Final Project Report

Shenbiao Dai

May 10, 2019

1 Introduction

In this final project, we will use one of Quasi-Newton method called **BFGS Method** to implement on a nonlinear least square problem, which is based on a dataset in document `Gauss3.dat`. The code to insert dataset is stored in `DataPrepare.m`. In order to polish the algorithm, we introduce **Limited Memory BFGS(L-BFGS) Method** to reduce the use of memory. To compare the difference between the two methods, we list the outcomes in each steps and time-comsumption to reach the convergence.

1.1 Problem Statement

We introduce the whole problem below.

- Implement BFGS
  Test implement on non-linear LS problem

$$\min_{\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{f}(\mathbf{x}, \boldsymbol{\beta})\|_2^2 \tag{1}$$

  where

$$\mathbf{y} = (y_1, y_2, \ldots, y_m)^t \tag{2}$$

$$\mathbf{f}(\mathbf{x}, \boldsymbol{\beta}) = (f(x_1, \boldsymbol{\beta}), f(x_2, \boldsymbol{\beta}), \ldots, f(x_m, \boldsymbol{\beta}))^t \tag{3}$$

  with

$$f(x, \boldsymbol{\beta}) = \beta_1 \cdot \mathbf{e}^{-\beta_2 x} + \beta_3 \cdot \mathbf{e}^{-(x-\beta_4)^2/\beta_5^2} + \beta_6 \cdot \mathbf{e}^{-(x-\beta_7)^2/\beta_8^2} \tag{4}$$

  Detailed data (x_i, y_i) for $1 \le i \le m$ can be found at file `Gauss3.dat`.
  Initialize your $\boldsymbol{\beta}$ estimate with the starting values provided there, and initialize BFGS matrix with the true Hessian matrix at the initial point. Run your code for up to 20 steps for a tolerance of 10^{-3}, and report error if your code fails before reaching 20 steps.

- Repeat the same work with L-BFGS for $m = 3$.

1.2 Iteration Method

1.2.1 The Iteration in BFGS Method

First, denote that

$$g(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{f}(\mathbf{x}, \boldsymbol{\beta})\|_2^2 \tag{5}$$

For the optimization problem $\min_{\boldsymbol{\beta}} g(\boldsymbol{\beta})$, we formulate the original Newton Method

$$\boldsymbol{\beta}^{k+1} = \boldsymbol{\beta}^k - \alpha^k \left(\nabla^2 g(\boldsymbol{\beta}^k)\right)^{-1} \nabla g(\boldsymbol{\beta}^k) \tag{6}$$

Although in Newton Method, we can obtain superlinear convergence, which means if $\boldsymbol{\beta}^*$ is the optimal solution,

$$\lim_{k\to\infty}\frac{\|\boldsymbol{\beta}^{(k+1)}-\boldsymbol{\beta}^*\|}{\|\boldsymbol{\beta}^{(k)}-\boldsymbol{\beta}^*\|}=0 \tag{7}$$

However, it is expensive to compute the Jacobi matrix. Instead, we are inclined to adopt Quasi-Newton Method, which means that we need to find a typically symmetric PSD matrix $\mathcal{B}$ such that

$$\boldsymbol{\beta}^{k+1}=\boldsymbol{\beta}^k-\alpha^k\mathcal{B}_k^{-1}\nabla g(x^k) \tag{8}$$

in order to find the balance between keeping the quick convergence speed and reducing the computation. In this project, we design that

$$\mathcal{B}_{k+1}=\mathcal{B}_k+\frac{\mathbf{y}_{k+1}\mathbf{y}_{k+1}^t}{\mathbf{y}_{k+1}^t\mathbf{s}_{k+1}}-\frac{\mathcal{B}_k\mathbf{s}_{k+1}\mathbf{s}_{k+1}^t\mathcal{B}_k}{\mathbf{s}_{k+1}^t\mathcal{B}_k\mathbf{s}_{k+1}} \tag{9}$$

where

$$\mathbf{y}_{k+1}=\nabla g\left(\boldsymbol{\beta}^{(k+1)}\right)-\nabla g\left(\boldsymbol{\beta}^{(k)}\right) \tag{10}$$

$$\mathbf{s}_{k+1}=\boldsymbol{\beta}^{(k+1)}-\boldsymbol{\beta}^{(k)} \tag{11}$$

Assume that

$$\mathcal{B}_0=\nabla^2 g(\boldsymbol{\beta}^{(0)}) \tag{12}$$

where the only burdensome computation hides and $\boldsymbol{\beta}_0$ is given in the file `Gauss3.dat`. When we need to figure out the inverse matrix of $\mathcal{B}$ in MATLAB, we can use $I\backslash\mathcal{B}$ instead of function `inv` to reduce the amount of computation, because '\' uses LU decomposation to solve linear system of equations.

1.2.2 The Revision in L-BFGS Method

When implementing the algorithm in BFGS method, the problem is that $\mathcal{B}_k^{-1}$ may be too large to fit in the memory. However, we could avoid such problem if setting a number m previously and going back only $m+1$ steps to update the matrix $\mathcal{B}_k^{-1}$. In this case, we mimics BFGS method with linear memory instead of quadratic memory.
Let's return to the original $\mathcal{B}_{k+1}^{-1}$-update procedure in Eq.9. If we recursively implement Eq.9, the storage of each $\mathcal{B}_{k+1}^{-1}$ is unnecessary, which can be replaced by the storage of vectors $\mathbf{y}_{k+1}$ and $\mathbf{s}_{k+1}$. At first, we inverse both sides of Eq.9 and obtain that

$$\mathcal{B}_{k+1}^{-1}=\left(I-\frac{\mathbf{s}_{k+1}\mathbf{y}_{k+1}^t}{\mathbf{s}_{k+1}^t\mathbf{y}_{k+1}}\right)\mathcal{B}_k^{-1}\left(I-\frac{\mathbf{s}_{k+1}\mathbf{y}_{k+1}^t}{\mathbf{s}_{k+1}^t\mathbf{y}_{k+1}}\right)^t+\frac{\mathbf{s}_{k+1}\mathbf{s}_{k+1}^t}{\mathbf{s}_{k+1}^t\mathbf{y}_{k+1}} \tag{13}$$

which can be recursively implemented. If we extend Eq.13 within m steps, we have

$$\begin{aligned}\mathcal{B}_{k+1}^{-1}=&\frac{\mathbf{s}_{k+1}\mathbf{s}_{k+1}^t}{\mathbf{s}_{k+1}^t\mathbf{y}_{k+1}}+\prod_{j=1}^{m+1}\left(I-\frac{\mathbf{s}_{k-m+j}\mathbf{y}_{k-m+j}^t}{\mathbf{s}_{k-m+j}^t\mathbf{y}_{k-m+j}}\right)\mathcal{B}_{k-m}^{-1}\prod_{j=1}^{m+1}\left(I-\frac{\mathbf{y}_{k-m+j}\mathbf{s}_{k-m+j}^T}{\mathbf{s}_{k-m+j}^T\mathbf{y}_{k-m+j}}\right)\\&+\sum_{j=0}^{m-1}\prod_{i=0}^{j}\left(I-\frac{\mathbf{s}_{k+1-j+i}\mathbf{y}_{k+1-j+i}^t}{\mathbf{s}_{k+1-j+i}^t\mathbf{y}_{k+1-j+i}}\right)\left(\frac{\mathbf{s}_{k-j}\mathbf{s}_{k-j}^T}{\mathbf{s}_{k-j}^T\mathbf{y}_{k-j}}\right)\prod_{i=0}^{j}\left(I-\frac{\mathbf{s}_{k+1-j+i}\mathbf{y}_{k+1-j+i}^t}{\mathbf{s}_{k+1-j+i}^t\mathbf{y}_{k+1-j+i}}\right)\end{aligned} \tag{14}$$

To simplify the method, set $\mathcal{B}_{k-m}^{-1}=\mathcal{B}_0^{-1}$. Notice that in Eq.8, $\mathcal{B}_k^{-1}\nabla g\left(x^k\right)$ always appear together. In order to reduce quadratic memory into linear memory, we condider $\mathcal{B}_{k+1}^{-1}\nabla g\left(x^{k+1}\right)$. Denote that

$$\alpha_p=\begin{cases}\dfrac{\mathbf{s}_{k+1}^t\mathbf{h}}{\mathbf{s}_{k+1}^t\mathbf{y}_{k+1}}, & p=k+1\\ \dfrac{\mathbf{s}_t^t\mathbf{q}_{t+1}}{\mathbf{s}_t^t\mathbf{y}_t}, & p=k,\ldots,k+1-m\end{cases} \tag{15}$$

$$\mathbf{q}_p = \begin{cases} \mathbf{h} - \alpha_{k+1}\mathbf{y}_{k+1}, & p = k+1 \\ \prod_{j=t}^{k+1}(I - \dfrac{\mathbf{y}_j\mathbf{s}_j^t}{\mathbf{s}_j^t\mathbf{y}_j})\mathbf{h} = \mathbf{q}_{p+1} - \alpha_p\mathbf{y}_p, & p = k, \ldots, k+1-m \end{cases} \tag{16}$$

Left multiply $\mathbf{h}$ on both sides of Eq.14, and according to Eq.15, Eq.16, we obtain that

$$\begin{aligned} \mathcal{B}_{k+1}^{-1}\mathbf{h} =& \alpha_{k+1}\mathbf{s}_{k+1} + \prod_{j=1}^{m+1}\left(I - \frac{\mathbf{s}_{k-m+j}\mathbf{y}_{k-m+j}^t}{\mathbf{s}_{k-m+j}^t\mathbf{y}_{k-m+j}}\right)\mathcal{B}_0^{-1}\mathbf{q}_{k+1-m} \\ &+ \sum_{j=0}^{m-1}\prod_{i=0}^{j}\left(I - \frac{\mathbf{s}_{k+1-j+i}\mathbf{y}_{k+1-j+i}^t}{\mathbf{s}_{k+1-j+i}^t\mathbf{y}_{k+1-j+i}}\right)\left(\frac{\mathbf{s}_{k-j}\mathbf{s}_{k-j}^T}{\mathbf{s}_{k-j}^T\mathbf{y}_{k-j}}\right)\mathbf{q}_{k+1-j} \end{aligned} \tag{17}$$

Let

$$\mathbf{z}_{k+1-m} = \mathcal{B}_0^{-1}\mathbf{q}_{k+1-m} \tag{18}$$

We translate Eq.17 into

$$\begin{aligned} \mathcal{B}_{k+1}^{-1}\mathbf{h} &= \alpha_{k+1}\mathbf{s}_{k+1} + \prod_{j=1}^{m+1}\left(I - \frac{\mathbf{s}_{k-m+j}\mathbf{y}_{k-m+j}^t}{\mathbf{s}_{k-m+j}^t\mathbf{y}_{k-m+j}}\right)\mathbf{z}_{k+1-m} \\ &\quad + \sum_{j=0}^{m-1}\prod_{i=0}^{j}\left(I - \frac{\mathbf{s}_{k+1-j+i}\mathbf{y}_{k+1-j+i}^t}{\mathbf{s}_{k+1-j+i}^t\mathbf{y}_{k+1-j+i}}\right)\alpha_{k-j}\mathbf{s}_{k-j} \\ &= \alpha_{k+1}\mathbf{s}_{k+1} + \prod_{j=2}^{m+1}\left(I - \frac{\mathbf{s}_{k-m+j}\mathbf{y}_{k-m+j}^t}{\mathbf{s}_{k-m+j}^t\mathbf{y}_{k-m+j}}\right)\mathbf{z}_{k+2-m} \\ &\quad + \sum_{j=0}^{m-2}\prod_{i=0}^{j}\left(I - \frac{\mathbf{s}_{k+1-j+i}\mathbf{y}_{k+1-j+i}^t}{\mathbf{s}_{k+1-j+i}^t\mathbf{y}_{k+1-j+i}}\right)\alpha_{k-j}\mathbf{s}_{k-j} \end{aligned} \tag{19}$$

where

$$\mathbf{z}_{k+2-m} = \mathbf{z}_{k+1-m} + (\alpha_{k+1-m} - \beta_{k+1-m})\,\mathbf{s}_{k+1-m} \tag{20}$$

$$\beta_{k+1-m} = \frac{\mathbf{y}_{k+1-m}^T\mathbf{z}_{k+1-m}}{\mathbf{s}_{k+1-m}^T\mathbf{y}_{k+1-m}} \tag{21}$$

If we let $\mathbf{h} = \nabla g\left(x^{k+1}\right)$, the technique of recursive iteration can be used in the update of each $\mathbf{x}^{k+1}$.

2 Algorithm

2.1 The Algorithm of BFGS Method

Algorithms can be included using the commands as shown in algorithm 1. The code is stored in `BFGS.m`.

Algorithm 1 BFGS Method's Algorithm

1: Initialize β_0
2: $k \leftarrow 0$
3: $N \leftarrow 50$ ▷ Set the maximum of iterations
4: $TOL \leftarrow 1\mathrm{e}-3$
5: $B_0 \leftarrow Jg(\beta_0)$ ▷ J denotes Jacobi matrix, code stored in `Hessian.m`
6: **while** $k < N$ **do**
7: $\quad k \leftarrow k+1$
8: $\quad d \leftarrow -B_0 \backslash I * \nabla g(\beta_0)$ ▷ Gradient code stored in `nabla_g.m`

Choose α ▷ Code stored in `stepsize.m` and `Armijo_stepsize.m`
$\beta \leftarrow \beta_0 + \alpha * d$
if $|g(\beta_0) - g(\beta)| < TOL$ **then**
break
end if
$s \leftarrow \beta - \beta_0$
$y \leftarrow \nabla g(\beta) - \nabla g(\beta_0)$
$B_0 \leftarrow B_0 + \frac{y * y'}{y' * s} - \frac{B_0 * s * s' * B_0}{s' * B_0 * s}$
$\beta_0 \leftarrow \beta$ ▷ Update β each step
end while
return β ▷ The coefficient of original function

2.2 The Algorithm of BFGS Method

Algorithms can be included using the commands as shown in algorithm 2. The code is stored in `LBFGS.m`.

Algorithm 2 L-BFGS Method's Algorithm

Initialize β_0
$k \leftarrow 0$
$N \leftarrow 50$ ▷ Set the maximum of iterations
$TOL \leftarrow 1\mathrm{e}-3$
$m \leftarrow 3$
$m_0 \leftarrow 8$ ▷ Set the number of unknown coefficient
$B_0 \leftarrow Jg(\beta_0)$ ▷ J denotes Jacobi matrix, code stored in `Hessian.m`
$q \leftarrow \nabla g(\beta_0)$ ▷ Code stored in `nabla_g.m`
$y \leftarrow \mathrm{zeros}(m0, m+1)$
$s \leftarrow \mathrm{zeros}(m0, m+1)$
$a \leftarrow \mathrm{zeros}(m+1, 1)$ ▷ (Step 8-10)Previously leave space for memory
while $k < m$ **do** ▷ Use **Algorithm 1** in the first m steps for further recursion
$k \leftarrow k + 1$
$d \leftarrow -B_0 \backslash I * \nabla g(\beta_0)$
Choose αCode stored in `stepsize.m` and `Armijo_stepsize.m`
$\beta \leftarrow \beta_0 + \alpha * d$
if $|g(\beta_0) - g(\beta)| < TOL$ **then**
break
end if
$s(:, k+1) \leftarrow \beta - \beta_0$
$y(:, k+1) \leftarrow \nabla g(\beta) - \nabla g(\beta_0)$
$B_0 \leftarrow B_0 + \frac{y(:,k+1) * y(:,k+1)'}{y(:,k+1)' * s(:,k+1)} - \frac{B_0 * s(:,k+1) * s(:,k+1)' * B_0}{s(:,k+1)' * B_0 * s(:,k+1)}$
$\beta_0 \leftarrow \beta$ ▷ Update β each step
end while
$z = B_0 \backslash I * q$
while $k < N$ **do**
$k \leftarrow k + 1$
$d \leftarrow -z$
Choose αCode stored in `stepsize.m` and `Armijo_stepsize.m`
$\beta \leftarrow \beta_0 + \alpha * d$
if $|g(\beta_0) - g(\beta)| < TOL$ **then**
break

end if
$ss \leftarrow \beta - \beta_0$
$s \leftarrow [s(:, 2 : m + 1), ss]$ ▷ Update the stored $m + 1$ vectors **s**
$yy \leftarrow \nabla g(\beta) - \nabla g(\beta_0)$
$y \leftarrow [y(:, 2 : m + 1), ss]$ ▷ Update the stored $m + 1$ vectors **y**
$q \leftarrow \nabla g(\beta_0)$
for $t = m + 1, \ldots, 1$ **do**
$a(t) \leftarrow \dfrac{s(:, t)' * q}{s(:, t)' * y(:, t)}$
$q \leftarrow q - a(t) * y(:, t)$
end for
for $t = 1, \ldots, m + 1$ **do**
$b \leftarrow \dfrac{y(:, t)' * z}{s(:, t) * y(:, t)}$
$z \leftarrow z + (a(t) - b) * s(:, t)$
end for
$\beta_0 \leftarrow \beta$ ▷ Update β each step
end while
return β ▷ The coefficient of original function

In this algorithm, the basic strategy in the procedure of limited implement is setting zeros matrix previously with length m in order to store the vectors **s** and **y** which are needed in the Eq.17. In this case, the necessary memory highly depends on the choice of m.

2.3 Choice of Step Size

2.3.1 Dichotomy Method

We implement a simple algorithm to decide the step size α in **Algorithm 1** and **Algorithm 2**.

Algorithm 3 Dichotomy Method

$\alpha \leftarrow 1$
while $g(\beta_0 + \alpha * d) \geq g(\beta_0)$ **do**
$\alpha \leftarrow \alpha/2$
if $\alpha < TOL$ **then**
break
end if
end while
return α

2.3.2 Successive Stepsize Reduction - Armijo Rule

Now we introduced another simple algorithm to decide the step size, which is called *Armijo Rule*. The former method is typically a simple version of *line search*. However, a great amount of computation is unavoidable in this method. Therefore, Armijo rule, to some extent, introduce slackness in the process of choosing α. It may be useful in this project

Algorithm 4 Armijo Rule

Set parameter s, γ, σ
while $g(\beta_0) - g(\beta_0 + \gamma * s * d) < -\sigma * \gamma * s * \nabla g(\beta_0)' * d$ **do**
$\gamma \leftarrow \gamma * \gamma$
if $\gamma < TOL$ **then**
break

6: **end if**
7: **end while**
8: **return** α

In this algorithm, σ is introduced to set a threshold for reduction. γ is reduction factor which is used to reduce step size when reduction of function is not obtained. s is set to decide the initial step size and usually constant.

3 Programming and Practice

We implement the main program and some necessary sub-programs for functions(g, ∇g, Jg, step size function, etc.).

3.1 Comparison of outcomes

First, we list the outcomes of BFGS and L-BFGS with $\boldsymbol{\beta}_1$ and $\boldsymbol{\beta}_1$ using dichotomy method in Table 2. Two initial values of $\boldsymbol{\beta}$ are listed in Table 1. Among the outcomes in Table 3, the corresponding value of $g(\boldsymbol{\beta})$ satisfies Eq.22 with 4 decimal places. In each table, k stands for iteration times.

$\boldsymbol{\beta}_1$	$\boldsymbol{\beta}_2$
94.9000	96.0000
0.0090	0.0096
90.1000	80.0000
113.0000	110.0000
20.0000	25.0000
73.8000	74.0000
140.0000	139.0000
20.0000	25.0000

Table 1: Two initial values of $\boldsymbol{\beta}$

	BFGS		L-BFGS($m = 3$)	
	$\boldsymbol{\beta}_1$	$\boldsymbol{\beta}_2$	$\boldsymbol{\beta}_1$	$\boldsymbol{\beta}_2$
$\boldsymbol{\beta}$	98.940084	98.940442	98.928996	98.942250
	0.010946	0.010946	0.010945	0.010947
	100.694660	100.695786	100.733022	100.725810
	111.636229	111.636241	111.662293	111.647174
	23.300469	23.300611	23.326685	23.307845
	73.704402	73.704674	73.634839	73.681483
	147.761646	147.761682	147.788621	147.726469
	19.668264	19.668295	19.644602	19.653323
k	19	19	57	56

Table 2: Comparison of Outcomes with Dichotomy Method

$$g(\boldsymbol{\beta}) = 1.2445 \times 10^{-3} \tag{22}$$

We need to point out that if we set the tolerance in the Line 17 of **Algorithm2** as 10^{-3}, the program will exit without obtaining the theoretical value in Eq.22. In order to fix the problem, we renew the tolerance into 10^{-5} and have the outcomes in Table 2.

In addition, to test the feasiblity of Armijo Rule in this problem, we use the first initial value $\boldsymbol{\beta}_1$. In fact, the outcomes highly depend on the values of γ and s if regarding s as constant. Set the default value of σ as 10^{-5}, we have the outcomes in Table 3.

	$s = 0.1$		$s = 0.2$	
	$\gamma = 0.1$	$\gamma = 0.2$	$\gamma = 0.1$	$\gamma = 0.2$
$\boldsymbol{\beta}$	97.864889	97.762552	97.925835	97.609050
	0.010794	0.010740	0.010775	0.010657
	82.461134	82.031508	84.674100	81.396586
	114.226500	114.216794	113.861398	114.211601
	28.562446	28.492106	27.613976	28.383327
	54.402614	54.461474	57.168552	54.498184
	142.550417	142.438017	142.813033	142.279511
	26.352572	26.285938	25.746692	26.178435
k	120	58	67	28
$g(\boldsymbol{\beta})$	9742.136663	9856.347223	9375.596966	10046.269753

Table 3: Comparison of Outcomes with Different Parameters

3.2 Analysis of Outcomes

3.2.1 Effect of Armijo Rule

In Table 3, we can see that in each circumstance, $g(\boldsymbol{\beta})$ converges to the same value with different iteration steps. However, it is contrary from the theoretical value in Eq.22.

The reason is shown in Figure 1. If the value of m is not big each step, it is easy for the algorithm to move the point $\boldsymbol{\beta}$ into a location which is close to another local minimum. It is highly possible with the conclusion in Table 3 that $\boldsymbol{\beta}$ has converged to another local minimum. However, if we examine the gradient of g on the point $\boldsymbol{\beta}$, it is significantly larger than that of convergent point by dichotomy method. Though Armijo Rule has been regarded as *Golden Standard* in choosing step size, it is not reliable in this problem with relatively small dataset.

We still use the dichotomy method for further research.

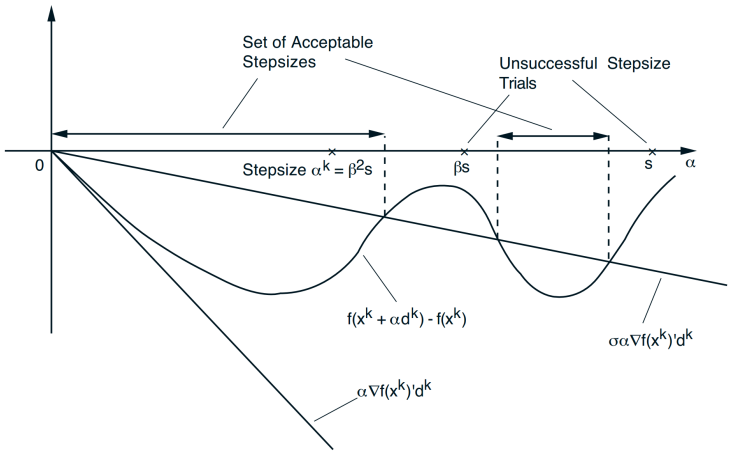

Figure 1: The Sketch of Armijo Rule[1]

3.2.2 Difference between Non-limited and Limited Method

As for BFGS method, we have introduced two versions — *Non-limited* with no recursion and *Limited* with m recursion. There are two interfaces which deserve attention:

(a) The convergence speed in L-BFGS method.

(b) The effect of different choice of m.

For part (a), as is shown in Table 2, L-BFGS method required much more iteration steps to obtain the minimum. In fact, in the process of implementing the program, we found out that the convergence speed is extremely slow in later iterations. Additionally, the tolerance 10^{-3} is not enough to guarantee that the iteration will terminate on the theoretical value in Eq.22. We reset the tolerance as 10^{-10}, then we can obtain the theoretical value after a great number of iterations.

If we set the threshold of iterations $N = 20$, obviously, L-BFGS method cannot meet the demand. However, in **Algorithm 2**, we have shown that L-BFGS method significantly save the memory for implementing the method, which is practical when processing mass of data instead of only 250 items in `Gauss3.dat`.

3.3 Further Research

3.3.1 Discussion on m

If we are interested in part (b) in **Section 3.2.2**, we can plot figures to illustrate the behaviour of different m in L-BFGS method. The code to plot the figures is stored in `LBFGSmtest.m`.

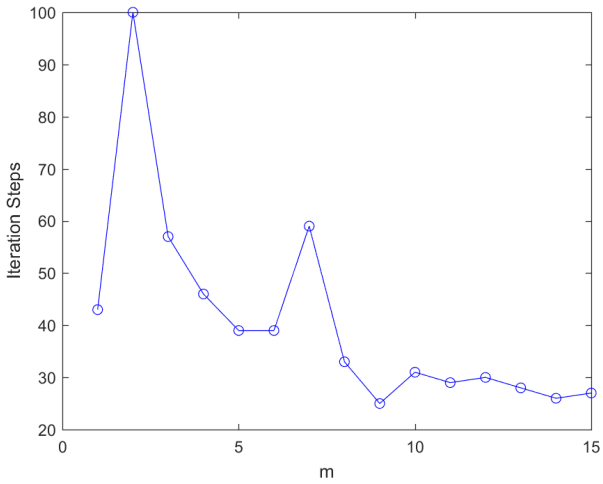

Figure 2: Behaviour of Different m when Initial Value is $\boldsymbol{\beta}_1$

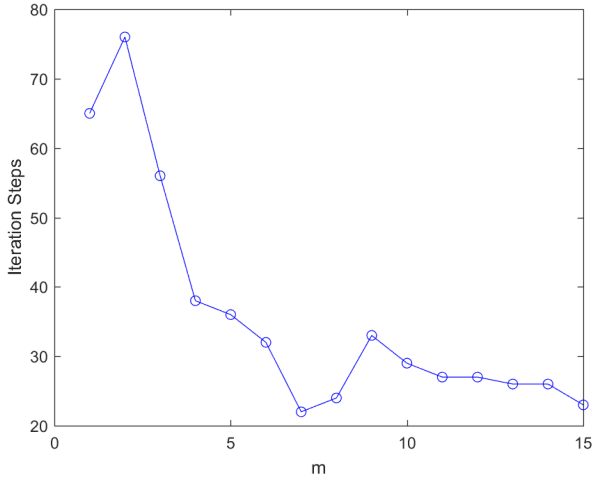

Figure 3: Behaviour of Different m when Initial Value is $\boldsymbol{\beta}_2$

In Figure 2 and Figure 3, the threshold of iterations $N = 100$. The figures provide the best choice of m among integers between 1 and 15. There are two possible explanations for the general decreasing trend:

(1) When m is larger, more information left for limited implement, which reduces requirement of iterations.

(2) When m is larger, more steps are implement by Non-limited implement, which means that the commence of limited implement is closer to the theoretical value.

3.3.2 Different Choice of $\mathcal{B}_0$

Notice that if we want to define a Hessian matrix as $\mathcal{B}_0$, a great amount of computation is need. Instead, if we adopt an alternative way to initialize $\mathcal{B}_0$

$$\mathcal{B}_0 = \mathcal{J}\mathcal{J}^T \tag{23}$$

where $\mathcal{J}$ is the Jacobian matrix of $\mathbf{f}(\mathbf{b}, \boldsymbol{\beta})$, we have

	BFGS		L-BFGS($m = 3$)	
	$\boldsymbol{\beta}_1$	$\boldsymbol{\beta}_2$	$\boldsymbol{\beta}_1$	$\boldsymbol{\beta}_2$
$\boldsymbol{\beta}$	98.940319	98.940443	98.940453	98.941893
	0.010946	0.010946	0.010946	0.010947
	100.697051	100.695636	100.690240	100.706264
	111.636475	111.636244	111.634348	111.651032
	23.300686	23.300575	23.298994	23.314587
	73.705004	73.704944	73.709810	73.652981
	147.762167	147.761697	147.759292	147.784747
	19.667564	19.668229	19.670642	19.663607
k	14	15	30	26

Table 4: Comparison of Outcomes with Dichotomy Method

From the outcome, generally, the new choice of $\mathcal{B}_0$ makes the algorithm more efficient and the convergence speed is faster.

Furthermore, if we adopt the Armijo Rule to choose step size again, we find out that after more than 100 iterations, $\boldsymbol{\beta}$ *can* be converged to the theoretical value in Eq.22.

4 Conclusion

Fron different choice of $\mathcal{B}_0$ in Eq.12 and Eq.23, we figure out that BFGS method implement faster than L-BFGS method. But from the procedure in L-BFGS method, it save memory at the cost of speed. Moreover, if we choose $\mathcal{B}_0$ as what in Eq.23, the convergence speed is generally faster. Each method can converge to the theoretical value in Eq.22. Surprisingly, the choice of $\mathcal{B}_0$ in Eq.23 makes Armijo Rule converges, while another choice does not. To some extent, Armijo Rule guarantee convergence, but the convergence speed is much slower.

References

[1] D. P. Bertsekas, *Nonlinear Programming*. Athena Scientific, 2016.