* (also called the Armijo-Goldstein condition) is met. Here is the following image from [5] showing this condition.

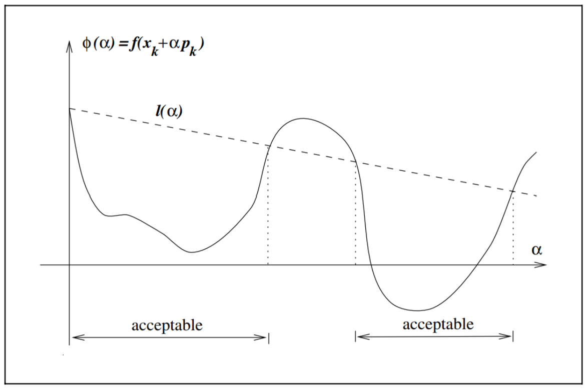


Figure 1: Figure 3.3 from [5] illustrating the sufficient decrease condition.

Essentially, a decrease in direction, proportional to the gradient, is required. This ensures the method will decrease from the function value at each iteration.

A further constraint, known as the curvature condition, is placed on α_k to further improve this method. Here is the following image from [5] showing this condition.

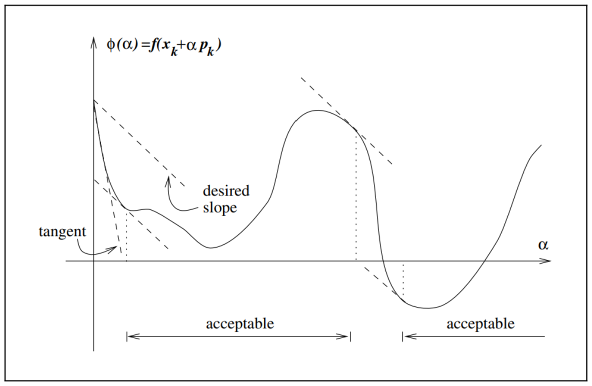


Figure 2: Figure 3.4 from [5] illustrating the curvature condition.

The curvature condition ensures the step is taken into a region where the function flattens out. After all, our goal is to move towards a minimum. Together, these conditions ensure α_k is chosen adequately.

Below are the Armijo-Goldstein condition and the curvature condition written formally:

The **Armijo-Goldstein condition**:

$$f(x_k + \alpha_k d_k) \le f(x_k) + c_1 \alpha \nabla f_k^T d_k$$

And the **curvature condition**:

$$\nabla f(x_k + \alpha_k d_k)^T d_k \ge c_2 \nabla f_k^T d_k$$

The parameters c_1 and c_2 can be tuned to improve the line search algorithm. In my implementation, I set $c_1 = 10^{-4}$ and $c_2 = 0.9$. In Newton's method, as the steps become closer to the function minimum, $\alpha_k = 1$ and the method converges quadratically. On the other-hand, gradient descent choose a small α_k and exhibits a "zig-zagging behavior" that we will see in the implementation.

1.2 Newton Descent

Gradient descent converges linearly. For some applications, this may be sufficient, however, a method that converges faster can improve our algorithm. By reformulating the optimization problem as a root-finding problem, we can use Newton's method. This method converges quadratically.

When $\nabla f(x^*) = 0$, $f(x^*)$ is either a local minimum, or a local maximum. If we know the function is concave down for all points, then x^* is a global minimum and the solution to our optimization problem. This is the basis of Newton's method.

By reformulating the optimization problem as a root-finding problem, we obtain a method that converges quadratically. In the next section we will derive Newton's method.

Derivation

Assuming f is twice differentiable, the second order Taylor series of

$$f(x_k + \alpha p) \approx f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T \nabla^2 f(x_k + \alpha p) p$$

If we let this be a function of p, called $m_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T \nabla^2 f(x_k + \alpha p) p$. The rate of change of f along the direction p, evaluated at x_k, is given by:

$$\nabla m_k(p) = \nabla f(x_k) + \nabla^2 f(x_k)p$$

$m_k(p)$ is the function we want to minimize this function in order to find the direction of greatest descent. Therefore, we seek to solve the following equation:

$$\implies \nabla f(x_k) + \nabla^2 f(x_k)p = 0$$

$$\implies \nabla f(x_k) = -\nabla^2 f(x_k)p$$

$$\implies (\nabla^2 f(x_k))^{-1}\nabla f(x_k) = -p$$

$$\implies p_k = -(\nabla^2 f(x_k))^{-1}\nabla f(x_k)$$

And finally, we have arrived at the Newton descent direction p. Using our descent formula, we then arrive at the Newton descent step:

$$x_{k+1} = x_k - \alpha_k(\nabla^2 f(x_k))^{-1}\nabla f(x_k)$$

Recall the descent property: $\nabla f(x_k)^T p_k < 0$ which is the condition required for descent. We then look to find the requirement for the Newton descent direction to be a descent direction.

$$\implies \nabla f(x_k)^T p_k = -\nabla f(x_k)^T(\nabla^2 f(x_k))^{-1}\nabla f(x_k) < 0$$

$$\implies 0 < \nabla f(x_k)^T(\nabla^2 f(x_k))^{-1}\nabla f(x_k)$$

A sufficient condition for this to hold true is the Hessian matrix, $H(x_k) = \nabla^2 f(x_k)$, has to be symmetric and positive definite (SPD). Additionally, as $m_k(p)$ was derived using the Taylor series approximation for $f(x_k + \alpha p) \approx m_k(p)$, the difference between $m_k(p)$ and the true function $f(x_k + \alpha p)$ cannot be too large. If these conditions hold, then the Newton direction is a reliable descent direction. In practice, the inverse Hessian isn't directly computed from the Hessian. If the Hessian is ill-conditioned, the inversion will lead to numerical instability. Instead, and LU decomposition is used.

1.3 Implementation and testing

Here, I have implemented the Newton descent direction in Python. It uses a Backtracking Line Search with the Armijo-Goldstein condition to determine α. I applied Newton and steepest descent to different functions and observed their behavior.

1.3.1 Implementation

Notes:

1. *The stopping condition of the algorithms is specified by a tolerance for the difference in iterates. For all of these examples, this was set as tolerance* $= 1 \times 10^{-16}$. *That is, 16 digits of accuracy.*

2. *The Hessian and gradient are calculated ahead of time and these are called by the methods. They are not being calculated by the algorithm at all.*

3. *To prevent an infinite number of iterations in the case where the method does not converge, the maximum number of iterations is set to 100,000.*

The simplest case is a simple **quadratic function**: $f(x, y) = 6x^2 + y^2$

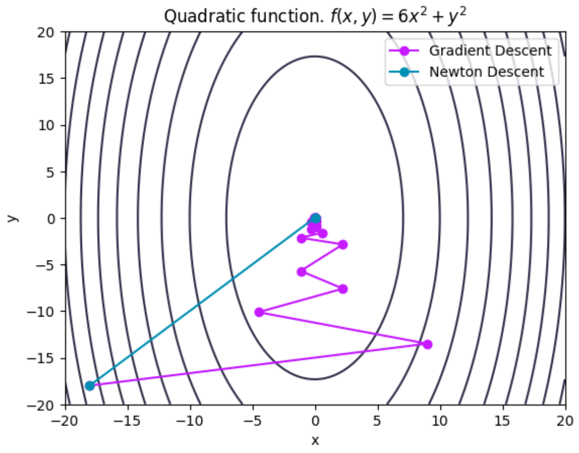


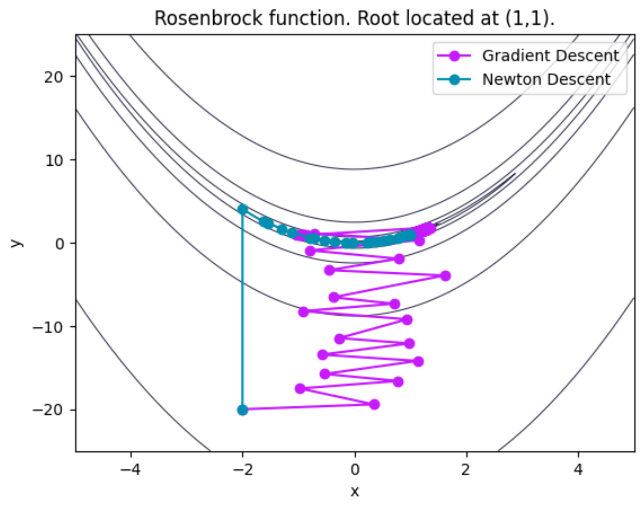
Figure 3: Gradient descent converged in **105 iterations** and Newton descent converged in **1 iteration** with $x_0 = (-18, 18)$. We can clearly see the zig-zagging behavior of Newton descent.

For all choices of x_0, both gradient descent and Newton descent will converge to the root located at $(0, 0)$. Newton's method uses a quadratic approximation of f, so the minimization immediately finds the minimum. In gradient descent, the number of iterations depends on the distance x_0 is from the root. This is a boring example with an obvious outcome. We will now begin to apply the methods to more complicated functions.

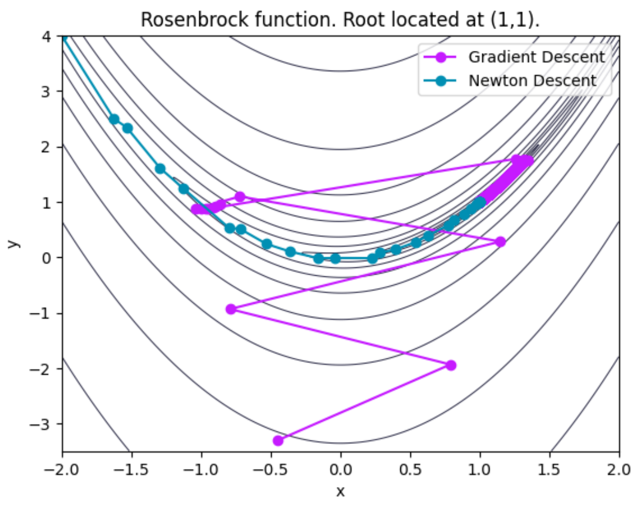
A function that is often used to test optimization methods is the **Rosenbrock function**:

$$f(x, y) = (a - x)^2 + b * (y - x^2)^2$$

Here, I chose $a = 1$ and $b = 100$. The Rosenbrock function has a steep entrance to a "banana-shaped" valley. The valley is relatively flat and only has a single root. Here, the root is located at the point $(1, 1)$.

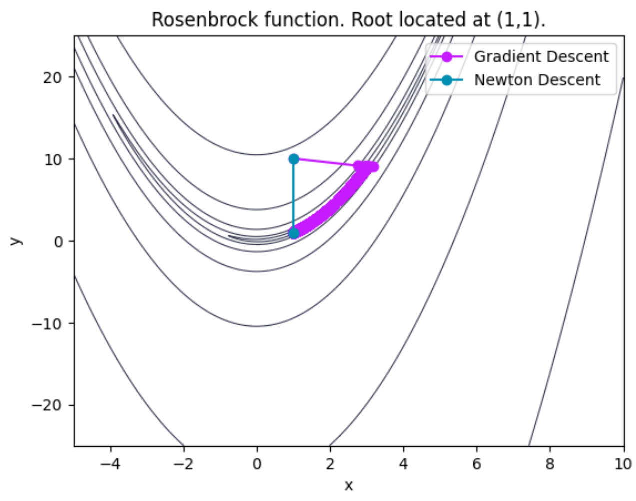


(a) $x_0 = (-2, -20)$

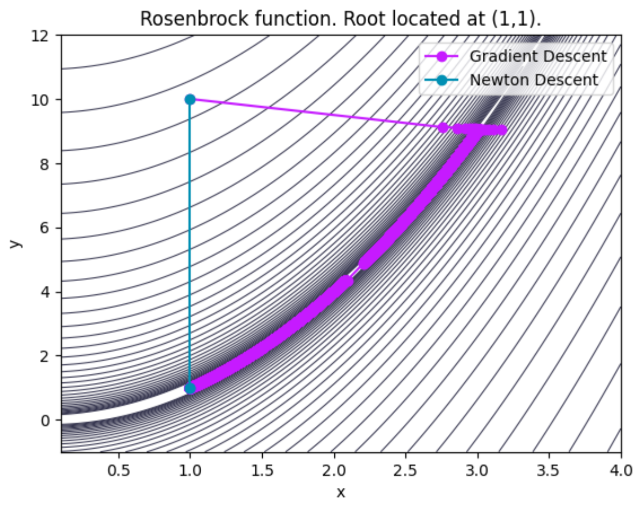


(b) $x_0 = (-2, -20)$

Figure 4: Gradient descent converged in **33,354 iterations**. Newton descent converged in **26 iterations**.



(a) $x_0 = (1, 10)$



(b) $x_0 = (1, 10)$

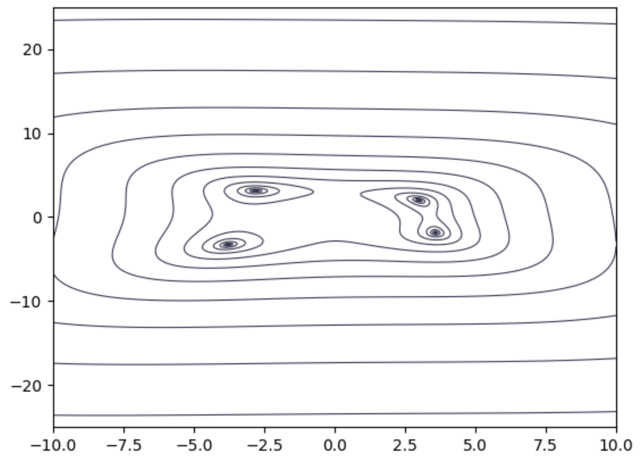
Figure 5: Gradient descent converged in **55,184 iterations**. Newton descent converged in **1 iteration**. In this case x_0 is too good and we see the same behavior as the quadratic function. In general, Newton descent converges quadratically.

The Rosenbrock function is difficult for optimization methods because most of the function is flat compared to the steep entrance of the ”banana-shaped” basin. Furthermore, the basin is relatively flat, so gradient descent doesn't perform well. Newton descent finds the root much faster than gradient descent. Gradient descent entered the valley relatively fast, but it proceeded to take thousands of subsequent steps in order to locate the correct point. We will now examine the case where a function has multiple minima/roots, and examine the difference between the two methods.

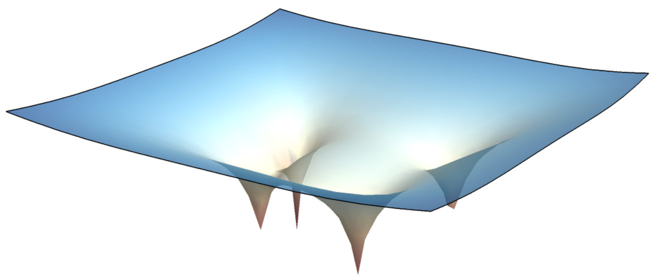
The final function that I used to test my implementation of the Newton Descent Direction is **Himmelblau's function**:

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

This is a multi-modal function that is commonly used to test the robustness of an optimization method. I used Newton and Gradient descent with different initial conditions to see which minima the methods converged to.



(a) Contour plot of Himmelblau's function.



(b) 3-D Plot. The Z axis has been plotted on a log scale to highlight the shape of the function.

Figure 6: Himmelblau's function. This function has 4 local minima.

First, I will show the case where Newton descent failed to find one of the correct roots:

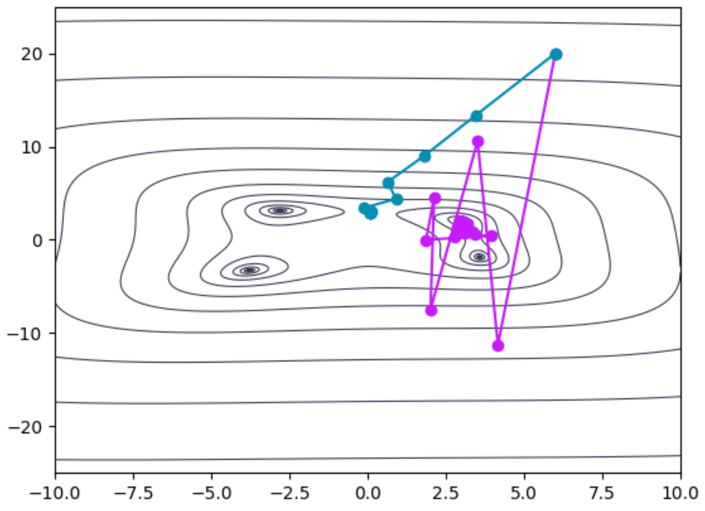
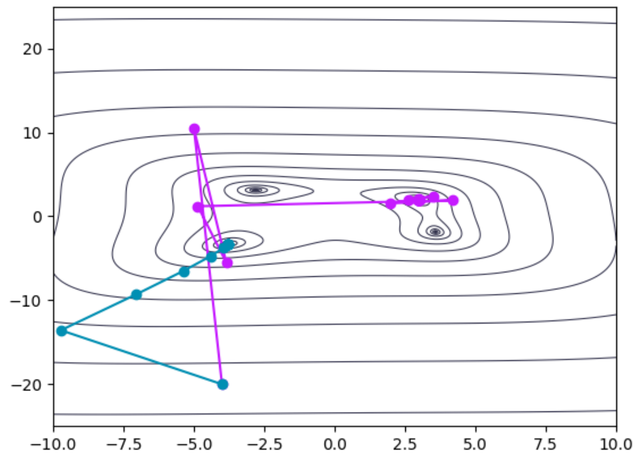
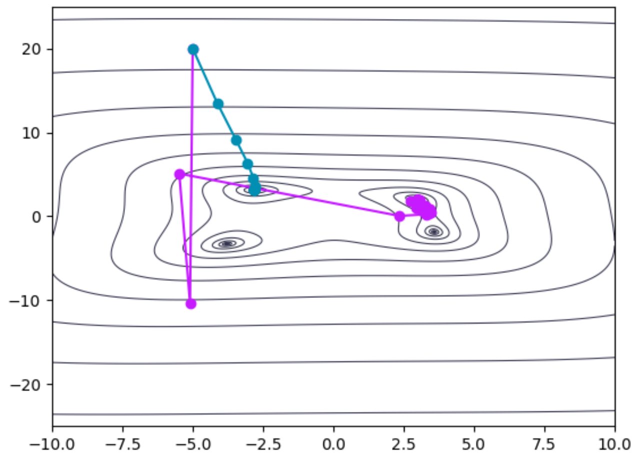


Figure 7: Newton descent failed to find the minima with $x_0 = (6, 20)$

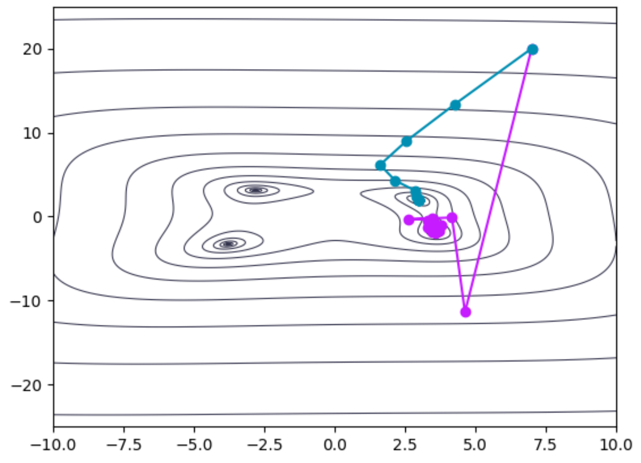
For a few choices of x_0, Newton descent would converge to the saddle point located at $(0.0866775, 2.8842547)$. The gradient is zero at this point and the backtracking line search was unable to "push" the method away from this point. There are a few points around $x_0 = (6, 20)$ that displayed this behavior. This is expected. The Hessian is not SPD at this point, so we aren't guaranteed convergence. For nearly all other choices, the method converged just fine to one of the four roots of Himmelblau's function.



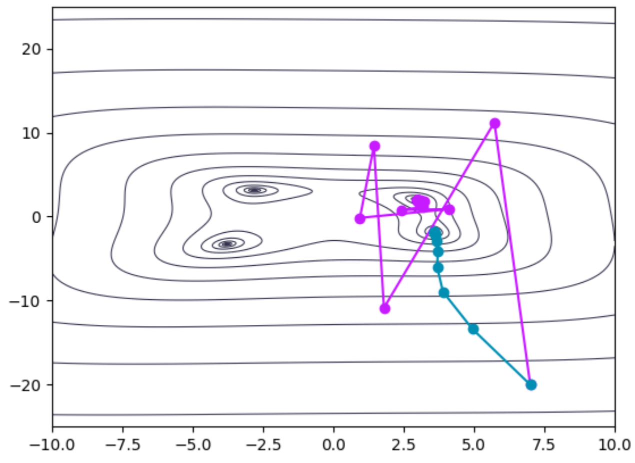
(a) Root located at $(-3.7793, -3.2832)$ with initial point $(-4, -20)$.



(b) Root located at $(-2.8051, 3.1313)$ with initial point $(-5, 20)$.



(c) Root located at $(3.0000, 2.0000)$ with initial point $(7, 20)$.



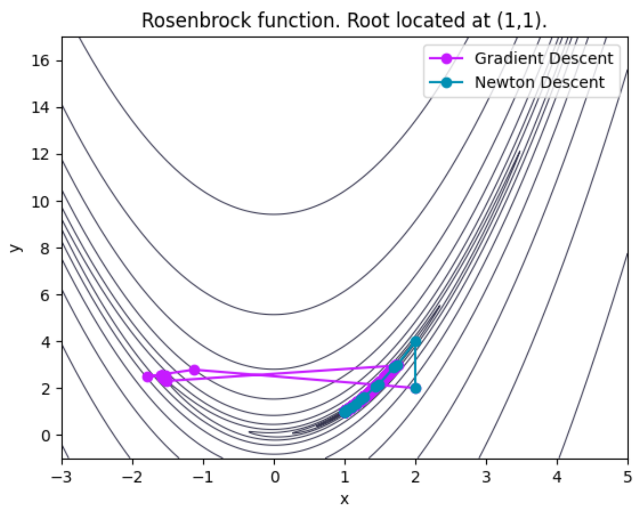
(d) Root located at $(3.5844, -1.8481)$ with initial point $(7, -20)$.

Figure 8: Using Newton's method to find the roots of Himmelblau's function. Gradient descent is unable to find all of the roots.

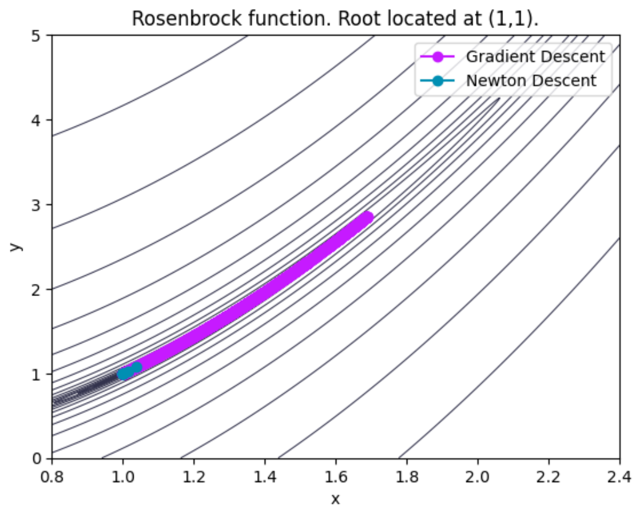
Newton descent was able to reliably found the 4 different minima with initial conditions that were closest to one particular minima. Gradient descent was unable to locate the left minima and consistently gravitated towards the right minima. Gradient descent behaves rather sporadically and will shoot off in different directions. It is difficult to choose the correct initial conditions such that steepest descent finds a root other than the one located at $(3, 2)$.

1.3.2 Convergence rate

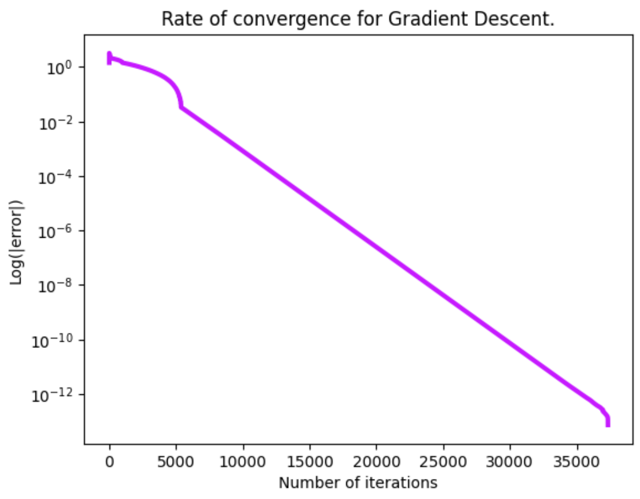
Next, we examine the convergence rate of Newton descent. To test this, I used the Rosenbrock function with the initial condition within the basin of convergence. For the example below, I choose $x_0 = (2, 2)$ as the starting point.



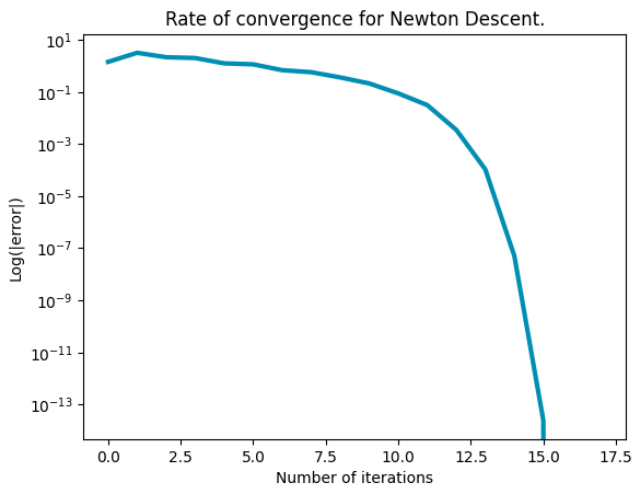
(a) Gradient descent converged in **37,335 iterations**.



(b) Newton descent converged in **16 iterations**.



(c) Gradient Descent displays a linear convergence.



(d) Newton Descent displays a quadratic convergence.

Figure 9: Comparison between the convergence rates of Newton and Gradient descent. Newton displays a super-linear convergence rate, whereas Gradient descent has a linear convergence rate.

We can clearly see the benefit of using Newton descent over gradient descent. Newton descent converges quadratically, whereas steepest descent only converges linearly.

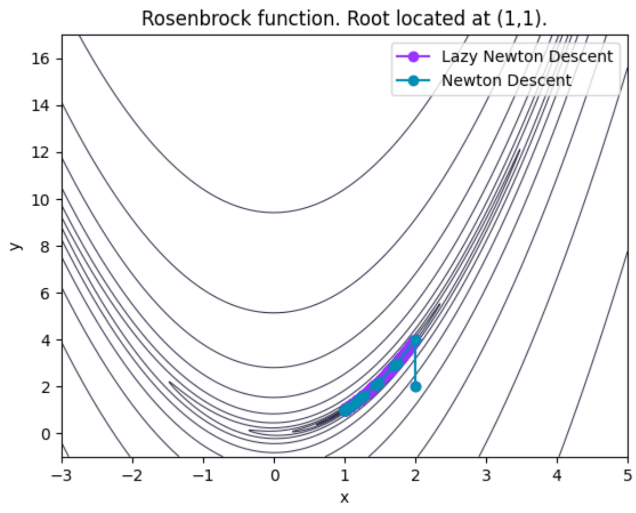
1.4 Variations of Newton descent

While Newton's method is much faster than steepest descent, computing the Hessian matrix is expensive. A few methods have been developed to skip this step and thus speed up the algorithm. We will explore two of these methods in the section below, and two more in the extension. For the rest of this paper, I will only use to Rosenbrock function to test the methods, because it shows their behavior well.

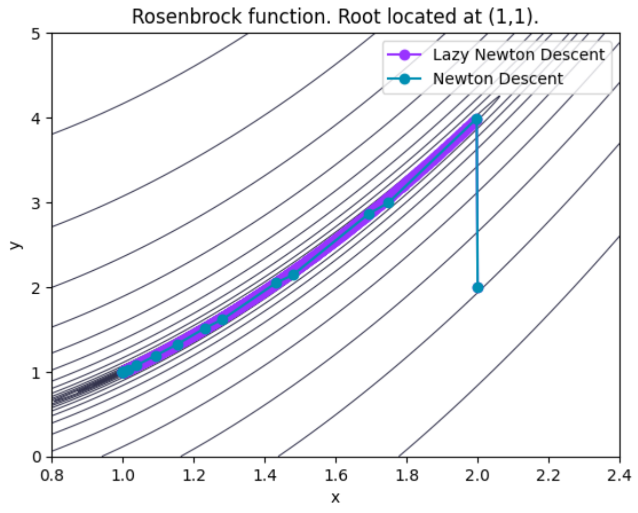
1.4.1 Lazy Newton

One possible way to speed up the Newton's descent algorithm is to limit the number of times the Hessian matrix is inverted. Lazy Newton inverts the Hessian once at the first step. This Hessian inverse is then used at each subsequent iteration instead of calculating the Hessian again. Essentially, Lazy Newton uses the Hessian at x_0 as the guess for the Hessian at each subsequent iteration.

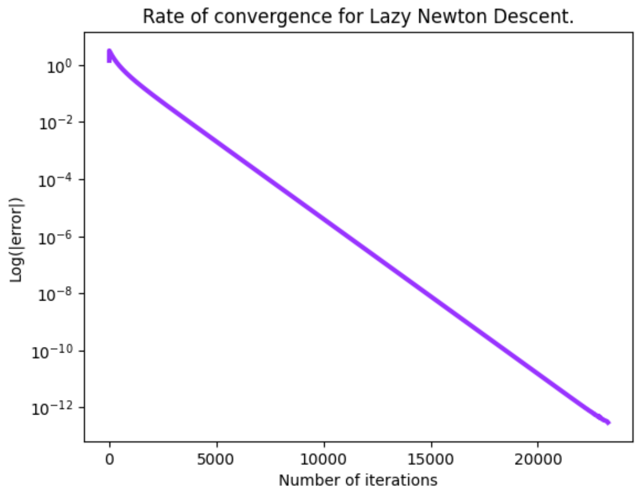
First, I will show the results of Lazy Newton applied to the Rosenbrock function with the same initial conditions as I showed in section 1.3.2.



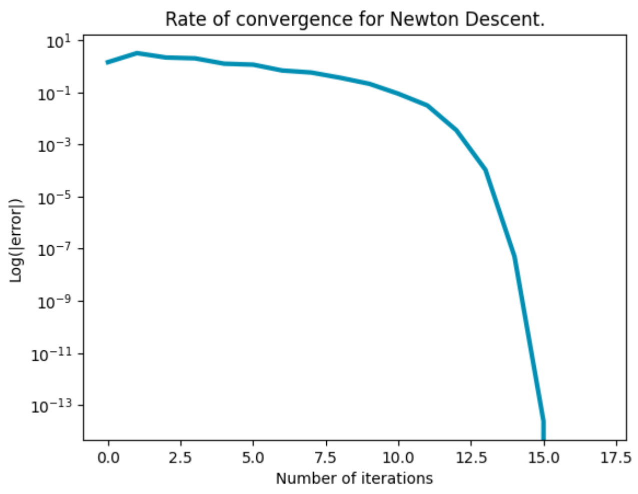
(a) Lazy Newton descent converged in **23,274 iterations**.



(b) Newton descent converged in **16 iterations**.



(c) Lazy Newton Descent displays a linear convergence.



(d) Newton Descent displays a quadratic convergence.

Figure 10: Comparison between the convergence rates of Newton and Gradient descent. Newton displays a super-linear convergence rate, whereas Gradient descent has a linear convergence rate.

Lazy Newton shows linear convergence. However, we can see the method converges in 23,274 iterations which is quicker than gradient descent which converged in 37,335 iterations. Just like regular Newton, Lazy Newton converges with a single iteration on the quadratic function. Lazy Newton performs better than gradient descent and offers a good alternative for cases where the Hessian is expensive to compute and invert.

1.4.2 Broyden-Fletcher-Goldfarb-Shanno algorithm

Instead of computing the Hessian directly, the Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS) algorithm seeks to generate an approximation of the Hessian (B_{k+1}) by using the secant equation:

$$B_{k+1}s_k = y_k, \quad \text{where } s_k = x_{k+1} - x_k = \alpha_k p_k \text{ and } y_k = \nabla f_{k+1} - \nabla f_k$$

This is similar to the quasi-Newton method in root-finding called *Broyden's Method*. However, in optimization, we are seeking to find the zeroes of the first derivative of $f(x_k)$, and for our method's to converge, we require that the Hessian is SPD. This extra constraint means we cannot use *Broyden's Method* because it doesn't ensure B_{k+1} is SPD, and therefore we are not guaranteed to converge. For BFGS, B_{k+1} will be SPD if the following condition, called the *curvature condition*, is satisfied:

$$s_k^T y_k > 0 \implies s_k^T B_{k+1} s_k > 0$$

The derivation in [5] uses an approximation for the inverse Hessian H_{k+1} instead of B_{k+1}:

$$H_{k+1}s_k = y_k$$

H_{k+1} is chosen so that it is the update closest to the current update with respect to the weighted Frobenius norm. In other words, H_{k+1} is found by solving:

$$\min_H \|H - H_k\| \quad \text{with respect to the Frobenius norm}$$

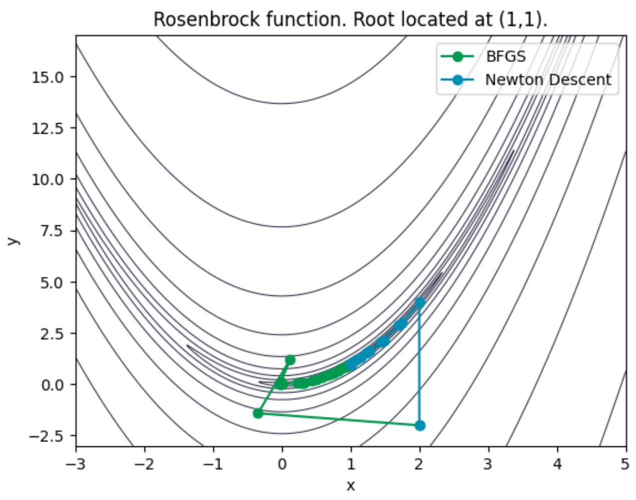
The solution directly obtains the following update formula:

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \quad \text{where } \rho_k = \frac{1}{y_x^T s_k}$$

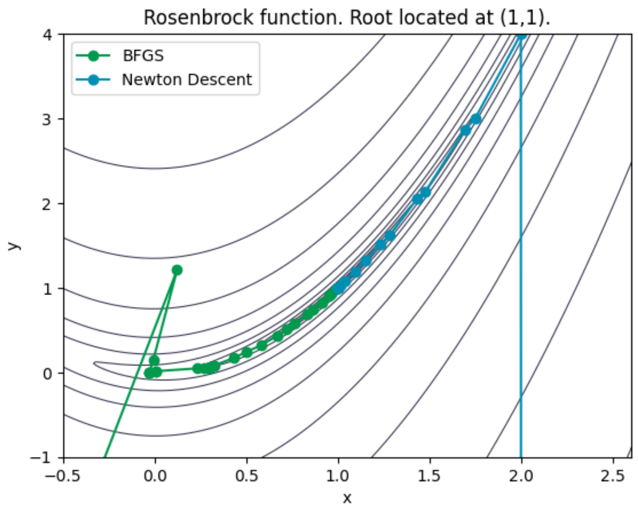
Where instead of computing B_{k+1} and solving the LU decomposition, the approximate inverse Hessian can be directly applied. There is still a formula for computing B_{k+1}, and it can be derived from H_{k+1} using the Sherman-Morrison-Woodbury formula. There are a few choices for the initial guess for H_0. The simplest way is to choose $H_0 = I$, the identity matrix.

I created two implementations of BFGS. One uses the update formula for H_{k+1}. The other implementation uses the update formula for B_{k+1} and performs an LU solve. They perform identically, but for the graphs below I use the inverse Hessian approximation.

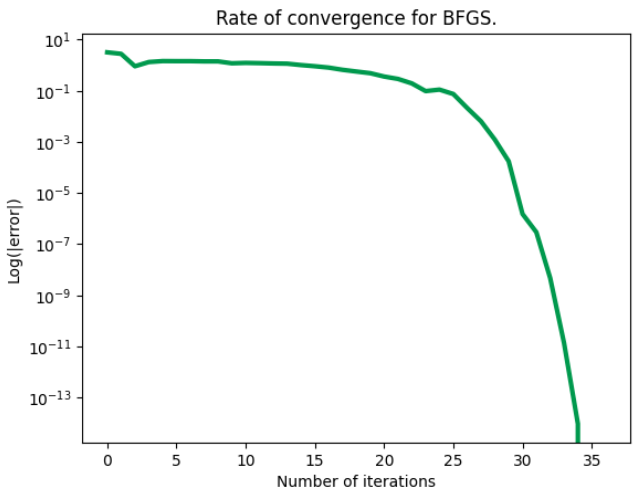
Here, I use the initial approximation: $H_0 = I_n$. It takes BFGS around 23 iterations until the Hessian is approximated well enough that quadratic convergence is displayed.



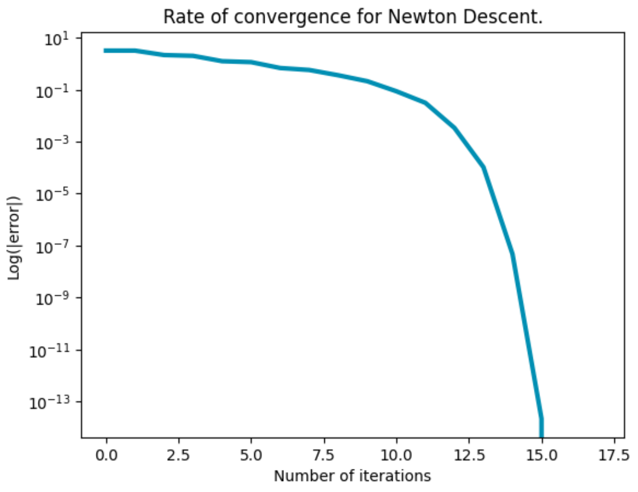
(a) BFGS converged in **35 iterations**.



(b) Newton descent converged in **16 iterations**.



(c) BFGS displays super-linear convergence.



(d) Newton Descent displays quadratic convergence.

Figure 11: Comparison between BFGS and Newton descent applied to the Rosenbrock function.

While Newton converges quadratically, the cost per iteration is usually higher than BFGS, especially for higher dimensional functions. This makes BFGS a better method for functions with more than 2-dimensions. Furthermore, BFGS doesn't require the Hessian to be computed at any point. This important because there are many of applications where the Hessian can't be computed, or it isn't known. This makes BFGS a powerful tool for these cases.

1.5 Conclusions for introductory material

I derived the Newton descent direction and applied it to a few different functions. We see that the method performs much better than gradient descent. Newton descent converges at a faster rate (super-linearly) compared to the linear convergence of steepest descent. Then, we found few cases where Newton descent fails to converge to the correct minimum. Inverting the Hessian matrix is a costly operation. To avoid this, we examined two methods that estimate the Hessian: Lazy Newton and the BFGS method. In the next section, we will examine Trust Region methods as opposed to the line search methods we have discussed. Additionally, I will implement two more quasi-Newton methods: the *symmetric rank-1 update* and the *DFP method*.

2 Trust Region methods and other quasi-Newton methods

2.1 Introduction

For the extension of this project, I will examine Trust Region methods as an alternative to the line search method we used in the previous section. Additionally, in this extension, we will look at two other quasi-Newton methods: **SR1** and **DFP**.

2.2 Trust region methods

Recall, line search methods find a direction p_k and search along this direction to find an acceptable step-length α_k. In Trust Region methods, our approach is different. We look at our current step, and find a "region" that we trust our approximation $m_k(p)$ that we derived in *section 1.2*.

Recall the problem from our derivation of Newton descent:

$$\min(p): \;\; m_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T \nabla^2 f(x_k + \alpha p) p$$

Because we want a small trust region (TR) where the solution is accurate, we constrain p such that $||p||_2 \leq \Delta_k$. Where we define Δ_k as the TR radius. That is, the size of the trust region. Furthermore, we generalize the problem by replacing $\nabla^2 f(x_k + \alpha p)$ with the Hessian approximation B_k.

$$\min(p): \;\; m_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T B_k p$$

From here on, I will refer to this problem as the TR minimization problem. The solution to this, p_k, is the TR direction. Now that the TR problem has been specified, we seek to answer the following questions:

1. How do we choose the TR radius Δ_k?
2. How, and to what accuracy, do we solve the TR minimization problem?

To answer the first question, given as step p_k, we define:

$$\rho_k = \frac{f(x_k) - f(x_k + p_k)}{m_k(0) - m_k(p_k)}$$

ρ_k quantifies how close the actual function is to the approximation $m_k(p)$. There are three case for ρ_k:

$$\rho_k = \begin{cases} \rho_k < 0 \text{ or } \rho_k \approx 0, & \text{Reject step and decrease } \Delta_k \\ 0 < \rho_k < 1, & \text{Do not alter trust region} \\ \rho_k \approx 1, & \text{Expand the region because there is good agreement between } m_k \text{ and } f \end{cases}$$

In practice, tolerances are used to specify cut-off points for ρ_k. Additionally, we need to know p_k to use this ratio. This is found with the TR minimization problem. There are three common methods to approximate the solution to the TR minimization problem. We will look at one of the more common methods: *the dogleg method*.

2.2.1 Dogleg Method

The dogleg method seeks to find an approximate solution to the TR optimization sub problem. In the line search method, we solve the minimization problem of α, but don't minimize the function exactly. We said that it wasn't necessary and was costly to do so. In a similar fashion, the TR minimization subproblem is also costly to solve, and it isn't necessary for our purposes. Instead, the dogleg method approximates p_k.

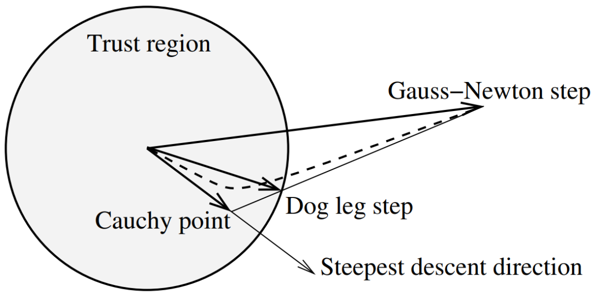


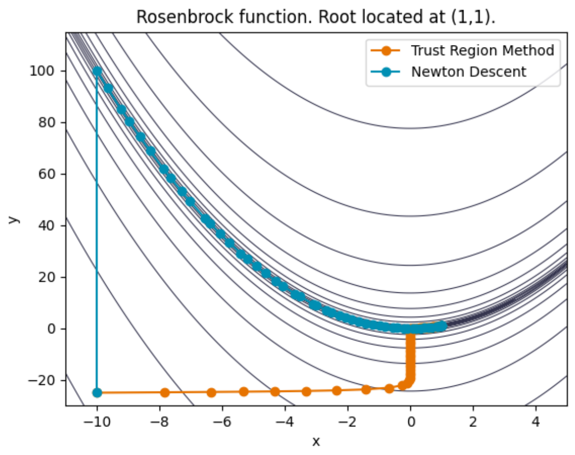
Figure 12: Figure 2 from [4] showing the dogleg method.

The dashed line represents the trajectory of p_k subject to the constraint $||p||_2 \leq \Delta_k$. This is the trajectory that we want to approximate with the dogleg method. The method generates two vectors- the Cauchy point and the Gauss-Newton step, and then connects these vectors together. The Cauchy point is a minimization of the function gradient $\nabla f(x_k)$ within the trust region. The Gauss-Newton is simply the step: $G_k = B^{-1}(B - \lambda I)p$. The point where the connecting vector and the trust region meet is the approximation of p_k.

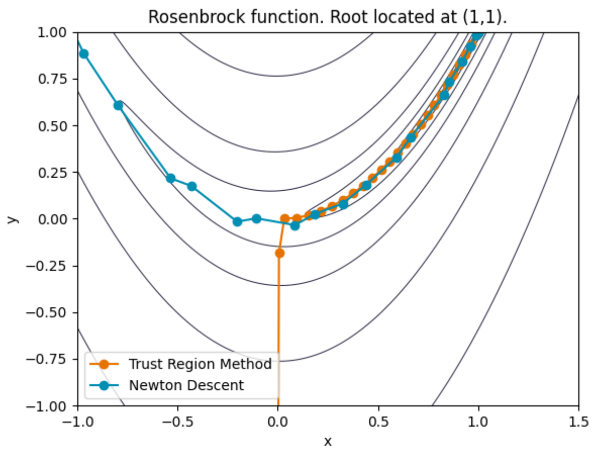
2.2.2 Implementation

For my implementation, I am using Scipy functions to solve the TR minimization problem. At some point, I plan on implementing my own version of the dogleg method and comparing it to my current implementation with Scipy.

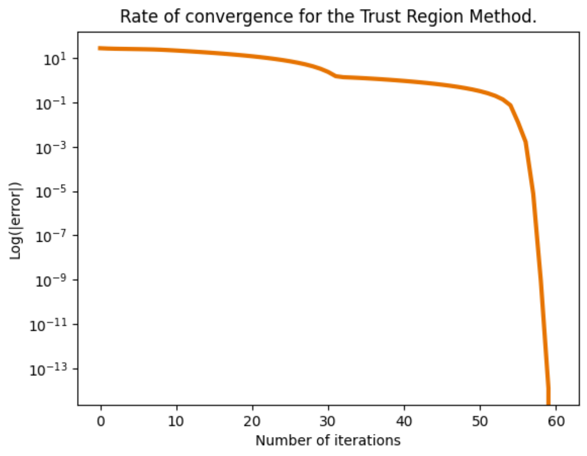
Additionally, my implementation of TR will use the true Hessian for the TR minimization problem. Of course, the Hessian approximations that BFGS, DFP, and SR1 generate can be used instead. In fact, according to [5], SR1 maintains relevance for its applications in the Trust Region method. As we'll see later on, as a quasi-Newton method, SR1 performs poorly.



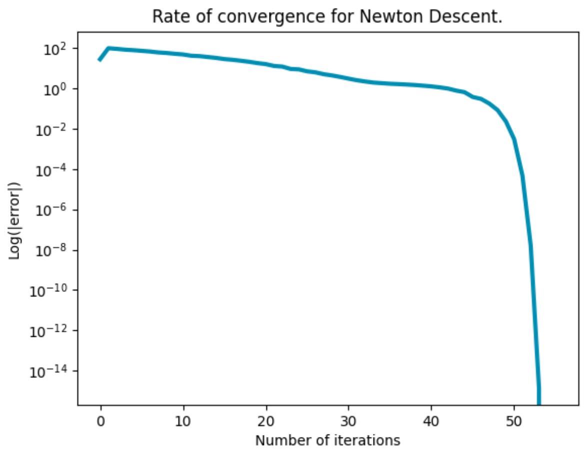
(a) TR converged in **64 iterations**.



(b) Lazy Newton descent converged in **54 iterations**.



(c) Trust Region displays quadratic convergence.



(d) Newton Descent displays quadratic convergence.

Figure 13: Comparison between Trust Region, with the true Hessian, and Newton descent applied to the Rosenbrock function.

2.3 Other Quasi-Newton Methods

We have already seen two quasi-Newton methods: lazy-Newton and BFGS. I will now introduce two more quasi-Newton methods that can be used to generate approximation to the Hessian matrix.

2.3.1 DFP

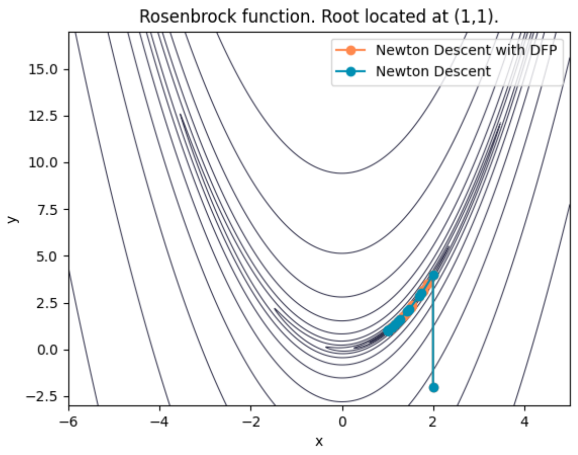
The DFP update formula was actually discovered before BFGS. The two methods are quite similar, and the derivation is nearly the same. However, instead of solving $\min_H \|H - H_k\|$, you instead solve:

$$\min_B \|B - B_k\| \quad \text{with respect to the Frobenius norm}$$

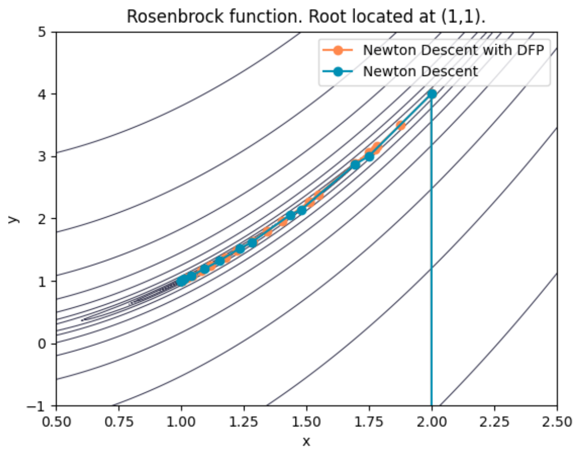
An equation for B_{k+1} is found, and then the Sherman-Morrison-Woodsbury formula is used to find the update for the approximation of the inverse Hessian:

$$H_{k+1} = H_k - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} + \frac{s_k s_k^T}{y_k^T s_k}$$

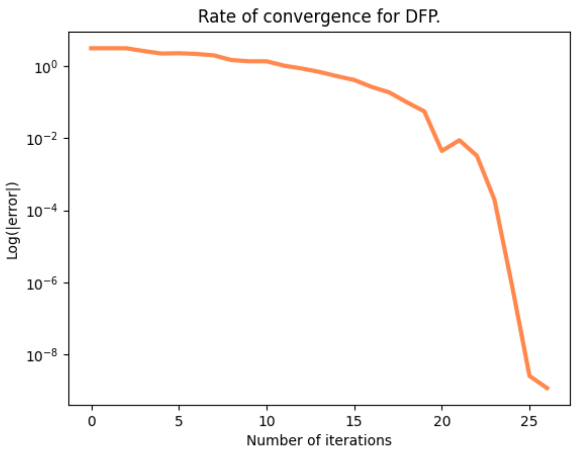
DFP isn't as good as BFGS at correcting bad Hessian approximations. I had to use the true Hessian for H_0 in order for the DFP to converge reliably.



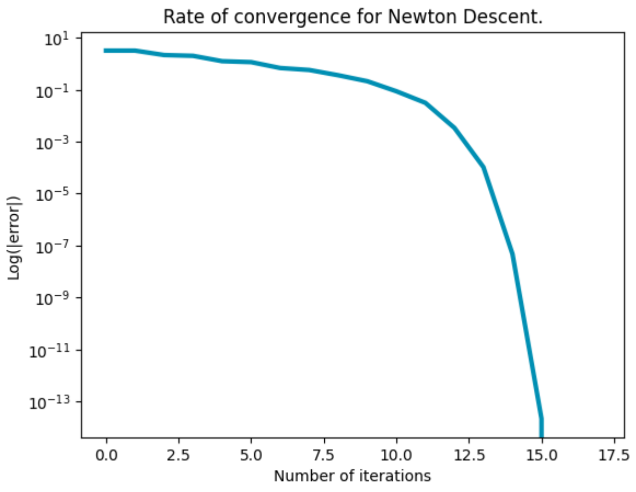
(a) DFP converged in **25 iterations**.



(b) Newton descent converged in **16 iterations**.



(c) DFP displays super-linear convergence.



(d) Newton Descent displays quadratic convergence.

Figure 14: Comparison between DFP and Newton descent applied to the Rosenbrock function.

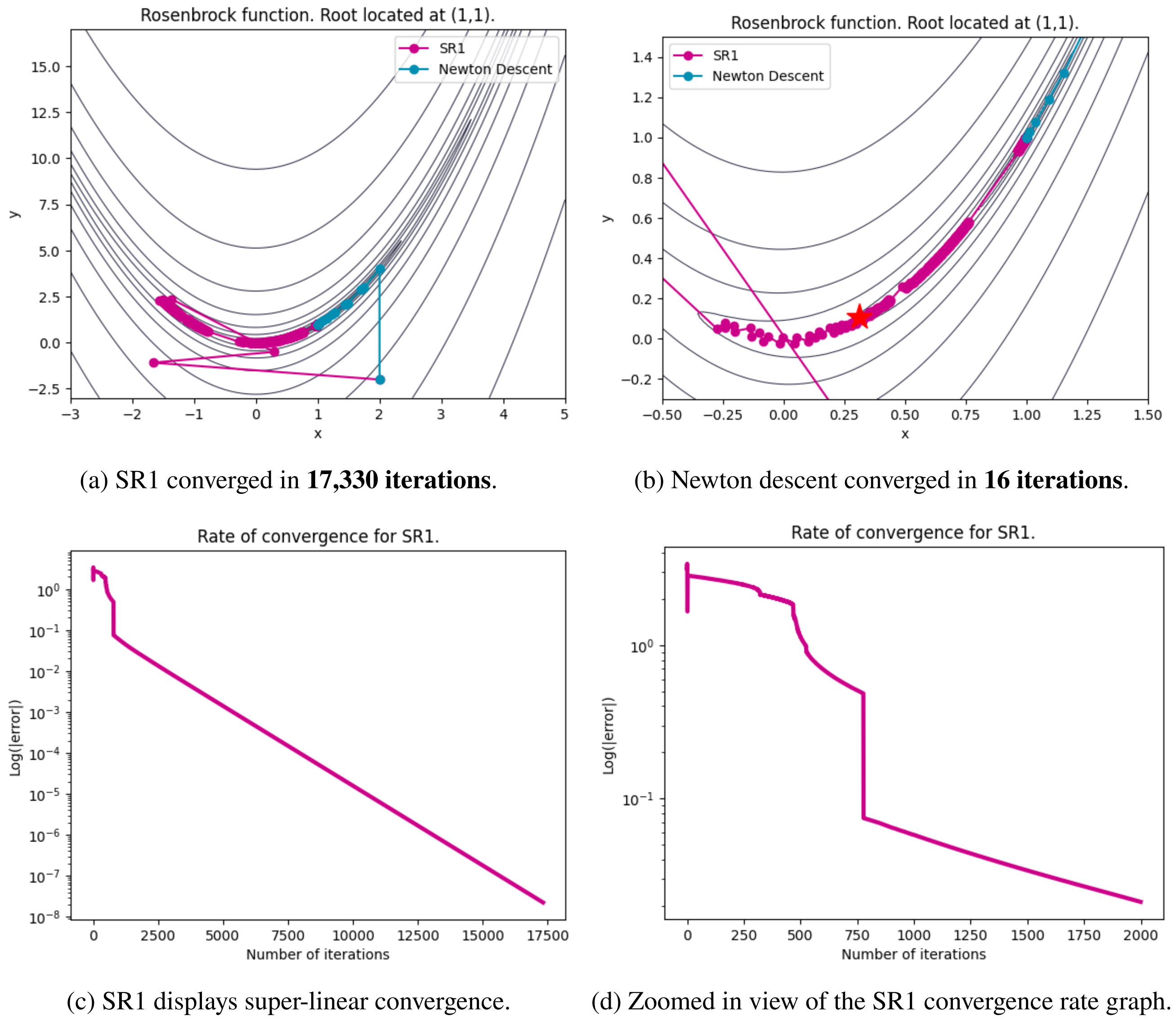
I noticed the DFP was more sensitive to the location of x_0. While it will almost always converge, there can be some spikes in the error curves. DFP wouldn't perform well unless the true Hessian was used for the choice of H_0. I used variations of H_0 that were scalar changes of the identity matrix, that is $H_0 = \beta I$, where β is some constant. This can be scaled to better resemble the true Hessian at x_0. I found that DFP was very sensitive to small perturbations in β. On the other-hand, BFGS performed well under all conditions that I used.

2.3.2 Symmetric rank 1 update (SR1)

The final quasi-Newton method that we will examine is the symmetric rank-1 update or SR1. Unlike BFGS and DFP, which used a rank-2 update at each iteration of H_k, SR1 is a symmetric rank-1 update. This is the only symmetric rank-1 update that solves the secant equation: $\min_B \|B - B_k\|$:

$$H_{k+1} = H_k + \frac{(s_k - H_k y_k)(s_k - H_k y_k)^T}{(s_k - H_k y_k)^T y_k}$$

After implementing the SR1 update, we get the following results:



(a) SR1 converged in **17,330 iterations**.

(b) Newton descent converged in **16 iterations**.

(c) SR1 displays super-linear convergence.

(d) Zoomed in view of the SR1 convergence rate graph.

Figure 15: Comparison between SR1 and Newton descent applied to the Rosenbrock function. The red star represents the point where my implementation begins skipping updates to the inverse Hessian approximation.

The SR1 update displays some interesting behavior. My implementation has a condition that checks if the denominator is close to zero. If it is, the SR1 update is skipped and the inverse Hessian approximate from the previous iteration is used. The denominator approaches zero when there isn't a symmetric rank-1 update that satisfies the secant equation. This is a well-known issue with SR1, and it is something that can even happen for the simplest cases. When my condition works, the method resembles that of Lazy-Newton, but the last approximate is used instead of the Hessian at x_0.

Looking at the error curve in more detail, we can see that the method starts out super-linear until the denominator of the SR1 update becomes too small, then the linear lazy Newton behavior takes over. I placed a red star that shows the point where the inverse Hessian approximate stops updating. We can see that just before this, the method looks more similar to the other quasi-Newton methods that I have shown.

3 Conclusion

The exploration of optimization techniques reveals a clear hierarchy in terms of efficiency and applicability. Newton's method, with its quadratic convergence, outperforms gradient descent, which suffers from a linear rate of convergence and a tendency to zig-zag while converging. However, the computational cost of Hessian inversion presents a challenge. Variants like Lazy Newton and quasi-Newton methods such as BFGS provide compelling alternatives, balancing performance and computational efficiency.

The BFGS algorithm stands out as a robust method. Approximating the Hessian without explicitly computing it makes it suitable for high-dimensional problems. Conversely, methods like SR1, demonstrate significant sensitivity to initial conditions.

The comparisons between these methods highlight the importance of selecting the right tool for the problem at hand. Newton's method remains ideal when accuracy and convergence speed are important, and Quasi-Newton methods serve as practical solutions when computational resources are limited. These findings not only reinforce the theoretical advantages of Newton-based methods but also shed light on their real-world applicability.

References

[1] Junlei Hu Alexandru V. Asimit Tao Gao and Eun-Seok Kim. “Optimal Risk Transfer: A Numerical Optimization Approach”. In: *North American Actuarial Journal* 22.3 (2018), pp. 341–364. DOI: 10.1080/10920277.2017.1421472. eprint: https://doi.org/10.1080/10920277.2017.1421472. URL: https://doi.org/10.1080/10920277.2017.1421472.

[2] Raymond M. Hicks and Preston A. Henne. “Wing design by numerical optimization”. In: *Journal of Aircraft* 15.7 (July 1978), pp. 407–412. DOI: 10.2514/3.58379. URL: https://doi.org/10.2514/3.58379.

[3] David M. Himmelblau. *Applied Nonlinear Programming*. McGraw-Hill, 1976.

[4] M.L.A. Lourakis and A.A. Argyros. “Is Levenberg-Marquardt the most efficient optimization algorithm for implementing bundle adjustment?” In: *Tenth IEEE International Conference on Computer Vision (ICCV’05) Volume 1*. Vol. 2. 2005, 1526–1531 Vol. 2. DOI: 10.1109/ICCV.2005.128.

[5] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. 2e. New York, NY, USA: Springer, 2006.

[6] B.T. Polyak. “Newton’s method and its use in optimization”. In: *European Journal of Operational Research* 181.3 (2007), pp. 1086–1096. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2005.06.076. URL: https://www.sciencedirect.com/science/article/pii/S0377221706001469.

[7] Carlos Vilas et al. “Dynamic optimization of distributed biological systems using robust and efficient numerical techniques”. In: *BMC Systems Biology* 6.1 (July 2012), p. 79. ISSN: 1752-0509. DOI: 10.1186/1752-0509-6-79. URL: https://doi.org/10.1186/1752-0509-6-79.

4 Code

Implementation of **Newton Descent**:

```
def NewtonDescent(x0, F, J, H, tol, Nmax):

   # Checks to see if the function is already at the root
   if norm(F(x0)) == 0:
      xstar = x0
      ier = 0
      its = 0
      return

   xStep = [x0]

   for its in range(Nmax):
      # Evaluate J
      Jeval = J(x0)

      # Evaluates the Hessian and computes its inverse
      Heval = H(x0)
      Hinv = inv(Heval)

      # Find the step length
      p0 = Hinv.dot(Jeval)
      dk = -p0

      # alpha = backTrackingLineSearch(x0, F, Jeval, dk, p = 0.5,
         alpha=1, c=1e-4)
      alpha = backTrackingLineSearch(x0, F, Jeval, dk)

      # Calculate the step
      x1 = x0 - alpha*p0
      xStep = xStep + [x1.tolist()]

      # If we found the root (to within the tolerance),
      # return it and 0 for the error message
      if (norm(x1 - x0) < tol):
        xstar = x1
        ier = 0
        return[xstar, ier, its, xStep]

      x0 = x1

   # If we didn't find the root, return the step and an error of 1
   xstar = x1
   ier = 1
   return[xstar, ier, its, xStep]
```

Implementation of **Gradient Descent**:

```
def GradientDescent(x0, F, J, tol, Nmax):

   # Checks to see if the function is already at the root
   if norm(F(x0)) == 0:
      xstar = x0
      ier = 0
      its = 0
      return[xstar, ier, its, x0]


   xStep = [x0]
   for its in range(Nmax):
      # Evaluate J and compute its inverse
      Jeval = J(x0)

      # Find the step length
      dk = -Jeval
      alpha = backTrackingLineSearch(x0, F, Jeval, dk)

      # Calculate the step
      x1 = x0 - alpha * Jeval
      xStep = xStep + [x1.tolist()]


      # If we found the root (to within the tolerance), return it and 0
         for the error message
      if (norm(x1 - x0) < tol):
        xstar = x1
        ier = 0
        return[xstar, ier, its, xStep]

      x0 = x1

   # If we didn't find the root, return the step and an error of 1
   xstar = x1
   ier = 1
   return[xstar, ier, its, xStep]
```

Implementation of **Lazy Newton**:

```
def LazyNewtonDescent(x0, F, J, H, tol, Nmax):

   # Checks to see if the function is already at the root
   if norm(F(x0)) == 0:
      xstar = x0
      ier = 0
      its = 0
      return

   xStep = [x0]

   # Evaluates the Hessian and computes its inverse
   Heval = H(x0)
   Hinv = inv(Heval)

   for its in range(Nmax):
      # Evaluate J
      Jeval = J(x0)

      # Find the step length
      p0 = Hinv.dot(Jeval)
      dk = -p0

      alpha = backTrackingLineSearch(x0, F, Jeval, dk)

      # Calculate the step
      x1 = x0 - alpha*p0
      xStep = xStep + [x1.tolist()]

      # If we found the root (to within the tolerance),
      # return it and 0 for the error message
      if (norm(x1 - x0) < tol):
        xstar = x1
        ier = 0
        return[xstar, ier, its, xStep]

      x0 = x1

   # If we didn't find the root, return the step and an error of 1
   xstar = x1
   ier = 1
   return[xstar, ier, its, xStep]
```

Different functions used (including their gradient and Hessian matrix):

```
## Quadratic
# Quadratic function
def Bowl(x):
   return 6*x[0]**2 + x[1]**2

# Jacobian of the quadratic function
def BowlJ(x):
   J = np.zeros(2)

   J[0] = 12*x[0]
   J[1] = 2*x[1]

   return J

# Hessian of the quadratic function
def BowlH(x):
   H = np.zeros([2,2])

   H[0,0] = 12
   H[1,1] = 2

   return H

## Rosenbrock
# Rosenbrock function
def Rosenbrock(x):
   import numpy as np
   a = 1; b = 100;
   return (a - x[0])**2 + b*(x[1] - x[0]**2)**2

# Jacobian of the Rosenbrock function
def RosenbrockJ(x):
   a = 1; b = 100;

   J = np.zeros(2)

   J[0] = -2*(a - x[0]) - 4*b*x[0]*(x[1] - x[0]**2)
   J[1] = 2*b*(x[1] - x[0]**2)

   return J

# Hessian of the Rosenbrock function
def RosenbrockH(x):
   a = 1; b = 100;

   H = np.zeros([2,2])
```

```
    H[0,0] = 2 + 8*b*x[0]**2 - 4*b*(x[1] - x[0]**2)
    H[0,1] = -4*b*x[0]
    H[1,0] = -4*b*x[0]
    H[1,1] = 2*b

    return H

## Himmelblau
# Himmelblau's function
def Himmelblau(x):
    return (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 - 7)**2

# Jacobian of Himmelblau's function
def HimmelblauJ(x):
    J = np.zeros(2)

    J[0] = 4*x[0]*(x[0]**2 + x[1] - 11) + 2*(x[0] + x[1]**2 - 7)
    J[1] = 2*(x[0]**2 + x[1] - 11) + 4*x[1]*(x[0] + x[1]**2 - 7)

    return J

# Hessian of Himmelblau's function
def HimmelblauH(x):
    H = np.zeros([2,2])

    H[0,0] = 2 + 8*x[0]**2 + 4*(x[1] + x[0]**2 - 11)
    H[0,1] = 4*x[0] + 4*x[1]
    H[1,0] = 4*x[0] + 4*x[1]
    H[1,1] = 2 + 8*x[1]**2 + 4*(x[0] + x[1]**2 - 7)

    return H
```

Code used for **plotting and figure generation**:

```
import matplotlib.pyplot as plt

import numpy as np
from numpy.linalg import norm

from method_functions import GradientDescent, NewtonDescent,
   LazyNewtonDescent
from functions import Rosenbrock, RosenbrockH, RosenbrockJ, Bowl, BowlJ,
   BowlH, Himmelblau, HimmelblauJ, HimmelblauH, Bohachevsky,
   BohachevskyJ, BohachevskyH
from plottingFunctions import PlotHimmelblau

# Initial conditions and parameters
x0 = [-2, -20]; tol = 1e-16; Nmax = 100000

# Finds the roots with Gradient Descent and then returns and prints the
   output
root, error, iterations, GDSteps = GradientDescent(x0, Rosenbrock,
   RosenbrockJ, tol, Nmax)
print("Root:", root, "\n\t Number of iterations:", iterations)

# Finds the roots with Newton Descent and then returns and prints the
   output
root, error, iterations, NDSteps = NewtonDescent(x0, Rosenbrock,
   RosenbrockJ, RosenbrockH, tol, Nmax)
print("Root:", root, "\n\t Number of iterations:", iterations)

# Converts the lists into numpy arrays for plotting
GDSteps = np.array(GDSteps)
NDSteps = np.array(NDSteps)

# Grid for plotting the function
x = np.linspace(-5, 5, 1000)
y = np.linspace(-25, 25, 1000)
X, Y = np.meshgrid(x, y)
evalPoints = [X, Y]

# Plot of the Rosenbrock function
plt.plot(GDSteps[:,0], GDSteps[:,1],
         color='#c61aff',
         marker="o",
         zorder=1,
         label="Gradient Descent")
plt.plot(NDSteps[:,0], NDSteps[:,1],
         color='#008fb3',
         marker="o",
         zorder=1,
         label="Newton Descent")
plt.contour(X, Y, Rosenbrock(evalPoints),
         levels=np.logspace(-5, 5, 10),
         colors='#33334d',
         linewidths=0.75,
         zorder=0)
plt.legend(); plt.title("Rosenbrock function. Root located at" + r"
   (1,1)."); plt.xlabel("x"); plt.ylabel("y")
plt.show()

# Grid for plotting the function
x = np.linspace(-2, 2, 1000)
y = np.linspace(-3.5, 4, 1000)
X, Y = np.meshgrid(x, y)
evalPoints = [X, Y]

# Plot of the Rosenbrock function
plt.plot(GDSteps[14:,0], GDSteps[14:,1],
         color='#c61aff',
         marker="o",
         zorder=1,
         label="Gradient Descent")
plt.plot(NDSteps[1:,0], NDSteps[1:,1],
         color='#008fb3',
         marker="o",
         zorder=1,
         label="Newton Descent")
plt.contour(X, Y, Rosenbrock(evalPoints),
         levels=np.logspace(-5, 4, 20),
         colors='#33334d',
         linewidths=0.75,
         zorder=0)
plt.legend(); plt.title("Rosenbrock function. Root located at" + r"
   (1,1)."); plt.xlabel("x"); plt.ylabel("y")
plt.show()

# Initial conditions and parameters
x0 = [-18, -18]; tol = 1e-16; Nmax = 100000

# Finds the roots with Gradient Descent and then returns and prints the
   output
root, error, iterations, GDSteps = GradientDescent(x0, Bowl, BowlJ, tol,
   Nmax)
print("Root:", root, "\n\t Number of iterations:", iterations)

# Finds the roots with Newton Descent and then returns and prints the
   output
root, error, iterations, NDSteps = NewtonDescent(x0, Bowl, BowlJ, BowlH,
   tol, Nmax)
```

```
print("Root:", root, "\n\t Number of iterations:", iterations)

# Converts the lists into numpy arrays for plotting
GDSteps = np.array(GDSteps)
NDSteps = np.array(NDSteps)

# Grid for plotting the function
x = np.linspace(-20, 20, 1000)
y = np.linspace(-20, 20, 1000)
X, Y = np.meshgrid(x, y)
evalPoints = [X, Y]

# Plot of the Rosenbrock function
plt.plot(GDSteps[:,0], GDSteps[:,1],
         color='#c61aff',
         marker="o",
         label="Gradient Descent",
         zorder=1)
plt.plot(NDSteps[:,0], NDSteps[:,1],
         color='#008fb3',
         marker="o",
         label="Newton Descent",
         zorder=1)
plt.contour(X, Y, Bowl(evalPoints),
         levels=10,
         colors='#33334d',
         zorder=0)
plt.legend(); plt.title("Quadratic function." + r" $f(x,y) = 6x^2 +
   y^2$"); plt.xlabel("x"); plt.ylabel("y")
plt.show()

# Initial conditions and parameters
# x0 = [-2, -20]; tol = 1e-16; Nmax = 100000
x0 = [1, 10]; tol = 1e-16; Nmax = 100000

# Finds the roots with Gradient Descent and then returns and prints the
   output
root, error, iterations, GDSteps = GradientDescent(x0, Rosenbrock,
   RosenbrockJ, tol, Nmax)
print("Root:", root, "\n\t Number of iterations:", iterations)

# Finds the roots with Newton Descent and then returns and prints the
   output
root, error, iterations, NDSteps = NewtonDescent(x0, Rosenbrock,
   RosenbrockJ, RosenbrockH, tol, Nmax)
print("Root:", root, "\n\t Number of iterations:", iterations)

# Converts the lists into numpy arrays for plotting
GDSteps = np.array(GDSteps)
```

```
NDSteps = np.array(NDSteps)

# Grid for plotting the function
x = np.linspace(-5, 10, 1000)
y = np.linspace(-25, 25, 1000)
X, Y = np.meshgrid(x, y)
evalPoints = [X, Y]

# Plot of the Rosenbrock function
plt.plot(GDSteps[:,0], GDSteps[:,1],
         color='#c61aff',
         marker="o",
         zorder=1,
         label="Gradient Descent")
plt.plot(NDSteps[:,0], NDSteps[:,1],
         color='#008fb3',
         marker="o",
         zorder=1,
         label="Newton Descent")
plt.contour(X, Y, Rosenbrock(evalPoints),
         levels=np.logspace(0.5, 35, 40),
         colors='#33334d',
         linewidths=0.75,
         zorder=0)
plt.legend(); plt.title("Rosenbrock function. Root located at" + r"
   (1,1)."); plt.xlabel("x"); plt.ylabel("y")
plt.show()

# Grid for plotting the function
x = np.linspace(0.1, 4, 1000)
y = np.linspace(-1, 12, 1000)
X, Y = np.meshgrid(x, y)

evalPoints = [X, Y]

# Plot of the Rosenbrock function
plt.plot(GDSteps[:,0], GDSteps[:,1],
         color='#c61aff',
         marker="o",
         zorder=1,
         label="Gradient Descent")
plt.plot(NDSteps[:,0], NDSteps[:,1],
         color='#008fb3',
         marker="o",
         zorder=1,
         label="Newton Descent")
plt.contour(X, Y, Rosenbrock(evalPoints),
         levels=np.logspace(0.5, 5, 40),
         colors='#33334d',
          linewidths=0.75,
          zorder=0)
plt.legend(); plt.title("Rosenbrock function. Root located at" + r"
    (1,1)."); plt.xlabel("x"); plt.ylabel("y")
plt.show()

# Grid for plotting the function
x = np.linspace(-10, 10, 1000)
y = np.linspace(-25, 25, 1000)
X, Y = np.meshgrid(x, y)

evalPoints = [X, Y]

plt.contour(X, Y, Himmelblau(evalPoints),
          levels=np.logspace(-20, 6, 50),
          colors='#33334d',
          linewidths=0.75,
          zorder=0)

x0 = [-4, -20];
PlotHimmelblau(x0);
plt.show()

plt.contour(X, Y, Himmelblau(evalPoints),
          levels=np.logspace(-20, 6, 50),
          colors='#33334d',
          linewidths=0.75,
          zorder=0)

x0 = [-5, 20]
PlotHimmelblau(x0);
plt.show()

plt.contour(X, Y, Himmelblau(evalPoints),
          levels=np.logspace(-20, 6, 50),
          colors='#33334d',
          linewidths=0.75,
          zorder=0)

x0 = [7, 20];
PlotHimmelblau(x0);
plt.show()

plt.contour(X, Y, Himmelblau(evalPoints),
          levels=np.logspace(-20, 6, 50),
          colors='#33334d',
          linewidths=0.75,
          zorder=0)
```

```
x0 = [7, -20];
PlotHimmelblau(x0);
plt.show()

# Initial conditions and other parameters
x0 = [2, 2]; tol = 1e-16; Nmax = 100000

# Finds the roots with Gradient Descent and then returns and prints the
   output
root, error, iterations, GDSteps = GradientDescent(x0, Rosenbrock,
   RosenbrockJ, tol, Nmax)
print("Root:", root, "\n\t Number of iterations:", iterations)

# Finds the roots with Newton Descent and then returns and prints the
   output
root, error, iterations, NDSteps = NewtonDescent(x0, Rosenbrock,
   RosenbrockJ, RosenbrockH, tol, Nmax)
print("Root:", root, "\n\t Number of iterations:", iterations)

# Converts the lists into numpy arrays for plotting
GDSteps = np.array(GDSteps)
NDSteps = np.array(NDSteps)

# Choice of the true value of the root
x_root = 1; y_root = 1

# Calculation of the error for each of the axes
GD_error = np.array([abs(x_root - GDSteps[:,0]), abs(y_root -
   GDSteps[:,1])])
ND_error = np.array([abs(x_root - NDSteps[:,0]), abs(y_root -
   NDSteps[:,1])])

# Calculates the norm distance for each of the iterations
GD_error = np.sum(np.abs(GD_error)**2,axis=0)**(1./2)
ND_error = np.sum(np.abs(ND_error)**2,axis=0)**(1./2)

# Log plot of the error for Gradient Descent
plt.semilogy(GD_error,
          color="#c61aff",
          linewidth=3)
plt.xlabel("Number of iterations"); plt.ylabel("Log(|error|)");
   plt.title("Rate of convergence for Gradient Descent.")
plt.show()

# Log plot of the error for Gradient Descent
plt.semilogy(ND_error,
          color="#008fb3",
          linewidth=3)
plt.xlabel("Number of iterations"); plt.ylabel("Log(|error|)");
```

```
   plt.title("Rate of convergence for Newton Descent.")
plt.show()

# Plot of the Rosenbrock function
plt.plot(GDSteps[:,0], GDSteps[:,1],
         color='#c61aff',
         marker="o",
         zorder=1,
         label="Gradient Descent")
plt.plot(NDSteps[:,0], NDSteps[:,1],
         color='#008fb3',
         marker="o",
         zorder=1,
         label="Newton Descent")

# Grid for plotting the function
x = np.linspace(-3, 5, 1000)
y = np.linspace(-1, 17, 1000)
X, Y = np.meshgrid(x, y)
evalPoints = [X, Y]

plt.contour(X, Y, Rosenbrock(evalPoints),
         levels=np.logspace(-5, 5, 20),
         colors='#33334d',
         linewidths=0.75,
         zorder=0)
plt.legend(); plt.title("Rosenbrock function. Root located at" + r"
   (1,1)."); plt.xlabel("x"); plt.ylabel("y")
plt.show()


# Plot of the Rosenbrock function
plt.plot(GDSteps[300:,0], GDSteps[300:,1],
         color='#c61aff',
         marker="o",
         zorder=1,
         label="Gradient Descent")
plt.plot(NDSteps[10:,0], NDSteps[10:,1],
         color='#008fb3',
         marker="o",
         zorder=1,
         label="Newton Descent")


# Grid for plotting the function
x = np.linspace(0.8, 2.4, 1000)
y = np.linspace(0, 5, 1000)
X, Y = np.meshgrid(x, y)
evalPoints = [X, Y]
```

```
plt.contour(X, Y, Rosenbrock(evalPoints),
         levels=np.logspace(-4, 3, 20),
         colors='#33334d',
         linewidths=0.75,
         zorder=0)
plt.legend(); plt.title("Rosenbrock function. Root located at" + r"
   (1,1)."); plt.xlabel("x"); plt.ylabel("y")
plt.show()

# Initial conditions and other parameters
x0 = [2, 2]; tol = 1e-16; Nmax = 100000

# Finds the roots with Gradient Descent and then returns and prints the
   output
root, error, iterations, LNDSteps = LazyNewtonDescent(x0, Rosenbrock,
   RosenbrockJ, RosenbrockH, tol, Nmax)
print("Root:", root, "\n\t Number of iterations:", iterations)

# Finds the roots with Newton Descent and then returns and prints the
   output
root, error, iterations, NDSteps = NewtonDescent(x0, Rosenbrock,
   RosenbrockJ, RosenbrockH, tol, Nmax)
print("Root:", root, "\n\t Number of iterations:", iterations)

# Converts the lists into numpy arrays for plotting
LNDSteps = np.array(LNDSteps)
NDSteps = np.array(NDSteps)

# Choice of the true value of the root
x_root = 1; y_root = 1

# Calculation of the error for each of the axes
LGD_error = np.array([abs(x_root - LNDSteps[:,0]), abs(y_root -
   LNDSteps[:,1])])
ND_error = np.array([abs(x_root - NDSteps[:,0]), abs(y_root -
   NDSteps[:,1])])

# Calculates the norm distance for each of the iterations
LGD_error = np.sum(np.abs(LGD_error)**2,axis=0)**(1./2)
ND_error = np.sum(np.abs(ND_error)**2,axis=0)**(1./2)

# Log plot of the error for Gradient Descent
plt.semilogy(LGD_error,
          color="#9933ff",
          linewidth=3)
plt.xlabel("Number of iterations"); plt.ylabel("Log(|error|)");
   plt.title("Rate of convergence for Lazy Newton Descent.")
plt.show()
```

```
# Log plot of the error for Gradient Descent
plt.semilogy(ND_error,
          color="#008fb3",
          linewidth=3)
plt.xlabel("Number of iterations"); plt.ylabel("Log(|error|)");
   plt.title("Rate of convergence for Newton Descent.")
plt.show()

# Plot of the Rosenbrock function
plt.plot(LNDSteps[:,0], LNDSteps[:,1],
         color='#9933ff',
         marker="o",
         zorder=1,
         label="Lazy Newton Descent")
plt.plot(NDSteps[:,0], NDSteps[:,1],
         color='#008fb3',
         marker="o",
         zorder=1,
         label="Newton Descent")

# Grid for plotting the function
x = np.linspace(-3, 5, 1000)
y = np.linspace(-1, 17, 1000)
X, Y = np.meshgrid(x, y)
evalPoints = [X, Y]

plt.contour(X, Y, Rosenbrock(evalPoints),
         levels=np.logspace(-5, 5, 20),
         colors='#33334d',
         linewidths=0.75,
         zorder=0)
plt.legend(); plt.title("Rosenbrock function. Root located at" + r"
   (1,1)."); plt.xlabel("x"); plt.ylabel("y")
plt.show()


# Plot of the Rosenbrock function
plt.plot(LNDSteps[:,0], LNDSteps[:,1],
         color='#9933ff',
         marker="o",
         zorder=1,
         label="Lazy Newton Descent")
plt.plot(NDSteps[:,0], NDSteps[:,1],
         color='#008fb3',
         marker="o",
         zorder=2,
         label="Newton Descent")
```

```
# Grid for plotting the function
x = np.linspace(0.8, 2.4, 1000)
y = np.linspace(0, 5, 1000)
X, Y = np.meshgrid(x, y)
evalPoints = [X, Y]

plt.contour(X, Y, Rosenbrock(evalPoints),
         levels=np.logspace(-4, 3, 20),
         colors='#33334d',
         linewidths=0.75,
         zorder=0)
plt.legend(); plt.title("Rosenbrock function. Root located at" + r"
   (1,1)."); plt.xlabel("x"); plt.ylabel("y")
plt.show()

# Grid for plotting the function
x = np.linspace(-10, 10, 1000)
y = np.linspace(-25, 25, 1000)
X, Y = np.meshgrid(x, y)

evalPoints = [X, Y]

plt.contour(X, Y, Himmelblau(evalPoints),
         levels=np.logspace(-20, 6, 50),
         colors='#33334d',
         linewidths=0.75,
         zorder=0)

x0 = [6, 20];
PlotHimmelblau(x0);
plt.show()
```