Quasi-Newton Methods for Optimization

Walter Virany, Ryan Foley

May 9, 2023

1 Abstract

Newton descent is an optimization method that uses second-order derivative information to iteratively find the minimum of a function. Newton descent has the advantage of a unit step length, which can be expensive to compute in other descent methods. In this paper, we derive the Newton direction and develop the corresponding descent method. We show that, under certain conditions, the Newton descent method converges quadratically to the minimum of a function. Furthermore, we explore variations of Newton descent which trade off the quadratic rate of convergence for improved computational expense.

2 Introduction

Optimization is a fundamental problem in many areas of science, engineering, and mathematics. Given a function, the goal of optimization is to find the input that results in an optimum. One commonly used family of optimization algorithms are gradient-based descent methods, which use the first-order derivative to iteratively search for an optimal solution. However, methods like steepest descent suffer from potential drawbacks. For example, gradient-based methods can get stuck in local optima or converge slowly.

To overcome these issues, second-order methods have been developed that use the Hessian matrix to inform the optimization process. One such method is Newton descent, which has the advantage over steepest descent because it uses a natural unit step length. A class of methods called "Quasi-Newton" methods have further been developed to improve upon Newton descent. These methods seek to improve the efficiency and computational cost of Newton's method, which can prove to be a problem for large systems of equations.

In this paper, we derive the Newton direction in descent, which allows us to formulate our optimization method. Then, we analytically show that Newton descent converges quadratically and verify the convergence empirically. Furthermore, we explore variations of the Newton descent method, including "Lazy Newton" descent, the Broyden-Fletcher-Goldfarb-Shannon rank-one update (BFGS) formula, the Davidon-Fletcher-Powell (DFP) formula, and the Symmetric-Rank-One (SR1) method and compare performance. We find that each of these quasi-Newton methods has its own set of advantages and disadvantages, but in general prove to be more efficient in practice because they avoid the direct computation of the inverse of the Hessian matrix.

3 Mathematical Formulation

3.1 Derivation of Newton Direction

Suppose we want to minimize some continuously differentiable function $f : \mathbb{R}^n \to \mathbb{R}$. The first-order Taylor expansion about x_k gives

$$f(x_k + \alpha p) = f(x_k) + \nabla f(x_k + \alpha p)^T p$$

where α is the step size we take and p is the descent direction. The rate of change of f along the direction p at x_k is given by the dot product $\nabla f(x_k)^T p$. Thus, we can minimize this rate of change to obtain the unit direction of most rapid decrease. To do so, we consider this dot product as

$$|\nabla f(x_k)||p|\cos\theta = |\nabla f(x_k)|\cos\theta$$

where θ is the angle between the two vectors and $|p| = 1$. This product is minimized when $\cos\theta = -1$. That is, when $\theta = \pi$ and $\nabla f(x_k)$ and p point in opposite directions. This gives our direction of steepest descent, $-\nabla f(x_k)$.

Now, we consider the quadratic model $m_k(p)$ of $f(x_k + \alpha p)$,

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

where $\nabla^2 f(x_k)$ is our Hessian matrix. Differentiating with respect to p, we get

$$\nabla f(x_k) + \nabla^2 f(x_k + \alpha p) p$$

Setting this equal to zero and solving for p, we retrieve the Newton direction:

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

Thus, we define the Newton step:

$$x_{k+1} = x_k + p$$

3.2 Convergence

In order to understand the limits of an optimization algorithm, we need to study its convergence properties. We begin by considering the Newton step:

$$\begin{aligned} x_{k+1} &= x_k + p \\ x_{k+1} - x^* &= x_k - x^* - \nabla^2 f_k^{-1} \nabla f_k \\ &= \nabla^2 f_k^{-1}[\nabla^2 f_k(x_k - x^*) - (\nabla f_k - \nabla f_*)] \end{aligned}$$

where $\nabla f_* = 0$. By Taylor's theorem, we have that

$$\nabla f_k - \nabla f_* = \int_0^1 \nabla^2 f_k(x_k + t(x^* - x_k))(x_k - x^*)dt$$

Thus,

$$\begin{aligned} ||\nabla^2 f(x_k)(x_k - x^*) - (\nabla f_k - \nabla f(x^*))|| &= \left|\left| \int_0^1 [\nabla^2 f(x_k) - \nabla^2 f(x_k + t(x^* - x_k))](x_k - x^*)dt \right|\right| \\ &\leq \int_0^1 ||\nabla^2 f(x_k) - \nabla^2 f(x_k + t(x^* - x_k))|| ||x_k - x^*|| dt \\ &\leq ||x_k - x^*||^2 \int_0^1 Lt dt \\ &= \frac{1}{2} L ||x_k - x^*||^2 \end{aligned}$$

where L is the Lipschitz constant for $\nabla^2 f(x)$ for x near x^*. Thus, we get

$$||x_{k+1} - x^*|| \leq L||\nabla^2 f(x^*)^{-1}|| ||x_k - x^*||^2 = \tilde{L}||x_k - x^*||^2$$

where $\tilde{L} = L||\nabla^2 f(x^*)^{-1}||$. Thus, we see that the sequence converges to x^* quadratically [1].

3.3 Implementation

Now, we are able to define an algorithm to find the minimum of a function using Newton descent. Given an initial guess x_0, function f, max number of iterations N, and convergence tolerance t, we define our procedure as follows:

Algorithm 1 Newton Descent

```
procedure NEWTON DESCENT($x_0, f, N, t$)
    $x \leftarrow x_0$
    $n \leftarrow 0$
    while $n < N$ do
        $g \leftarrow \nabla f(x)$
        $H \leftarrow \nabla^2 f(x)$
        $p \leftarrow -H^{-1}g$
        $x \leftarrow x + p$
        if $||p||_2 < t$ then return $x, f(x)$
        end if
    end while
end procedure
```

In order to analyze the performance of this descent algorithm, we test it on the Rosenbrock function, a class of functions defined in N variables as

$$f(\mathbf{x}) = f(x_1, x_2, \ldots, x_N) = \sum_{i=1}^{N-1}[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2]$$

which has a global minimum at $x = (1, 1, 1)$ inside a long, narrow "valley", making convergence difficult for many optimization methods. We test our method on this function for $N = 3$. Figure 1 shows the plot of the log of the error at each iterate on one run of Newton Descent starting with the initial guess $x_0 = (-1.2, 1.0, 0.5)$. We observe that our sequence converges quadratically to twelve digits of precision, our chosen tolerance.

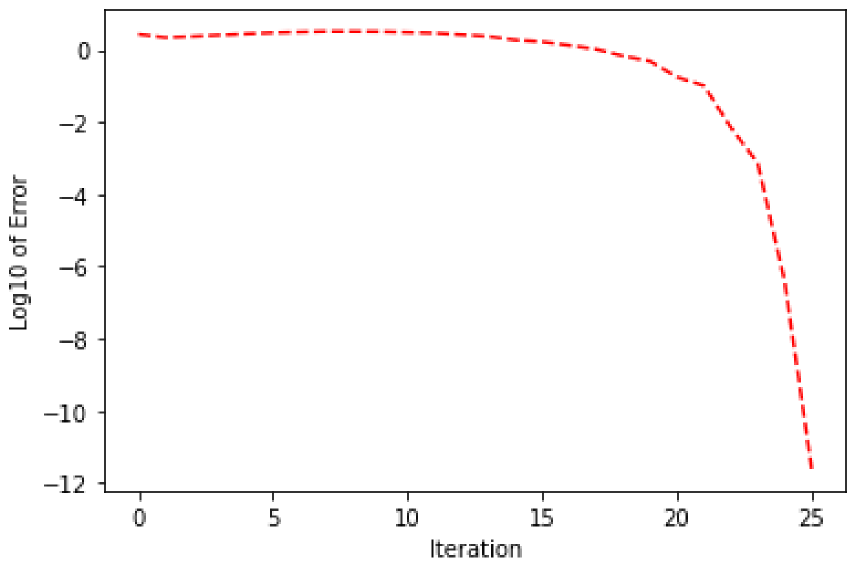

Figure 1: Log base 10 of the error at each iterate.

Table 1 shows the gradient norm for the last 5 iterations of the method. Thus, we can readily see the quadratic increase in digits of precision.

Newton Gradient Norm
1.00589023e-01
3.07914511e-01
7.23444412e-04
1.76441488e-05
2.62150533e-12

Table 1: Gradient norm of converging iterates for Newton's method

4 Quasi-Newton Methods

Although Newton descent proves to be a robust method for optimization with regard to its rate of convergence, it is computationally expensive to compute the Hessian at each iteration and then solve the corresponding linear system. Quasi-Newton methods offer an alternative to Newton's method in which they do not require the computation of the Hessian matrix at each iteration; however, this comes at the sacrifice of the quadratic rate of convergence. For most practical applications to large systems of equations, this trade off can be quite favorable. Many robust methods have been developed that can achieve a superlinear rate of convergence. Such methods include the Broyden-Fletcher-Goldfarb-Shannon (BFGS) method, the closely related DFP algorithm, and the symmetric-rank-1 (SR1) update formula.

4.1 Lazy Newton

In considering how to reduce the cost of computing the Hessian matrix, one fairly simple idea is to avoid updating it altogether. This method is referred to as "Lazy Newton", in which we only compute the Hessian once and use this initial Hessian at all iterations. Thus, our modified Newton step takes the form

$$x_{k+1} = x_k - (\nabla^2 f(x_0))^{-1} \nabla f(x_k).$$

This greatly reduces the computational cost of the descent algorithm, but simultaneously reduces the likelihood of convergence. In reference to our previous test, we examine the performance of Lazy Newton applied to the Rosenbrock function in three variables. Figure 2 shows the error sequence of Lazy Newton on the same plot as that of Newton's method.

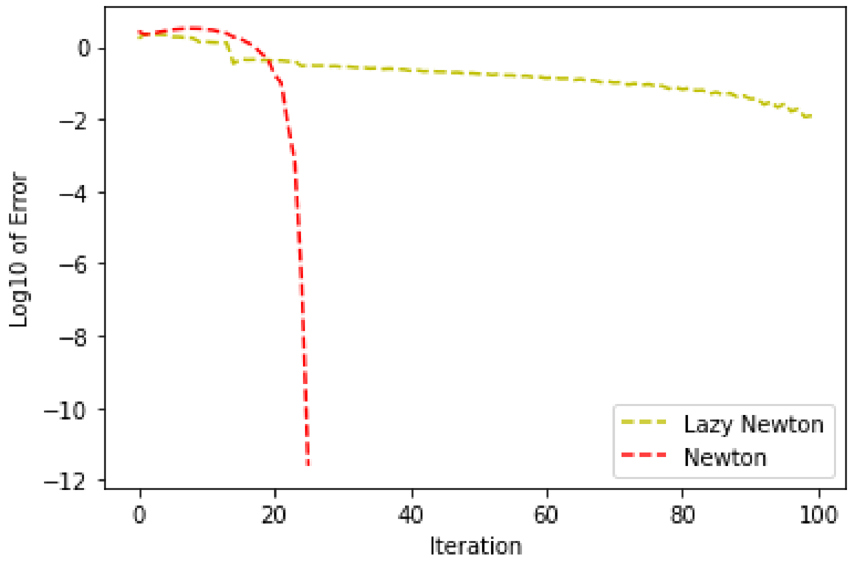

Figure 2: Log of error sequences for Lazy Newton (yellow) and Newton (red)

We observe that Lazy Newton starts converging at a similar rate to Newton, as to be expected. However, it cannot maintain it's rate of convergence, and after one hundred iterations it only manages to obtain two digits of precision. In order to make Lazy Newton work, we require a slightly more sophisticated approach.

4.1.1 Slacker Newton

While Lazy Newton might be sufficient for optimizing more ideal functions, it doesn't manage to converge when applied to the Rosenbrock function in three dimensions. In order to improve upon Lazy Newton, we consider a method in which we update the Hessian based on a certain condition. For example, we could compute the Hessian at x_k when k is a multiple of some natural number. Alternatively, we can implement a condition which updates the Hessian when we observe Lazy Newton failing to converge. Such a condition could be when the distance between two iterates is not sufficiently large. We unoriginally call this method "Slacker Newton". Figure 3 shows the error sequence of Slacker Newton in comparison to our previous two methods. Starting with the same initial guess as the previous two methods, we find that it does manage to converge to the root superlinearly after about seventy-five iterations. Although it takes many more iterations to converge, Slacker Newton has the advantage of computing about half as many Hessian updates as Newton.

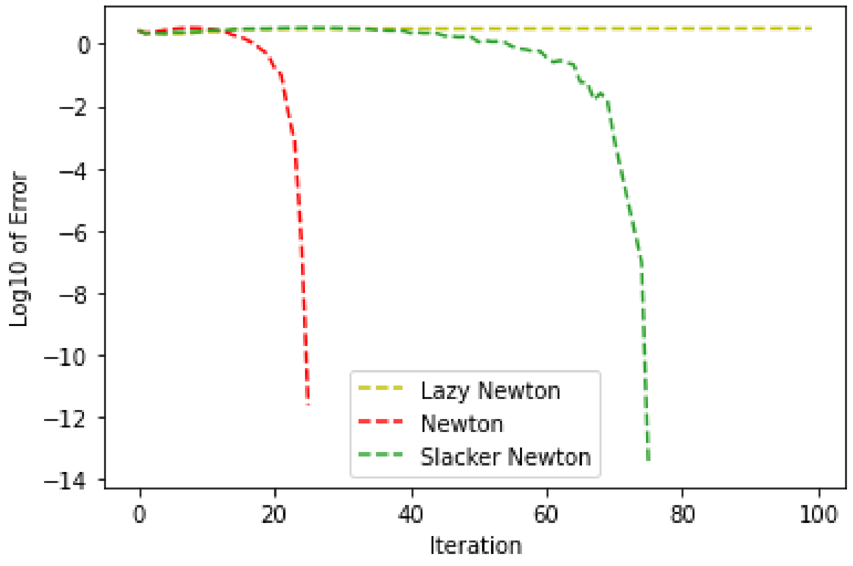

Figure 3: Error sequence of Slacker Newton (green), compared to Newton and Lazy Newton

4.2 Notes on Higher Dimensional Secant Methods

4.2.1 The Secant Formula

The following quasi-Newton methods are all based around a higher dimensional implementation of the secant method. The secant equation for higher dimensions is

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

Where $y_k = \nabla f_{k+1} - \nabla f_k$ and $s_k = x_{k+1} - x_k$. All of the update matrices B_k in the following methods follow this rule. This helps to maintain information about curvature calculated in the previous step, one of the main features of Newton descent.

4.2.2 The Sherman–Morrison Formula

The Sherman-Morrison Formula is used to find an update formula for the inverse of a matrix given a matrices update formula. This is incredibly useful in quasi-Newton methods since there is often a computational advantage in using the inverse of B_k. The following quasi-Newton methods and their update matrices B_k all meet the conditions to use the Sherman-Morrison Formula.

4.2.3 The Wolfe Conditions

The Wolfe conditions are a method for choosing an optimal step length for the Newton or quasi-Newton step. After an initial guess for α, typically 1, these two rules are looped through until neither applies or a maximum amount of iterations is reached. The first is if $f(x_k + \alpha * p) > f(x_k) + c_1 * \alpha * g(x_k)$, then divide α by two. Note that p is the original step, $g(x_k)$ is the gradient of the function at x_k, and c_1 is some number between zero and one. The second condition is if $g(x_k + \alpha * p)p < c2$, then multiply α by 2.

4.3 The BFGS Method

One of the most widely used quasi-Newton methods is the Broyden-Fletcher-Goldfarb-Shannon (BFGS) method. The BFGS method can be shown to have a superlinear rate of convergence under certain conditions. Consider the quadratic approximation of a function f:

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

where B_k is an n x n symmetric positive definite matrix. Similar to Newton's method, we get

$$p_k = -B_k^{-1} \nabla f_k$$

as the descent direction and our step is defined as

$$x_{k+1} = x_k + \alpha p$$

for some step length α. This follows the from the same formulation as Newton's method; however, using the BFGS method, B_k is an approximation of the Hessian. Once again, our goal is to avoid computing the Hessian at each iteration. So, instead of computing the Hessian at each iteration, we seek another method to update B_k. The BFGS update is defined as

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}.$$

In implementation, it is useful to define

$$H_k = B_k^{-1}$$

so that the descent direction can be computed by straightforward matrix multiplication instead of solving a linear system at each step. The update formula for H_k is

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

One might ask the question: what should the initial Hessian approximate be? There is no universally correct method for determining the initial approximate, but in practice some scalar multiple of the identity matrix is generally chosen.

Equipped with our formulation of the BFGS method, we now have the tools to implement it and compare its performance to the original Newton method. Figure 4 shows the error sequences of the BFGS method and the Newton method once again applied to the Rosenbrock function in three variables. We find that, although it doesn't accomplish the same quadratic rate of convergence as the Newton method, it is much more successful in converging to the minimum than the previous quasi-Newton methods discussed.

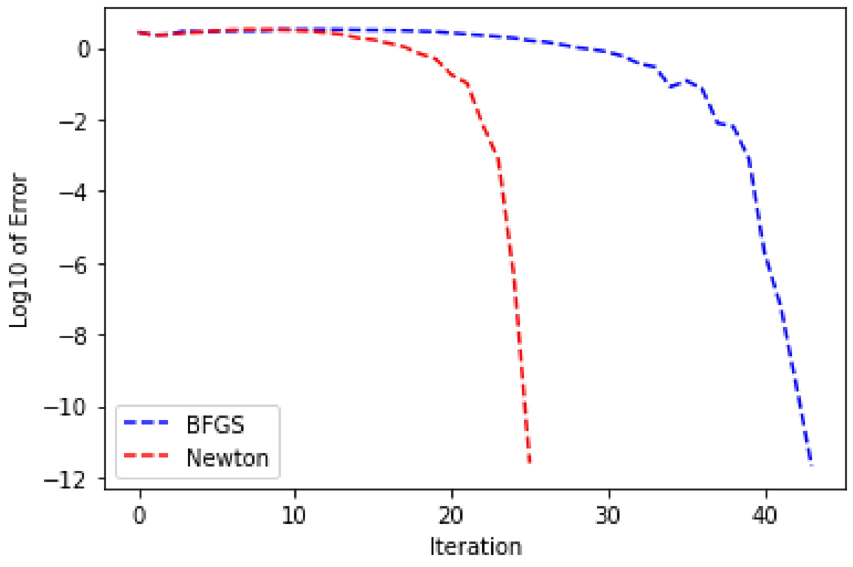

Figure 4: Error sequence of BFGS method (blue) in contrast to Newton's method (red)

Table 2 shows the gradient norms at the last few iterations of the BFGS method.

BFGS Gradient Norm
1.11429422e-01
2.30577981e-02
3.47250962e-03
2.45359492e-04
7.15258422e-06
2.60449102e-07
8.76176550e-10
7.06660575e-12
4.96506831e-16

Table 2: Gradient norm of converging iterates for BFGS method

4.3.1 DFP

The DFP method is a close relative to the BFGS method, and was actually developed first [3]. The DFP update formula is defined as

$$B_{k+1} = (I - \rho_k y_k s_k^T) B_k (I - \rho_k s_k y_k^T) + \rho_k y_k y_k^T,$$

with the corresponding inverse $H_k = B_k^{-1}$ defined as

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

As DFP is not quite as robust as BFGS, we find that it does not converge to the minimum of the Rosenbrock function when constrained to one hundred maximum iterations. Figure 5 demonstrates this failure to converge.

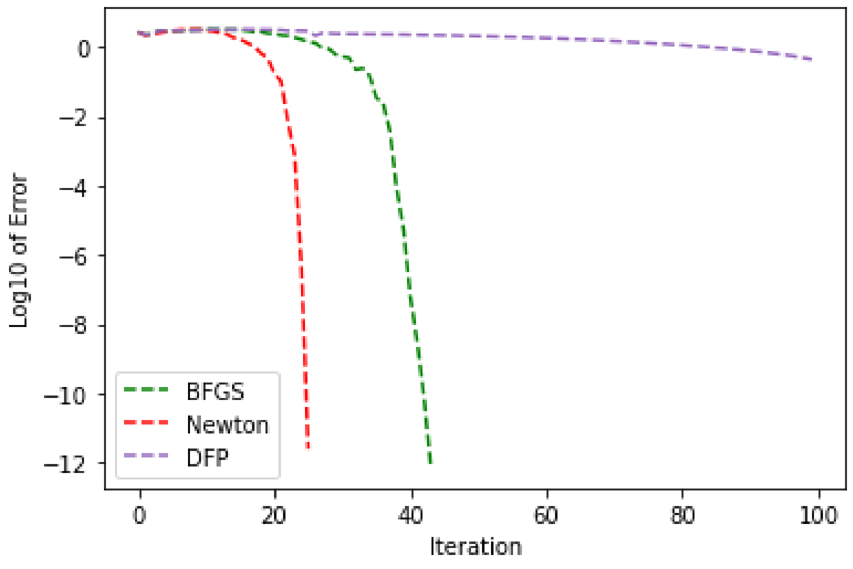

Figure 5: DFP (purple) in comparison to BFGS and Newton

Interestingly, DFP does actually manage to find a superlinear rate of convergence after roughly 1300 iterations. Figure 6 demonstrates this.

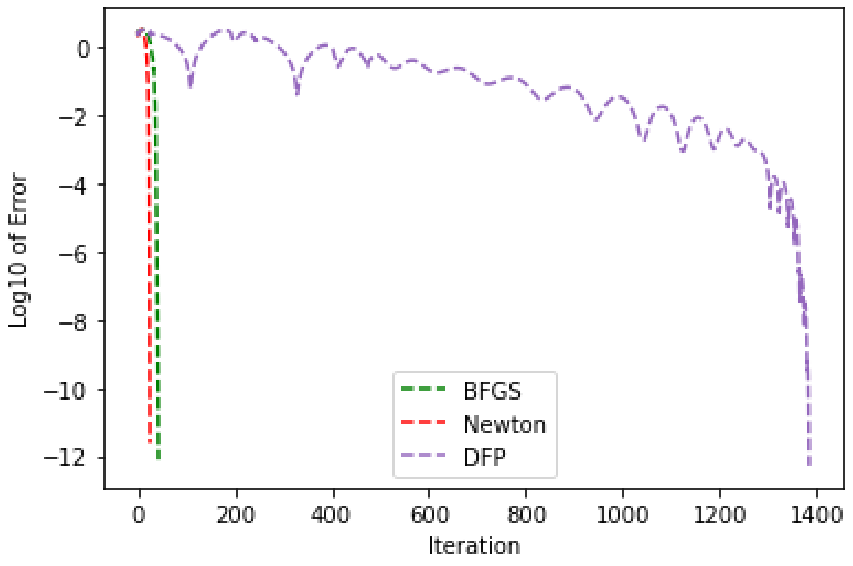

Figure 6: DFP (purple) successful convergence after 1300 iterations

4.4 SR1

The last quasi-Newton method to be analyzed is the Symmetric-Rank-One (SR1) update formula. Similar to the other secant methods discussed, the descent direction is defined as

$$p_k = -B_k^{-1} \nabla f_k.$$

However, in this case B_k is not necessarily a positive definite matrix. In the past, this was seen as a major drawback for the method. However, modern methods of path searching have yielded quite positive results with the method. One of these methods will be explored in the second implementation of SR1. In this implementation, the step is defined by

$$x_{k+1} = x_k + \alpha p.$$

Once again, B_k is an approximation of the hessian that fits all update goals outlined for secant method generalizations. To find s_k, we initially used the Wolfe conditions. However, since this failed to converge, we opted for a different method of selecting s_k. The Trust Region method defines an area around the current iterate in which it trusts that the model bears a heavy resemblance to the function. Then, it chooses a step that minimizes the function within that region. This is an effective form of step choice for a B_k that is not necessarily positive definite, however it does require several implementation notes. The first is while solving for the step is often a trivial linear algebra problem, a requirement of the method is that the norm of s_k is less then the radius of the trust region or ∇_k. To insure this we used the Cauchy point calculation. The Cauchy point states

$$s_k = -\tau \frac{\nabla_k}{|g_k|} g_k,$$

where g_k is the current gradient. When $g_k^T B_k g_k \leq 0$, $\tau = 1$. Otherwise $\tau = min(\frac{|g_k|^3}{\nabla_k g_k^T B_k g_k}, 1)$. For other implementation notes, ensure that your algorithm checks for the reliability of the trust region. These checks can include not updating x_k if the method is not accurate or growing or shrinking the trust region to better represent a realistic expectation.

The B_k update is $B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s k_k)^T s_k}$. To increase the stability of this algorithm, we skip the B_k update when the update wouldn't be positive definite or the denominator of the update is within some tolerance close to zero. We tested the SR1 update on the Rosenbrock function with the results demonstrated in Figure 7.

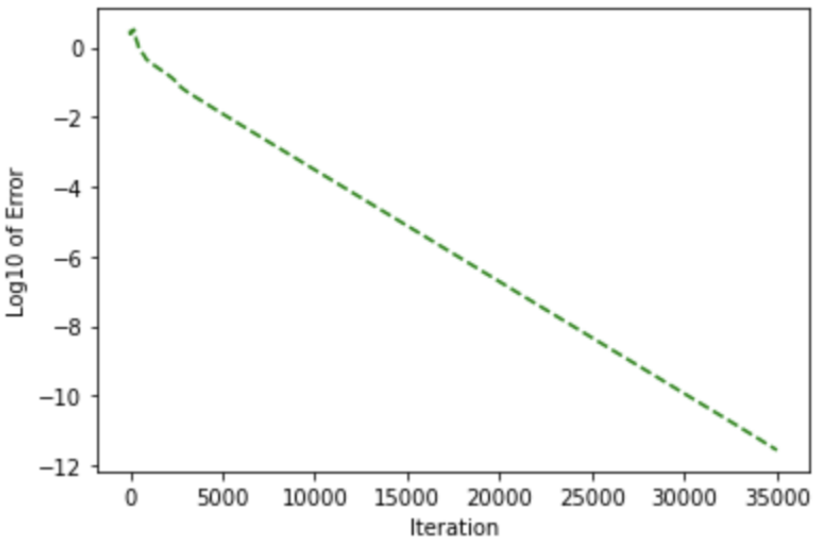

Figure 7: SR1 convergence after 35,000 iterations

Our implementation of the SR1 quasi-Newton method demonstrated linear convergence on the Rosenbrock function and converged to 12 digits of precision after 35,000 iterations. We attribute this extreme number of iterations to the lack of robustness of SR1 in comparison to BFGS and DFP, and also to the extreme behavior of the Rosenbrock function.

5 Discussion

5.1 Conclusions

In this paper, we derived the Newton direction in descent, which was based on the quadratic approximation of our function f at each iterate x_k. This was a natural extension of the steepest descent method, for which the descent direction is derived from the linear approximation of f. We then showed that, under certain conditions, Newton's method converges to the minimum of f at a quadratic rate of convergence.

Furthermore, we demonstrated its quadratic rate of convergence on the Rosenbrock function in three variables. While steepest descent failed to converge to the minimum of the Rosenbrock function, Newton successfully did so in only twenty-five iterations.

However, computing the Hessian inverse at each iteration of Newton's method can be too computationally expensive for practical purposes. We addressed this issue by delving into various quasi-Newton methods, which each have their own procedures for avoiding the inversion of the Hessian. For example, the "Lazy Newton" method computes the Hessian only once and repeatedly uses it at each iteration. Another approach is to compute an approximate Hessian, which is the motivation behind the BFGS, DFP, SR1 methods, and a much larger class of methods known as the Broyden class.

Although these methods have the advantage of efficiency, that comes at a cost of the rate of convergence. In this paper, we were able to demonstrate the superlinear rates of convergence of BFGS and DFP, and we found that BFGS was, given our function, the strongest quasi-Newton method we implemented. Nevertheless, each method has its own advantages and disadvantages, and determining which method is best for a given application can be regarded as a form of art.

5.2 Future Work

In future work, we could explore more quasi-Newton methods and implement our methods on a larger variety of functions. This would allow for a deeper understanding of the differences in performance and which methods are optimal for different applications. Furthermore, we could apply our methods to real-world problems [4], and test them against industry standards. Alternatively, we could explore the case in which a function has a Hessian matrix which is invertible but not positive definite.

6 References

[1] Nocedal, J., &; Wright, S. J. (2006). Chapter 3: Line Search Methods. In Numerical optimization. essay, Springer.

[2] Nocedal, J., &; Wright, S. J. (2006). Chapter 6: Quasi-Newton Methods. In Numerical optimization. essay, Springer.

[3] W. C. DAVIDON,Variable metric method for minimization, Technical Report ANL–5990 (revised), Argonne National Laboratory, Argonne, IL, 1959.

[4] Battiti, R., Masulli, F. (1990). BFGS Optimization for Faster and Automated Supervised Learning. In: International Neural Network Conference. Springer, Dordrecht.

7 Appendix

Rosenbrock Function:

```
def rosenbrock(x):
N = len(x)
sum = 0

for i in range(N-1):
    sum += 100 * (x[i+1] - x[i]**2)**2 + (1 - x[i])**2

return sum
```

```
'''
Returns the gradient of the Rosenbrock function in 3 variables at x
'''
def grad_rosenbrock_3D(x):
    grad = np.zeros(3)
    grad[0] = -400 * x[0] * (-x[0]**2 + x[1]) + 2 * (x[0] - 1)
    grad[1] = -400 * x[1] * (-x[1]**2 + x[2]) + 200 * (x[1] - x[0]**2) + 2 * (x[1] - 1)
    grad[2] = 200 * (-x[1]**2 + x[2])
    return grad


'''
Returns the hessian of the Rosenbrock function in 3 variables at x
'''
def hess_rosenbrock_3D(x):
    hess = np.zeros((3,3))
    hess[0, 0] = 1200*x[0]**2 - 400*x[1] + 2
    hess[0, 1] = -400*x[0]
    hess[1, 0] = -400*x[0]
    hess[1, 1] = 1200*x[1]**2 - 400*x[2] + 202
    hess[1, 2] = -400*x[1]
    hess[2, 1] = -400*x[1]
    hess[2, 2] = 200
    return hess
```

Newton Descent Implementation:

```
    def wolfe_conditions(x, f, grad, p, alpha=1.0, c1=0.25, c2=0.75, max_iters=100):
    t = 1.0
    fx = f(x)
    gxp = grad(x).dot(p)

    for i in range(max_iters):
        if f(x + t * p) > fx + c1 * t * gxp:
            t /= 2.0
        elif grad(x + t * p).dot(p) < c2 * gxp:
            t *= 2.0
        else:
            break

    return t

    def newton_descent(x0, f, grad, hess, tol=1e-6, Nmax=100):

    x = x0.copy()
    x_seq = np.zeros((Nmax+1, len(x0)))
    x_seq[0] = x
    g_seq = np.zeros((Nmax+1, len(x0)))
    n = 0
    while n < Nmax:

        g = grad(x)
        g_seq[n] = np.linalg.norm(g)

        H = hess(x)
        p = -np.linalg.solve(H, g)

        t = wolfe_conditions(x, f, grad, p)

        x1 = x + t*p

        if np.linalg.norm(p) < tol:
            return x1, x_seq, g_seq, n

        n += 1

        x_seq[n] = x1

        x = x1
```

Lazy Newton Implementation:

```
    def lazy_newton(x0, f, grad, hess, tol=1e-6, Nmax=100):

    x = x0.copy()
    x_seq = np.zeros((Nmax+1, len(x0)))
    x_seq[0] = x
    n = 0

    H = hess(x) # Computing the initial Hessian

    while n < Nmax:

        g = grad(x)

        # For Slacker Newton, update Hessian every few iterations
#         if n % 4 == 0:
#             H = hess(x)

        p = -np.linalg.solve(H, g)

        t = wolfe_conditions(x, f, grad, p)

        x1 = x + t * p

        x_seq[n+1] = x1

        if np.linalg.norm(p) < tol:
            return x, x_seq, n
        n += 1

        x = x1

    print("Max iterations reached")
    return x, x_seq, n
```

BFGS Implementation:

```
    def BFGS(x0, f, grad, hess, tol=1e-6, Nmax=100):

    x = x0.copy()
    x_seq = np.zeros((Nmax+1, len(x0)))
    x_seq[0] = x
    g_seq = np.zeros((Nmax+1, len(x0)))
    n = 0

    H = np.eye(3)

    while n < Nmax:

        g = grad(x)
        g_seq[n] = np.linalg.norm(g)

        if np.linalg.norm(g) < tol:
            return x, x_seq, g_seq, n

        p = -np.dot(H, g)

        t = wolfe_conditions(x, f, grad, p)

        x1 = x + t * p

        x_seq[n+1] = x1

        s = x1 - x
        y = grad(x1) - g
        rho = 1 / np.dot(y, s)
        I = np.eye(len(x))

        # Compute update for Hessian:
        u1 = I - rho * np.outer(s,y)
        u2 = I - rho * np.outer(y,s)

        U = np.matmul(H,u2)

        H = np.matmul(u1, U) + rho * np.outer(s, s)

        x = x1
        n +=1

    print("Max Iterations Reached")
    return x, x_seq, g_seq, n
```

DFP Implementation:

```
    def DFP(x0, f, grad, hess, tol=1e-6, Nmax=100):

    x = x0.copy()
    x_seq = np.zeros((Nmax+1, len(x0)))
    x_seq[0] = x
    n = 0

    B = np.eye(3)

    while n < Nmax:

        g = grad(x)

        if np.linalg.norm(g) < tol:
            return x, x_seq, n

        p = -np.linalg.solve(B, g)

        t = wolfe_conditions(x, f, grad, p)

        x1 = x + t * p

        x_seq[n+1] = x1

        s = x1 - x
        y = grad(x1) - g
        rho = 1 / np.dot(y, s)
        I = np.eye(len(x))

        u1 = I - rho * np.outer(y, s)
        u2 = I - rho * np.outer(s, y)

        U = np.matmul(B,u2)

        B = np.matmul(u1, U) + rho * np.outer(y, y)

        x = x1

        n +=1

    print("Max Iterations Reached")
    return x, x_seq, n
```

SR1 Implementation:

```
def SR1_update(B, s, y):

    denominator = np.dot(y - np.dot(B, s), s)

    if abs(denominator) < 1e-8:
        return B

    numerator = np.outer(y - np.dot(B, s), y - np.dot(B, s))

    B_new = B + numerator / denominator

    if not is_pos_def(B_new):
        return B
    return B_new

def SR1_TR(f, g, H, x0, maxiter=100000, tol=1e-12):
k=0
Bk = np.eye(3)
trust_radi = 0.32
eta = 0.5*10**-3
r=.4
x_seq = np.zeros((maxiter+1, len(x0)))
while k < maxiter:

    gk = g(x0)

    if np.linalg.norm(gk) < tol:
        break

    if np.dot(np.dot(gk,Bk), gk) <=0:
        sk = -((trust_radi/np.linalg.norm(gk))*gk)
    elif ((np.linalg.norm(gk)**3)/(trust_radi*np.dot(np.dot(gk,Bk), gk))) < 1 :
         sk = -((trust_radi/np.linalg.norm(gk))*gk)*(np.linalg.norm(gk)**3)/(trust_radi*n
    else:
        sk = -((trust_radi/np.linalg.norm(gk))*gk)
    yk=g(x0+sk)-gk
    ared = f(x0)-f(x0+sk)
    pred = -(np.dot(g(x0).T, sk)+0.5*np.dot(np.dot(sk,Bk),sk))

    if abs(ared/pred) > eta:

        x=x0+sk
    else:

        x = x0
    if abs(ared/pred) > 0.75:
        if np.linalg.norm(sk) > 0.8 * trust_radi:
            trust_radi = 2*trust_radi
    elif 0.1 <= abs(ared/pred) and abs(ared/pred) <=0.75:
        trust_radi = trust_radi
    else:
        trust_radi = 0.5*trust_radi

    if abs(np.dot(sk,(yk-np.dot(Bk,sk)))) >= r*np.linalg.norm(sk)*np.linalg.norm(yk-np.dot
        Bk = SR1_update(Bk, sk, yk)

    k += 1
    x0 = x
    x_seq[k] = x

return x0, x_seq, k
```