Quasi-Newton Method for Optimization of Nonlinear Unconstrained Problems

Ian Pylkkanen
MANE 6710

December 16, 2020

1 Executive Summary

This report details the quasi-Newton method of optimization and seeks to verify the method through a series of simple and complex unconstrained mathematical problems. The commonly used BFGS method is used in conjunction with a line-search to successfully perform minimizations on 4 different mathematical models. The process taken and the subsequent results are discussed in the sections that follow.

2 Introduction

The quasi-Newton method was first proposed in the mid 1950s by W.C. Davidon, who looked to reduce the computational cost of performing complex optimization problems. To do so, he created a method that would only require the objective function and its gradient to be supplied at each iteration. This allowed for a faster model capable of producing superlinear convergence without the need to compute the Hessian. This was a significant improvement over the slower more conventional steepest descent method [1].
The following subsections provide an overview of the BFGS and line-search methods used to perform the quasi-Newton optimization problems. The algorithms and underlying equations are presented and discussed.

2.1 The BFGS Method

Named after its discoverers, Broyden, Fletcher, Goldfarb, and Shanno, the BFGS method is the most popular quasi-Newton algorithm. It begins, as any optimization problem does, with the objective function. In this case it follows a quadratic model at the current iterate x_k as shown in Equation 1 [1].

$$m_k(p) = f_k + \nabla f_k^T p + \frac{1}{2} p^T B_k p \tag{1}$$

The important terms in the objective function relative to the BFGS method are B_k and p_k. B_k is an approximation of the true Hessian and is to be updated at every iteration. p_k is the search direction and is used to define the value of x at the next iterate [1].

$$x_{k+1} = x_k + \alpha_k p_k \tag{2}$$

In order to reduce the computation time, Davidon proposed that rather than recalculating B_k at each iteration, the information gained from the change in x_k to x_{k+1} should be used to define B_{k+1}. To do so the secant equation is used as shown in Equation 3 [1].

$$B_{k+1} s_k = y_k \tag{3}$$

Where s_k is the displacement and y_k is the change in gradient. The secant equation requires that s_k and y_k satisfy the curvature condition, such that the next iteration should result in a gradient that is less negative than before, or even positive. This concept is rather straight forward for a convex function, but for a nonconvex function a restriction needs to be put in place. This restriction comes in the form of the step length α_k, which is determined using the line-search method discussed in Section 2.2. The step length determines how far away the value of x_{k+1} is. The curvature condition can then be represented as [1]:

$$s_k^T y_k \geq (c_2 - 1)\alpha_k \nabla f_k^T p_k \tag{4}$$

$c_2 < 1$

Broyden, Fletcher, Goldfarb, and Shanno, however, found that rather than solving for the approximation of the Hessian, B_{K+1}, it was better to instead solve for its inverse, H_{k+1} [1].

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \tag{5}$$

$\rho_k = \frac{1}{y_k^T s_k}$

Using this equation for the unique solution of H_{k+1}, the BFGS algorithm was derived [1]. Where H_0 is defined as the initial approximation of the inverse Hessian. For this case H_0 is defined as the identity matrix. The BFGS algorithm is presented on the following page.

Algorithm 1: BFGS Method

Result: Minimum Value of x
Given x_0, covergence tolerance $\epsilon > 0$, define H_0;
$k \leftarrow 0$;
while $\nabla f_k > \epsilon$ **do**
    Compute search direction: $p_k = -H_k \nabla f_k$;
    Set $x_{k+1} = x_k + \alpha_k p_k$ *where* α_k *is computed from line-search (Section 2.2);*
    Define $s_k = x_{k+1} - x_k$ *and* $y_k = \nabla f_{k+1} - \nabla f_k$;
    Compute H_{k+1} *from Equation 5;*
    $k = k + 1$;
end

Overall the BFGS method is a fantastic method to use for complex optimization problems. The algorithm is robust and has a fast superlinear convergence. Each iteration can be performed at a cost of $O(n^2)$, which is significantly lower than a Newton's method approach [1].

2.2 Line-Search

The line-search is used to determine the step length, α_k, used in the BFGS algorithm. On a high level the line-search looks to determine an appropriate step length for x_{k+1} to satisfy both the sufficient decrease and curvature conditions. The curvature condition remains the same as previously mentioned while the sufficient decrease is a requirement that the function evaluation at x_{k+1} must be significantly smaller than that at x_k [1].

$$f(x_{k+1}) \leq f(x_k) + c_1 \alpha_k p_k^T \nabla f(x_k) \tag{6}$$

$0 < c_1 < c_2 < 1$

For the line-search Equation 6 can be rewritten in terms of the step length [1].

$$\phi(\alpha_k) \leq \phi(0) + c_1 \alpha_k \phi'(0) \tag{7}$$

Together, the sufficient decrease and curvature conditions are known as the Wolfe conditions.
The line-search searches along the search direction, p_k, to find a value for α_k that satisfies the Wolfe conditions. The line-search begins with an initial estimate α_0 and a generate sequence α_i. The algorithm will either terminate with a satisfactory step length, or determine that one does not exist. To do so a "bracketing phase", where the line-search selects a step length between the lower and upper bounds of $[\alpha_{lo}, \alpha_{hi}]$, is used along with a "selection phase" that zooms in to locate the final step length α_*. The selection phase will often reduce the size of the interval between the lower and upper bounds in order to narrow its search [1].

The algorithm for the line-search performs these steps in three "if statements". The first statement says that if the current step length violates the sufficient decrease condition, it will be sent to the *Zoom* function to find an appropriate value α_*. The second statement says that if the current step length passes the curvature condition, that value becomes α_*. Finally, the third statement says that if the current step length violates the curvature condition, it will be sent to the *Zoom* function to find an appropriate value α_*. The algorithm for the line-search is presented below [1].

Algorithm 2: Line-Search

Result: Step Length α_k
Set $\alpha_0 \leftarrow 0$, choose $\alpha_{max} > 0$ and $\alpha_1 \in (0, \alpha_{max})$;
$i \leftarrow 1$;
while do
  Evaluate $\phi(\alpha_i)$;
  if $\phi(\alpha_i) > \phi(0) + c_1\alpha_i\phi'(0)$ *or* $[\phi(\alpha_i) \geq \phi(\alpha_{i-1})$ *and* $i > 1$ **then**
    $\alpha_* \leftarrow$ **zoom**(α_{i-1}, α_i)
  end
  if $|\phi'(\alpha_i)| \leq -c_2\phi(0)$ **then**
    set $\alpha_* \leftarrow \alpha_i$
  end
  if $\phi'(\alpha_i) \geq 0$ **then**
    set $\alpha_* \leftarrow$ **zoom**(α_i, α_{i-1})
  end
  Choose $\alpha_{i+1} \in (\alpha_i, \alpha_{max})$;
  $i \leftarrow i + 1$;
end

When α is sent to *Zoom*, the function is tasked with finding an appropriate step length. As previously mentioned, the function operates by searching for an α_* within the bounds $[\alpha_{lo}, \alpha_{hi}]$. It begins with a trial step lenth, α_j, that is calculated using a form of interpolation. For this paper bisection interpolation is used as presented in Equation 8.

$$\alpha_j = \frac{\alpha_{hi} + \alpha_{lo}}{2} \tag{8}$$

Once the step length satisfies the Wolfe conditions the operation terminates, and α_* is sent to the BFGS algorithm as the step length α_k. The zoom algorithm is presented on the following page [1].

Algorithm 3: Zoom

Result: Step Length α_*
while do
  Interpolate (using quadratic, cubic, or bisection) to find a trial step length α_j between $[\alpha_{lo}, \alpha_{hi}]$;
  Evaluate $\phi(\alpha_j)$;
  if $\phi(\alpha_j) > \phi(0) + c_1\alpha_j\phi'(0)$ *or* $\phi(\alpha_j \geq \phi(\alpha_{lo})$ **then**
    $\alpha_{hi} \leftarrow \alpha_j$;
  else
    Evaluate $\phi(\alpha_j)$;
    if $|\phi'(\alpha_j)| \leq -c_2\phi'(0)$ **then**
      Set $\alpha_* \leftarrow \alpha_j$;
    end
    if $\phi'(\alpha_j)(\alpha_{hi} - \alpha_{lo}) \geq 0$ **then**
      $\alpha_{hi} \leftarrow \alpha_{lo}$;
    end
    $\alpha_{lo} \leftarrow \alpha_j$;
  end
end

3 Optimization Method

This section will discuss the procedure to setup and run the optimization of four distinct mathematical problems. These mathematical models are then used to verify the validity of the quasi-Newton method.
With the algorithms for the BFGS, line-search, and zoom function already defined, the process of setting up the optimization is actually quite simple. All that is required is the definition of the objective function and its gradient. For this report, four mathematical models are analyzed, and thus each is used as a separate objective function. First, two simple one-dimensional models are used to verify the line-search. These models are:

1. $f(x) = \cos(x)$
2. $f(x) = x^2$

Then, two complex mathematical models are defined to verify that the BFGS and line-search work together to successfully perform the optimization. These functions are the two-dimensional Booth Function [2], and the four-dimensional Colville Function [3] defined as follows.

3. Booth Function: $f(x_1, x_2) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$

4. Colville Function: $f(x_1, x_2, x_3, x_4) = 100(x_1^2 - x_2)^2 + (x_1 - 2)^2 + (x_3 - 1)^2 + 90(x_3^2 - x_4)^2 + 10.1((x_2 - 1)^2 + (x_4 - 1)^2) + 19.8(x_2 - 1)(x_4 - 1)$

Once each function is defined as its own objective function, their gradients must be calculated. To do so, complex step differentiation is used. Complex step is a means to estimate a derivative by passing a complex number into the function. The general equation is shown in Equation 9.

$$f'(x) = \frac{Im(f(x + ih))}{h} \tag{9}$$

Where h is the step-size and i is the imaginary unit.

Complex step is both an inexpensive and effective way to estimate the derivatives of complex mathematical problems such as the Booth or Colville functions. Once implemented, it will provide the gradient for each objective function, which can then be used to begin the optimization in the BFGS algorithm.

4 Results

This section will present and discuss the findings of the optimizations of the four mathematical models previously described. First, the verification of the line-search will be conducted using the simple 1D models. Next, the overall optimization will be verified using the Booth and Colville functions. All the MATLAB code used to perform this analysis can be found in Appendix A.

4.1 Line-Search Verification

In order to verify that the line-search algorithm is properly working, the 1D problems presented in Section 3 will be used. It will be shown that through multiple iterations the line-search eventually satisfies the Wolfe conditions. The value of x_k and $f(x_k)$ will be stored and plotted at each intermediate step length that runs through the line-search. The goal is to illustrate that while some of the points do not satisfy the conditions, over multiple iterations the line-search terminates with one that does. Figure 1 and Figure 2 show the results of this verification.

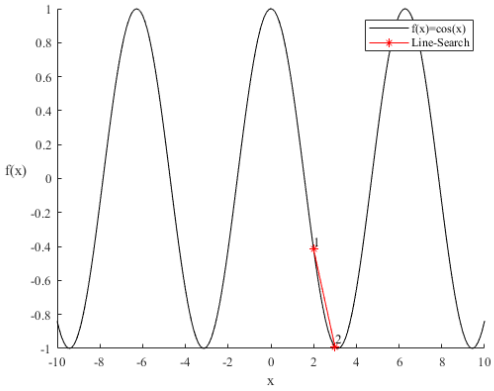

Figure 1: Cosine Function

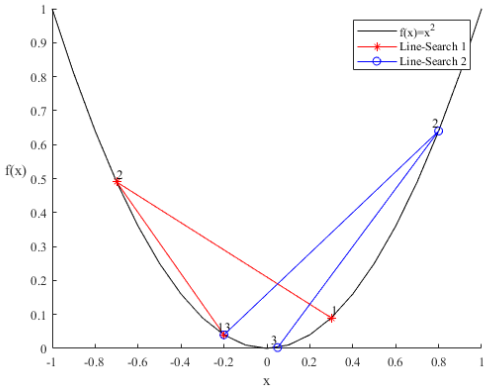

Figure 2: Quadratic Function

Figure 1 shows the first iteration of the line-search for the function $f(x) = \cos(x)$. The first point is the initial guess provided by the user. The second point is the result of the first iteration of the line-search along the search direction. The line-search algorithm finds a point that satisfies the Wolfe conditions. In this case the algorithm would terminate and pass the value of α_k into BFGS.
Figure 2 shows two iterations of the line-search for the function $f(x) = x^2$. It also displays two line-searches performed. The first line-search, shown in red, terminates after two iterations. Here point 1 (on the right side) is the initial guess and point 2 is the first step length in the search direction. This point however does not satisfy the sufficient decrease condition and so is passed into the *Zoom* function. *Zoom* then outputs the step length that results in point 3. This point does satisfy the Wolfe conditions and is passed into BFGS. However, BFGS determines it is not a local minimizer and tells the line-search to run again. This process is then repeated along the blue line, with the initial point being the previous point 3 (point 1 on the left-side) and terminating at point 3 (on the right side). This process would then be repeated until BFGS determines a local minimizer has been found.
As evident by both functions, the line-search is working perfectly. The next step is to verify that the BFGS algorithm is working in conjunction with the line-search to opeate the entire optimization problem.

4.2 BFGS Verification

The easiest way to verify that the optimization is working as a whole is to verify that the minimization is correct. The BFGS algorithm will output both the function evaluation, $f(x_k)$, and the value of x_k at the local minimum. To do so the Booth and Colville functions described is Section 3 are used. The output of BFGS will be compared to the values provided in literature for the local minimum of the Booth [2] and Colville [3] functions.
Let's begin with the 2D Booth Function. The local minimum of the Booth Function is:

$$f(x_1, x_2) = 0 \text{ at } (x_1, x_2) = (1, 3)$$

For the BFGS and line-search algorithms it is necessary to provide an initial guess of x_1 and x_2 as well as a value for the convergence tolerance, ϵ. The initial guess is arbitrarily set to $(2, 10)$ and the tolerance is set to $\epsilon = .01$. For this case, the result is:

Input: $(x_1, x_2) = (2, 10)$
$\epsilon = .01$
Output: $f(x_1, x_2) = 1.5414 * 10^{-6}$
$(x_1, x_2) = (1.001, 3.005)$

The results are very good, but can easily be made better by reducing the tolerance.

Input: $(x_1, x_2) = (2, 10)$
$\epsilon = 1 * 10^{-6}$
Output: $f(x_1, x_2) = 3.1377 * 10^{-17}$
$(x_1, x_2) = (1.000, 3.000)$

One more time.

Input: $(x_1, x_2) = (2, 10)$
$\epsilon = 1 * 10^{-30}$
Output: $f(x_1, x_2) = 0$
$(x_1, x_2) = (1.000, 3.000)$

For all three tolerances, the computation time was mere milliseconds. This process can then be repeated on the Colville Function. As provided from literature, the Colville Function has a local minimum at:

$$f(x_1, x_2, x_3, x_4) = 0 \text{ at } (x_1, x_2, x_3, x_4) = (1, 1, 1, 1)$$

Run at an arbitrary initial guess of $(x_1, x_2, x_3, x_4) = (3, 5, 2, 6)$ and an appropriately small tolerance of $\epsilon = 1 * 10^{-10}$, the output yields:

Input: $(x_1, x_2, x_3, x_4) = (3, 5, 2, 6)$
$\epsilon = 1 * 10^{-10}$
Output: $f(x_1, x_2, x_{3x4}) = 8.6012 * 10^{-27}$
$(x_1, x_2, x_3, x_4) = (1.000, 1.000, 1.000, 1.000)$

Unlike the Booth Function, the Colville Function cannot be run at a tolerance much lower than $\epsilon = 1 * 10^{-10}$ without getting unreasonably computationally expensive. At this tolerance however, the function still converges nicely to the local minimizer in a few milliseconds.

5 Conclusion

The quasi-Newton method using BFGS and line-search algorithms was successful in performing numerical optimization on both simple and complex mathematical models. By using such methods, computation time was kept minimal without the sacrifice of accuracy. For the reasons presented in this paper, quasi-Newton methods are extremely popular in optimizing highly complex unconstrained problems.

6 References

[1] J. Nocedal and S. J. Wrigth, Numerical Optimization, Second ed., New York, NY: Springer Science+Business Media, LLC, 2006.
[2] D. Bingham, "Booth Function," Simon Fraser University, 2013. [Online]. Available: https://www.sfu.ca/ ssurjano/booth.html. [Accessed 15 December 2020].
[3] D. Bingham, "Colville Function," Simon Fraser University, 2013. [Online]. Available: https://www.sfu.ca/ ssurjano/colville.html. [Accessed 15 December 2020].

7 Appendix A

Listing 1: Objective Function

```
% Objective Function
%
% Inputs:
% Initial Guess: xk
%
% Outputs:
% Function Evaluation: f
% Gradient: delf
%
% Ian Pylkkanen
% December 16, 2020

function [f, delf] = obj(xk)
% Step-size for Complex Step
h = 1e-60;

% Solve function at initial guess xk
f = subobj(xk);

% Define gradient matrix
delf = zeros(size(xk,1),1);

% Implement Complex Step to find gradient
for i=1:size(xk)
    xc = xk;
    xc(i) = xk(i) + complex(0,h);
    delf(i) = imag(subobj(xc))/h;
end

    % Subfunction to solve for f
    function [f] = subobj(dv)
        % Define variables
        x1 = dv(1);
        x2 = dv(2); % Uncomment of 2D problem
        x3 = dv(3); % Uncomment for 3D problem
        x4 = dv(4); % Uncomment of 4D problem

        % Define function f and solve
        %f = x1.^2;
        %f = cos(x1);
        %f = (x1+2*x2-7)^2 + (2*x1+x2-5)^2;
        f = 100*(x1^2-x2)^2 + (x1-1)^2 + (x3-1)^2 + 90*(x3^2-x4)^2 + 10.1*((x2-1)^2 + (x4-1)^2) +
            19.8*(x2-1)*(x4-1);
    end
end
```

Listing 2: Line-Search

```
% Line-Search Algorithm
%
% Adopted from: Nocedal, Wright "Numercial Optimization", 2006
% Algorithm 3.5
%
% Inputs:
% Initial Guess: xk
% Initial Step Length: alpha1
% Search Direction: pk
%
% Outputs:
% Step Length Satisfying Wolfe Conditions: alphastar
%
% Ian Pylkkanen
% December 16, 2020

function [alphastar] = linesearch(xk, alpha1, pk)
% Define Initial Parameters
alpha0 = 0;
alphamax = 5*alpha1;
xk0 = xk + alpha0*pk;
c1 = 10^-4;
c2 = .9;
i = 1;
% Call Objective Function to be minimized
[phi0, phip0] = obj(xk0);
% Define Objective at Initial Guess
phip0 = phip0'*pk;
[phi_0, phip_0] = obj(xk);
phip_0 = phip_0'*pk;

while 1
    xk1 = xk + alpha1*pk;
    [phi1, phip1] = obj(xk1);
    phip1 = phip1'*pk;
    % If alpha fails sufficient decrease, send to Zoom
    if phi1 > phi_0 + c1*alpha1*phip_0 || (phi1 >= phi0 && i>1)
        alphastar = zoom(alpha0, alpha1);
        break
    end
    % If alpha passes sufficient decrease, set alphastar=alpha1
    if abs(phip1) <= -c2*phip_0
        alphastar = alpha1;
        break
    end
    % If alpha fails curvature condition, send to Zoom
    if phip1 >= 0
        alphastar = zoom(alpha1, alpha0);
        break
    end
    alpha0 = alpha1;
    alpha1 = (alpha1+alphamax)/2;
    i = i+1;
end

% Zoom Function

% Adopted from: Nocedal, Wright "Numercial Optimization", 2006
% Algorithm 3.6

% Inputs:
% Lower and Upper Bounds: [alphalo, alphahi]

% Outputs:
% Step Length Satisfying Wolfe Conditions: alphastar

% Zoom Function looks to find alphastar to satisfy Wolfe Conditions
function [alphastar] = zoom(alphalo, alphahi)
while 1
    alphaj = (alphahi+alphalo)/2; % Define Trial Step Length
    xj = xk + pk*alphaj;
    [phij, phipj] = obj(xj);
    phipj = phipj'*pk;
    [philo, phiplo] = obj(xk + pk*alphalo);
    phiplo = phiplo'*pk;
    if phij > phi0 + c1*alphaj*phip0 || phij >= philo
        alphahi = alphaj;
    else
        if abs(phipj) <= -c2*phip0
            alphastar = alphaj;
            return
        end
        if phipj*(alphahi-alphalo) >= 0
            alphahi = alphalo;
        end
        alphalo = alphaj;
    end
end
```

```
end
end
```

Listing 3: BFGS

```
% BFGS Algorithm
%
% Adopted from: Nocedal, Wright "Numercial Optimization", 2006
% Algorithm 6.1
%
% Inputs:
% nital guess: xk
% Tolerance: eps
% Objective Function: obj
%
% Outputs:
% Local Minimizer: f(xk), xk
%
% Ian Pylkkanen
% December 16, 2020

function bfgs(xk,eps)

% Define Initial Guess of Inverse Hessian
Hk = eye(length(xk));

% Call Objective Function ant Initial Guess
[fk, delfk] = obj(xk);

while norm(delfk) > eps
    pk = -Hk*delfk; % Define Search Direction
    pk = pk/norm(pk);
    alpha1 = 1; % Initial Guess for Step-Length
    alphak = linesearch(xk, alpha1, pk); % Define Step Length from Line-Search
    xk1 = xk + alphak*pk; % Define new x-value
    sk = xk1 - xk; % Displacement
    [fk1, delfk1] = obj(xk1); % Solve objective at new xk1
    yk = delfk1 - delfk; % Change in Gradient
    rhok = 1./(yk'*sk); %
    I = eye(length(xk));
    Hk = (I - rhok*sk*yk')*Hk*(I - rhok*yk*sk') + rhok*sk*sk'; % Solve for Inverse Hessian
    xk = xk1;
    [fk, delfk] = obj(xk);
end
% Define Final Values
delfk = norm(delfk); % Gradient
xk; % x-values
fk = norm(fk); %Function Evaluation
end
```

NOTE: Extra functions were used in order to store and plot data for the line-search verification. This code was removed in order to provided a neater display.