Block BFGS

A class of quasi-Newton methods for
unconstrained continuous optimisation

Candidate Number:
1013471
Word Count:
7481

A thesis presented in partial
completion of the degree of
Master of Mathematics
in Mathematics

Department of Mathematics
University of Oxford
Trinity 2020

Contents

Chapter 1

Introduction

Consider a real-valued function $f : \Omega \to \mathbb{R}$ on some set Ω. The field of optimisation concerns itself with answering the following simple question: for what values of Ω does f take its smallest values? In this work we consider the unconstrained minimisation of smooth functions $f : \mathbb{R}^n \to \mathbb{R}$.

In the centuries since Pierre de Fermat proposed his general approach for finding local maxima and minima of differentiable functions in 1629 [9], the field of optimisation has permeated a huge variety of mathematical disciplines. Optimisation techniques can be utilised across engineering disciplines [7, 5], and they are fundamentally important in training statistical and machine learning models [21]. There is therefore a great demand for techniques that push the boundaries in this area, that produce solutions with the greatest possible efficiency.

We will describe and expand on a technique given by R. Schnabel [18], which itself generalises the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm discovered independently by those honoured in its name [3, 10, 12, 20]. We will refer to such generalisations as Block BFGS methods, following the example set by Gao and Goldfarb [11].

BFGS is itself a powerful algorithm that is widely used when second order derivatives are computationally expensive to obtain, or otherwise unavailable. For example, stochastic variants of the BFGS method are shown to be effective for online optimisation [19, 4]. In recent years, Block BFGS methods have received some well-deserved attention both in the deterministic setting [11] and for their potential to adapt to stochastic optimisation for applications in machine learning [14], the experiments in both of which show promising results. The methods in both of these works aim to overcome a shortcoming in the method of Schnabel, in which alterations must be made in order to maintain symmetry of an inverse-Hessian estimate. They propose replacing the use of *secant* equations, which will be explained later, with *sketching* equations, which require the computation of Hessian-vector products. This does indeed provide a solution to the symmetry problem, and they argue that since Hessian-vector products are not as computationally expensive as calculating the full Hessian, and such calculations can be completed in parallel, the performance gain may be worth

this extra expense.

All of the methods considered in this work use only first order derivative information. Large dimension problems pose issues for using second order information, and there are many applications which could benefit from Block BFGS methods, but where second order derivatives are simply unavailable. The study of first order methods is therefore worthwhile.

In Chapter 2 we formally define the problem of unconstrained continuous optimisation, and outline the benefits and disadvantages of some classical methods for solving this problem, focusing mainly on the class of quasi-Newton algorithms. We provide necessary and sufficient conditions under which we can expect superlinear convergence of quasi-Newton methods, and we let this theory guide us to derive the classical BFGS method. In Chapter 3 we follow and expand on the work of Schnabel [18] to derive a class of Block BFGS methods, and discuss the conditions under which we can expect linear and superlinear convergence from such methods. We give several examples of such methods in detail, and discuss the differences in performance we may expect to see from each.

In Chapter 4 we introduce a framework for comparing the performance of optimisation algorithms on sets of problems, and implement our Block BFGS methods and compare their performance with the BFGS method on an array of standard test problems. We demonstrate benefits of using Schnabel's method [18] over the classical BFGS method, and also show promising results of our own variants, inspired by the idea of incorporating sampled gradients to obtain local second-order information [1].

Chapter 2

Optimisation Preliminaries

In this chapter we review definitions and results necessary to build and explore powerful methods for optimisation. We define a standard line search method and give results concerning its convergence. We then move on to see how these methods con be incorporated into other optimisation routines such as steepest descent, Newton's method, and primarily focussing on quasi-Newton methods. We further derive the BFGS method and discuss its theoretical benefits. Throughout this chapter we observe several definitions and theorems from [15, 13, 6] without proof.

2.1 Problem Definitions

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuously differentiable function in n variables, and assume that f is bounded below. We aim to find x^* that solves the following:

$$x^* = \arg\min_{x \in \mathbb{R}^n} f(x). \tag{2.1.1}$$

A solution to this problem is known as a *global* minimiser. In general, computing a global minimiser proves impossible, and we must settle for weaker form of solution.

Definition 2.1.1. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuous function. $x^* \in \mathbb{R}$ is a **local minimiser** for f if there exists $\epsilon > 0$ such that for all $x \in \mathcal{B}(x^*, \epsilon)$ we have*

$$f(x^*) \leq f(x),$$

where $\mathcal{B}(x^*, \epsilon) \subseteq \mathbb{R}^n$ is the open ball of radius ϵ centred at x^*.

Definition 2.1.2. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuous function. f is **convex** on some set $\mathcal{U} \subseteq \mathbb{R}^n$ if for all $x, y \in \mathcal{U}$ and all $\lambda \in [0, 1]$ we have*

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda) f(y).$$

Lemma 2.1.3. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuously differentiable function, which is convex on some open set $\mathcal{U} \subseteq \mathbb{R}^n$. If $x^* \in \mathcal{U}$ satisfies $\nabla f(x^*) = 0$, then x^* is a local minimiser for f.*

Definition 2.1.4. *Let X be a real symmetric matrix. X is **positive definite** if every eigenvalue of X is strictly positive.*

Lemma 2.1.5. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a twice continuously differentiable function, and suppose $x \in \mathbb{R}^n$ gives a positive definite Hessian $\nabla^2 f(x)$. Then there exists an open set containing x on which f is convex.*

Theorem 2.1.6. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a twice continuously differentiable function. Suppose $x^* \in \mathbb{R}^n$ satisfies $\nabla f(x^*) = 0$ with $\nabla^2 f(x^*)$ positive definite. Then x^* is a local minimiser of f.*

2.2 Line search

Definition 2.2.1. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuous function. A direction $p \in \mathbb{R}^n$ is a **descent direction** for f at $x \in \mathbb{R}^n$ if there exists $\epsilon > 0$ such that for all $\eta \in (0, \epsilon)$, $f(x + \eta p) < f(x)$.*

Lemma 2.2.2. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuously differentiable function. A direction $p \in \mathbb{R}^n$ is a descent direction for f at $x \in \mathbb{R}^n$ if and only if*

$$\nabla f(x)^T p < 0.$$

Suppose we are at $x \in \mathbb{R}^n$ and we have some descent direction $p \in \mathbb{R}^n$ for f at x. A line search method is some method of deciding how far along that direction we should move in order to decrease f as much as possible. That is, we seek to find $\alpha > 0$ minimising

$$\phi(\alpha) = f(x + \alpha p).$$

Solving this sub-problem is known as *exact* line search, and is typically very expensive to implement, especially considering it would have to be solved many times in a typical optimisation problem. For this reason we will consider only *inexact* line search methods, which simply aim to improve on x *sufficiently* at each iteration.

2.2.1 Wolfe Line Search

One widely used condition for inexact line search is the *Armijo* condition:

$$f(x + \alpha p) \leq f(x) + c_1 \alpha \nabla f(x)^T p, \tag{2.2.1}$$

for some $c_1 \in (0, 1)$. Enforcing (2.2.1) ensures that α will only be allowed to be large if the resultant decrease in f is proportionally large.

One drawback of (2.2.1) is that any sufficiently small α would be accepted, which can be seen by considering the first-order Taylor series of $\phi(\alpha)$ about

$\alpha = 0$ and observing that p is a descent direction. For this reason (2.2.1) is commonly used alongside the *Curvature* condition:

$$\nabla f(x + \alpha p)^T p \geq c_2 \nabla f(x)^T p, \tag{2.2.2}$$

for some $c_2 \in (c_1, 1)$. To see why we may want (2.2.2) to hold, note that the left hand side is the derivative of ϕ with respect to α. When this is large and negative we can expect to decrease ϕ further by increasing α, so we want a lower bound on this quantity to ensure we take large steps when we can. (2.2.1 and 2.2.2) together define the Wolfe line search criteria.

2.2.2 Analytical Results

We will now review some basic results of the Wolfe line search method. Proofs can be found in Nocedal and Wright, Chapter 3 [15].

Lemma 2.2.3 (Existence)**.** *Suppose $f : \mathbb{R}^n \to \mathbb{R}$ is continuously differentiable, let p be a descent direction for f at x and assume f is bounded below along the ray $\{x + \alpha p : \alpha > 0\}$. If $0 < c_1 < c_2 < 1$ then there exists a step length α satisfying the Wolfe conditions (2.2.1 and 2.2.2).*

We obtain a global convergence result for this line search method by considering the angle θ between the search direction, p, and the steepest descent direction $-\nabla f(x)$, defined by:

$$cos\theta = \frac{-\nabla f(x)^T p}{\|\nabla f(x)\|_2 \|p\|_2} \tag{2.2.3}$$

Lemma 2.2.4 (Global Convergence)**.** *Consider a sequence $(x_k)_{k=0}^{\infty}$ generated by the relation $x_{k+1} = x_k + \alpha_k p_k$ for some $x_0 \in \mathbb{R}^n$ where p_k is a descent direction for f at x_k and α_k satisfies the Wolfe conditions (2.2.1 and 2.2.2). Suppose f is bounded below and is continuously differentiable on an open set $\mathcal{N}$ containing the level set $\mathcal{L} = \{x : f(x) \leq f(x_0)\}$. Assume further that ∇f is Lipschitz continuous on $\mathcal{N}$. Define the sequence of angles θ_k as in (2.2.3). Then*

$$\sum_{k=0}^{\infty} cos^2\theta_k \|\nabla f(x_k)\| < \infty. \tag{2.2.4}$$

Note that this result can be used to show global convergence to a local minimum of f provided the method of choosing p_k prevents $cos\theta_k \to 0$ as $k \to \infty$, that is provided the search directions do not become orthogonal to the steepest descent direction. In this case we must have $\limsup_{k\to\infty} \|\nabla f(x_k)\| = 0$ in order to satisfy (2.2.4).

2.3 Finding Search Directions

Now that we know how to improve on x_k given a descent direction, let us turn our attention to methods of obtaining such directions.

2.3.1 Steepest Descent

Perhaps the simplest way to ensure that our search direction p_k does not become orthogonal to $-\nabla f(x_k)$ is to choose $p_k = -\nabla f(x_k)$. This gives us the steepest descent method and we get global convergence guaranteed by Lemma 2.2.4. However, the steepest descent method enjoys only a linear rate of convergence [15] leading to very slow convergence for many problems, particularly when the Hessian $\nabla^2 f$ is ill-conditioned near the optimum. In order to improve this convergence rate we must make use of curvature information, which leads us to the following methods.

2.3.2 Newton's Method

Consider a quadratic model for f around x:

$$m_N(p) = f(x) + p^T \nabla f(x) + \frac{1}{2} p^T \nabla^2 f(x) p, \tag{2.3.1}$$

and assume $\nabla^2 f(x)$ is positive definite, which will hold sufficiently close to any local minimiser. (2.3.1) has a unique minimiser $p = -\left(\nabla^2 f(x)\right)^{-1} \nabla f(x)$, and Newton's method uses this as its search direction in the Wolfe line search method. It is shown by Nocedal [15] that provided $\nabla^2 f(x_k)$ remains *sufficiently* positive definite as $k \to \infty$ that we will have convergence of Newton's method in some neighbourhood of the minimiser, and convergence will be asymptotically quadratic. The drawback is, of course, having to calculate $\left(\nabla^2 f(x)\right)^{-1} \nabla f(x)$ at each iteration. This operation first requires computation of $\nabla^2 f(x)$ and then a further linear solve of the system

$$\nabla^2 f(x) z = \nabla f(x).$$

This becomes intractable as the dimension n of the problem grows.

2.3.3 Quasi-Newton Methods

Quasi-Newton Methods aim to strike a balance between the two methods above. We attempt to use an approximate quadratic model of f:

$$m_{QN}(p) = f(x) + p^T \nabla f(x) + \frac{1}{2} p^T B p, \tag{2.3.2}$$

where B is some symmetric, positive-definite matrix approximating the true Hessian. (2.3.2) has a unique minimiser

$$p = -B^{-1} \nabla f(x). \tag{2.3.3}$$

So we consider search directions of this form.

The computational cost can be greatly reduced by not directly computing second-order derivatives, and many such methods (we will see one later) are able to approximate B^{-1} directly so that a linear solve is not necessary. We

will now consider some results regarding the convergence of such methods which show that under fairly reasonable assumptions on B we will have superlinear convergence of such methods. Proofs of these results can be found in Nocedal and Wright [15].

Lemma 2.3.1. *Suppose f is twice continuously differentiable and consider an iteration of the form $x_{k+1} = x_k + \alpha_k p_k$ where p_k given by (2.3.3) is a descent direction for f at x_k and α_k satisfies the Wolfe conditions (2.2.1 and 2.2.2) with $c_1 \leq 1/2$. Suppose the sequence $(x_k)_{k=0}^{\infty}$ converges to some x^* satisfying $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is positive definite, and the search direction satisfies*

$$\lim_{k \to \infty} \frac{\left\| \left(B_k - \nabla^2 f(x^*) \right) p_k \right\|}{\|p_k\|} = 0. \tag{2.3.4}$$

Then:

1. *there exists k_0 such that for all $k > k_0$, $\alpha_k = 1$ will satisfy the Wolfe conditions (2.2.1 and 2.2.2);*
2. *and if $\alpha_k = 1$ for all $k > k_0$ then the sequence $(x_k)_{k=0}^{\infty}$ converges to x^* superlinearly.*

Lemma 2.3.2. *Under the same assumptions on f and the sequence of iterates $(x_k)_{k=0}^{\infty}$ as in Lemma 2.3.1, but further assuming that $\alpha_k = 1$ for all k, then $(x_k)_{k=0}^{\infty}$ converges superlinearly if and only if (2.3.4) holds.*

Lemmas 2.2.4, 2.3.1 and 2.3.2 together give us a very powerful convergence result for Quasi-Newton methods, under the assumption that we can ensure (2.3.4) holds. Under reasonable conditions on f, provided we use Hessian approximations B_k such that the search directions do not become asymptotically orthogonal to the steepest descent direction, we have convergence guaranteed by Lemma 2.2.4. We can then apply 2.3.1 to see that eventually $\alpha_k = 1$ will be admissible, so if our method chooses $\alpha_k = 1$ whenever admissible we get superlinear convergence by Lemma 2.3.2.

A key observation to make is that (2.3.4) is simply saying that asymptotically, the Hessian approximation B acts the same as the true Hessian along the search directions. This heuristic guides us in what Hessian approximations we will consider.

2.4 BFGS

2.4.1 The Secant Equation

Once again consider a quadratic model m_k given by (2.3.2) for a quasi-Newton method at iterate x_k, and search direction p_k given by (2.3.3), and new iterate $x_{k+1} = x_k + \alpha_k p_k$, where α_k satisfies the Wolfe conditions (2.2.1 and 2.2.2). At each iteration, we wish to use new curvature information to update our Hessian estimate. One way to do this is to ensure that the model at the following

iteration, m_{k+1}, interpolates ∇f at both x_{k+1} and x_k. This idea has the added benefit of only using gradients that we have already calculated, which helps keep computational costs down.

Of course the first of these conditions is simply

$$\nabla f(x_{k+1}) = \nabla m_{k+1}(0),$$

which always holds. The second leads to a very useful condition on the updated Hessian estimate. Let $s_k = x_{k+1} - x_k$ and $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$. Then we want

$$\begin{aligned}\nabla f(x_k) &= \nabla m_{k+1}(-s_k) \\ &= \nabla f(x_{k+1}) - B_{k+1}s_k,\end{aligned}$$

which rearranging gives the *Secant Equation*:

$$B_{k+1}s_k = y_k. \tag{2.4.1}$$

It is worth noting that the true Hessian $\nabla^2 f(x_k)$ asymptotically satisfies the secant equation as x_k approaches a local minimum x^* with positive definite Hessian $\nabla^2 f(x^*)$ [15]. Therefore enforcing that the Hessian approximation satisfies (2.4.1) is to ensure that this approximation acts similarly to the true Hessian along the search directions asymptotically. Recalling the condition (2.3.4) for superlinear convergence of quasi-Newton methods we see another reason why this would be beneficial.

We must also ensure that the secant equation can be satisfied by a symmetric positive definite matrix. The only reason there would not be a solution is if $s_k^T y_k < 0$, in which case

$$0 \leq s_k^T B_{k+1} s_k = s_k^T y_k < 0$$

would clearly not be possible. So provided we have $s_k^T y_k \geq 0$ for all k we will always be able to update the Hessian approximation to satisfy the new secant equation. This is where we see the benefit of the Wolfe line search, as the condition (2.2.2) gives us:

$$\begin{aligned}&\nabla f(x_{k+1})^T s_k \geq c_2 \nabla f(x_k)^T s_k \\ \Rightarrow\ &\nabla f(x_{k+1})^T s_k - \nabla f(x_k)^T s_k \geq c_2 \nabla f(x_k)^T s_k - \nabla f(x_k)^T s_k \\ \Rightarrow\ &\left(\nabla f(x_{k+1})^T - \nabla f(x_k)^T\right) s_k \geq (c_2 - 1)\nabla f(x_k)^T s_k \\ \Rightarrow\ &y_k^T s_k \geq 0 \text{ since } c_2 \in (c_1, 1) \text{ and } s_k \text{ is a descent direction.}\end{aligned}$$

Thus the Wolfe line search guarantees the existence of such an update to the Hessian approximation.

2.4.2 The BFGS Update

Since the quasi-Newton update requires the use of B^{-1} rather than B, let us consider approximating $H = B^{-1}$ directly. Suppose we are at $x \in \mathbb{R}^n$ with symmetric, positive-definite inverse-Hessian approximation H. Let $p = -H\nabla f(x)$

and $x_+ = x + \alpha p$ where α satisfies the Wolfe conditions (2.2.1 and 2.2.2). Let s and y be defined as before. We wish to find an updated symmetric inverse-Hessian approximation that satisfies the latest secant equation, and maintains as much information about the previous gradients as possible. A natural way to frame this problem is as follows:

$$min \, \|H_+ - H\| \tag{2.4.2}$$
$$\text{subject to } H_+ y = s \tag{2.4.3}$$
$$\text{and } H_+^T = H_+. \tag{2.4.4}$$

Of course in order to solve this problem we have to choose a matrix norm, and different choices of norm give rise to different quasi-Newton updates, each with their own properties, advantages and disadvantages. The BFGS update is widely regarded to be the best, and it is found by using the weighted Frobenius norm

$$\begin{aligned} \|X\|_W^2 &= \left\| W^{\frac{1}{2}} X W^{\frac{1}{2}} \right\|_F^2 \\ &= Tr\left(X^T W^T X W\right) \end{aligned}$$

where W is any matrix satisfying

$$Ws = y. \tag{2.4.5}$$

This results in the BFGS update

$$H_+ = \frac{ss^T}{y^T s} + \left(I - \frac{sy^T}{y^T s}\right) H \left(I - \frac{ys^T}{y^T s}\right). \tag{2.4.6}$$

The necessity of the condition (2.4.5) is demonstrated in the derivation of this result, which is given later in more generality. The use of the weighted Frobenius norm here results in a scale-invariant method [15] so that the solution does not depend on the units of the problem, which is desirable. We are thus able to state the BFGS algorithm in full.

Algorithm 1 BFGS

Input: Function f ; Initial Guess x ; Tolerance tol ; inverse-Hessian H
Output: Solution x^*

1: **while** $\|\nabla f(x)\| > tol$ **do**
2: $x_+ \leftarrow x - \alpha H \nabla f(x)$ where α satisfies the Wolfe conditions (2.2.1 and 2.2.2)
3: $s \leftarrow x_+ - x$
4: $y \leftarrow \nabla f(x_+) - \nabla f(x)$
5: Update H by (2.4.6)
6: $x \leftarrow x_+$
7: **end while**
8: $x^* \leftarrow x$

Chapter 3

Block BFGS

The remainder of this work considers how we may use further first-order gradient information to improve on the inverse Hessian update given by the BFGS method. Suppose that in stead of just one secant equation, we could form multiple. That is we had q direction vectors, $(s_i)_{i=1}^q$, and gradient differences $(y_i)_{i=1}^q$ along those directions. Then we could enforce the q secant equations:

$$H_+ Y = S \tag{3.0.1}$$

where

$$S = \begin{bmatrix} s^{(0)} & s^{(1)} & \cdots & s^{(q-1)} \end{bmatrix},$$

and

$$Y = \begin{bmatrix} y^{(0)} & y^{(1)} & \cdots & y^{(q-1)} \end{bmatrix}.$$

We would like to use

$$s_k^{(j)} = x_{k+1} - x_{k-j} \tag{3.0.2}$$

and

$$y_k^{(j)} = \nabla f(x_{k+1}) - \nabla f(x_{k-j}), \tag{3.0.3}$$

so that the corresponding secant equations would then ensure that the quadratic quasi-Newton model of the objective function interpolates the gradient of f at the previous q iterates.

Using the same reasoning as for the BFGS method, we can build a minimisation problem to find the optimal update to the inverse Hessian approximation that satisfies these secant equations (3.0.1). We must, however, enforce an additional assumption that the matrix $Y^T S$ is symmetric. Further, in order to guarantee that the update H_+ maintains positive-definiteness, we must assume that $Y^T S$ is positive-definite. In practice, neither of these assumptions will hold for the natural choice of S and Y given by (3.0.2) and (3.0.3), and we will discuss how we can deal with this later.

3.1 Block BFGS Update Derivation

Theorem 3.1.1. *Consider the following minimisation problem:*

$$min \, \|H_+ - H\|_W^2 \tag{3.1.1}$$
$$subject \ to \ H_+ Y = S \tag{3.1.2}$$
$$and \ H_+^T = H_+ \tag{3.1.3}$$

where we are using the weighted Frobenius norm given by

$$\|X\|_W^2 = Tr(X^T W^T X W),$$

where W is a symmetric non-singular matrix satisfying $WS = Y$. Assume that the matrix $Y^T S$ is symmetric. Then the solution to this minimisation problem is given by

$$H_+ = S(Y^T S)^{-1} S^T + (I - S(Y^T S)^{-1} Y^T) H (I - Y(Y^T S)^{-1} S^T). \tag{3.1.4}$$

Proof. The Lagrangian for this problem is

$$\begin{aligned}
\mathcal{L}(H_+, A, \Lambda) &= Tr\left((H_+ - H)^T W^T (H_+ - H) W\right) \\
&\quad - \sum_{i<j} \alpha_{ij} (H_{+ij} - H_{+ji}) - \sum_{i,j} \lambda_{ij} (H_+ Y - S)_{ij} \\
&= Tr\left((H_+ - H)^T W^T (H_+ - H) W\right) \\
&\quad - \sum_{i,j} \alpha_{ij} (H_{+ij}) - Tr(\Lambda^T (H_+ Y - S)) \text{ letting } A^T = -A \\
&= Tr\left((H_+ - H)^T W^T (H_+ - H) W\right) \\
&\quad - Tr(A^T H_+) - Tr(\Lambda^T (H_+ Y - S))
\end{aligned}$$

so that taking the derivative with respect to H_+ [17] we obtain

$$\frac{d\mathcal{L}}{dH_+} = 2W(H_+ - H)W - A - \Lambda Y^T. \tag{3.1.5}$$

We now seek to solve $\frac{d\mathcal{L}}{dH_+} = 0$, under the following assumptions:

$$\begin{gathered}
\text{(i) } A^T = -A; \quad \text{(ii) } W^T = W; \quad \text{(iii) } H_+^T = H_+; \quad \text{(iv) } H^T = H; \\
\text{(v) } H_+ Y = S; \quad \text{(vi) } WS = Y; \quad \text{(vii) } Y^T S = S^T Y.
\end{gathered} \tag{3.1.6}$$

Taking the transpose of (3.1.5) and using (3.1.6(i)) we obtain

$$4W(H_+ - H)W - \Lambda Y^T - Y\Lambda^T = 0. \tag{3.1.7}$$

Then post-multiplying by S and using (3.1.6(v) and 3.1.6(vi)) we get

$$4(Y - WHY) - \Lambda Y^T S - Y\Lambda^T S = 0, \tag{3.1.8}$$

and then pre-multiplying by S^T and using (3.1.6(vi)) again we get

$$4(S^TY - Y^THY) - S^T\Lambda Y^TS - S^TY\Lambda^TS = 0, \tag{3.1.9}$$

and we can see that

$$\Lambda^TS = 2(S^TY)^{-1}Y^T(S - HY) \tag{3.1.10}$$

is a solution to (3.1.9), using (3.1.6(vii)). Then substituting (3.1.10) into (3.1.8) we can rearrange to get

$$\Lambda = \big(4(Y - WHY) - 2Y(Y^TS)^{-1}Y^T(S - HY)\big)\,(Y^TS)^{-1}. \tag{3.1.11}$$

From here we can substitute (3.1.11) into (3.1.7) and rearrange to see:

$$\begin{aligned} H_+ = \quad & H + \frac{1}{4}W^{-1}(\Lambda Y^T + Y\Lambda^T)W^{-1} \\ = \quad & H + 2S(Y^TS)^{-1}S^T - HY(Y^TS)^{-1}S^T - S(Y^TS)^{-1}Y^TH \\ & - S(Y^TS)^{-1}S^T + S(Y^TS)^{-1}Y^THY(Y^TS)^{-1}S^T \\ = \quad & S(Y^TS)^{-1}S^T + (I - S(Y^TS)^{-1}Y^T)H(I - Y(Y^TS)^{-1}S^T). \end{aligned}$$

□

It is worth noting a few things here. First, provided we assume (3.1.6(vi)), we have a solution independent of W. This generality of the update formula (3.1.4) indicates to us how powerful the result may be. We can also see that provided Y^TS and H are symmetric, H_+ must also be symmetric. Further, provided Y^TS and H are positive-definite, H_+ is positive-definite, noting that it is the sum of two conjugated positive-definite matrices.

One could wonder whether there is an update satisfying the secant equations (3.0.1) that maintains positive-definiteness without the additional assumption of Y^TS being positive-definite. However we can easily see that

$$\begin{aligned} H_+ \text{ positive-definite } & \Rightarrow Y^TH_+Y \text{ positive-definite} \\ & \Rightarrow Y^TS \text{ positive-definite by (3.1.6(v)).} \end{aligned}$$

So in fact there can be no such update unless Y^TS is positive-definite, so we will next consider methods of perturbing Y to enforce this requirement.

3.2 Enforcing Symmetry

Schnabel gives several methods of perturbing Y by some small (in norm) matrix ΔY such that $(Y + \Delta Y)^TS$ is symmetric [18], and we derive and compare them here. Other works consider using Hessian-vector products to create the matrix $Y = \nabla^2 f(x)S$ [11, 1], and this method does indeed result in symmetric Y^TS. In fact, whenever the true Hessian $\nabla^2 f(x)$ is positive definite, this guarantees Y^TS

is positive definite also. However we will avoid such methods as the Hessian-vector product becomes computationally expensive for large problems, and we will use only first order information.

All of the methods considered here are derived in a similar way, so we will first prove a result that can be utilised for all of them.

Lemma 3.2.1. *The optimisation problem*

$$\begin{aligned} &\min \|X\|_W^2 = tr\left(W^T X^T W X\right) \\ &\textit{subject to } X^T A = B \end{aligned} \tag{3.2.1}$$

for non-singular symmetric W, and a matrix A with full rank, has solution

$$X = W^{-1} A \left(A^T W^{-1} A\right)^{-1} B^T$$

Proof. The Lagrangian for this problem is

$$\mathcal{L}(X, \Lambda) = tr\left(W^T X^T W X - \Lambda^T \left(X^T A - B\right)\right)$$

so taking the derivative [17] we see

$$\frac{d\mathcal{L}}{dX} = 2WXW - A\Lambda^T = 0 \tag{3.2.2}$$

so that we must have by premultiplying by A^T and rearranging:

$$\Lambda^T = 2\left(A^T A\right)^{-1} A^T W X W.$$

Then plugging back into (3.2.2) and pre and post multiplying by W^{-1} we get

$$X = W^{-1} A \left(A^T A\right)^{-1} A^T W X. \tag{3.2.3}$$

Then we can pre-multiply (3.2.3) by A^T and use the problem constraint (3.2.1) to see

$$B^T = A^T W^{-1} A \left(A^T A\right)^{-1} A^T W X$$

and then rearranging gives us

$$A^T A \left(A^T W^{-1} A\right)^{-1} B^T = A^T W X$$

which we can plug into (3.2.3) and cancel to get

$$X = W^{-1} A \left(A^T W^{-1} A\right)^{-1} B^T$$

□

3.2.1 Smallest Perturbation Method

First note the following expression can be found:

$$Y^T S - S^T Y = -L + L^T,$$

where L is a strictly lower triangular matrix. We are searching for some ΔY such that $(Y + \Delta Y)^T S$ is symmetric, and we note that if $\Delta Y^T S = L$ then we have satisfied this requirement. We may also choose the first column of ΔY to be zero since L has zeros in its top row, thus leaving the first secant equation unchanged. We therefore consider finding the smallest (in some norm) ΔY satisfying $\Delta Y^T S = L$. We consider using the weighted Frobenius norm again.

Lemma 3.2.2. *The solution to the minimisation problem*

$$\begin{aligned} &\min \|\Delta Y\|_W^2 = tr\left(W^T \Delta Y^T W \Delta Y\right) \\ &\textit{subject to } \left(\Delta Y\right)^T S = L \end{aligned}$$

is

$$\Delta Y = W^{-1} S \left(S^T W^{-1} S\right)^{-1} L^T$$

This is a direct application of Lemma 3.2.1. We consider two possible weightings that give usable formulae:

$$W = I \qquad \Rightarrow \qquad \Delta Y = S \left(S^T S\right)^{-1} L^T \tag{3.2.4}$$

$$\text{Any } W \text{ satisfying } WY = S \qquad \Rightarrow \qquad \Delta Y = Y \left(S^T Y\right)^{-1} L^T. \tag{3.2.5}$$

It is noted by Schnabel that (3.2.5) may be more appropriate for use in the Block BFGS update since it is similar to the norm by which we find this update [18].

3.2.2 Prioritising Recent Information

Thus far we have not considered how we are obtaining the matrices S and Y. If there is no ordering to their columns then the approach above seems sensible. If, however, we are updating the matrices with new columns added to the front as we progress through the optimisation, then it would make sense to prioritise minimising the change to the earlier columns of Y at the expense of losing less relevant older information. This line of thinking leads us to form the following algorithm.

Throughout the following, given a matrix X, X_j is the vector forming the j^{th} column of X, and $X_{<j}$ is the matrix formed by taking the first $j-1$ columns of X.

Algorithm 2 Prioritised Smallest Perturbation

1: Set $\left(\Delta Y\right)_1 \leftarrow 0$
2: **for** $j = 2, \cdots, q$ **do**
3: Solve $\min\|\delta_j\|$ subject to $\left(Y_j + \delta_j\right)^T S_{<j} = S_j^T \left(Y + \Delta Y\right)_{<j}$
4: Set $\left(\Delta Y\right)_j \leftarrow \delta_j$
5: **end for**

Algorithm 2 iteratively finds the minimal column vector $(\Delta Y)_j$ that ensures the leading $(j \times j)$-minor of $(Y + \Delta Y)^T S$ will be symmetric, and we hope that this will enable us to preserve as much of the more recent curvature information as possible.

Once again we have to select some matrix norm to minimise with respect to, and we choose to consider the weighted Frobenius norm as before. Applying Lemma 3.2.1, we see that Line (3) of Algorithm 2 has a closed form solution in this case:

$$\delta_j = W^{-1} S_{<j} \left(S_{<j}^T W^{-1} S_{<j} \right)^{-1} \left[S_j^T \left(Y + \Delta Y \right)_{<j} - Y_j^T S_{<j} \right]^T. \tag{3.2.6}$$

We consider two possible weightings that give usable formulae. If we use $W = I$ we obtain

$$\delta_j = S_{<j} \left(S_{<j}^T S_{<j} \right)^{-1} \left[S_j^T \left(Y + \Delta Y \right)_{<j} - Y_j^T S_{<j} \right]^T. \tag{3.2.7}$$

If in stead we use some W satisfying $W(Y + \Delta Y)_{<j} = S_{<j}$ we get

$$\delta_j = (Y + \Delta Y)_{<j} \left(S_{<j}^T \left(Y + \Delta Y \right)_{<j} \right)^{-1} \left[S_j^T \left(Y + \Delta Y \right)_{<j} - Y_j^T S_{<j} \right]^T. \tag{3.2.8}$$

We will see in Chapter 4 whether we can find any significant differences in the performance of these methods.

3.3 Enforcing Positive-Definiteness

We noted in chapter 2 that the BFGS update (2.4.6) was valid provided $y^T s > 0$, and this was guaranteed to hold by using the Wolfe line search. Unfortunately, simply implementing the Wolfe line search will be insufficient to ensure the Block BFGS update is valid. For this we require $Y^T S$ is positive-definite, and the Wolfe line search would simply guarantee that the first diagonal entry of $Y^T S$ is positive. While this does *help* us as it guarantees we have at least one useable secant equation, it is not sufficient. One practical method of overcoming this issue is by modifying the algorithm for Cholesky factorisation.

Definition 3.3.1. *For a symmetric, positive-definite matrix X, the **Cholesky Factorisation** of X is*

$$X = LL^T$$

where L is a lower triangular matrix.

Algorithm 3 Modified Cholesky Decomposition

Input: Symmetric matrix $A \in \mathbb{R}^{q \times q}$.
Output: Lower triangular $L \in \mathbb{R}^{q \times q}$; $BadColumns$.
$BadColumns \leftarrow \emptyset$
$L \leftarrow 0 \in \mathbb{R}^{q \times q}$
for $i = 1, \cdots, q$ **do**
  for $j = 1, \cdots, i$ **do**
    $s = \sum_{k=1}^{j} L_{ik} L_{jk}$
    if $A_{ii} \leq s$ **then**
      $L_{ik} \leftarrow 0$ for all k
      $BadColumns \leftarrow BadColumns \cup \{i\}$
    else if $j \in BadColumns$ **then**
      $L_{ij} \leftarrow 0$
    else if $i = j$ **then**
      $L_{ij} \leftarrow \sqrt{A_{ii} - s}$
    else
      $L_{ij} \leftarrow \left(L_{jj} \left(A_{ij} - s\right)\right)^{-1}$
    end if
  end for
end for
Delete all rows and columns in $BadColumns$ from L.

We apply Algorithm 3 to the matrix $Y^T S$ and remove the columns in $BadColumns$ from S and Y. Then L is the lower triangular matrix that gives the Cholesky factorisation

$$LL^T = Y^T S.$$

Since the Block BFGS update requires us to multiply some matrix by $\left(Y^T S\right)^{-1}$, the most effective way to do this is via a Cholesky factorisation and performing linear solves. Thus this method of enforcing positive-definiteness doesn't add significant additional computation.

3.4 The Block BFGS Methods

We are now in a position to state a Block BFGS algorithm that each of our variants will follow.

Algorithm 4 General Block BFGS

Input: Function f ; Initial Guess x ; Tolerance tol ; Inverse-Hessian H
Output: Solution x^*

while $\|\nabla f(x)\| > tol$ **do**
  $x,\ S,\ Y \leftarrow TakeSteps(x, H)$
  $\tilde{Y} \leftarrow Y + \Delta Y$, ΔY obtained by one of (3.2.4), (3.2.5), or by Algorithm 2 with one of (3.2.7) or (3.2.8).
  $L,\ BadColumns \leftarrow ModifiedCholesky(\tilde{Y}^T S)$
  Delete $BadColumns$ from $\tilde{Y}$ and S
  Update H by (3.1.4) with $\tilde{Y}$ and S.
end while
$x^* \leftarrow x$

We will explore different methods $TakeSteps$ of taking steps while obtaining new secant equations.

3.4.1 Block-BFGS

The first method, which we will refer to as Block-BFGS is the method considered by Schnabel [18], and is similar to the Block-BFGS method that was considered by Gao and Goldfarb [11]. This method takes q steps using the same inverse-Hessian approximation, building S and Y via (3.0.2) and (3.0.3) in the process, followed by the inverse-Hessian update. $TakeSteps$ for this method is given by the following algorithm.

Algorithm 5 Block-BFGS $TakeSteps$

Input: Iterate x ; inverse-Hessian H
Output: New iterate x ; New S ; New Y

$X, G \in \mathbb{R}^{n\times q}$ empty
for $i = q, \cdots, 1$ **do**
  $X_i \leftarrow x$
  $G_i \leftarrow \nabla f(x)$
  $x \leftarrow x - \alpha H \nabla f(x)$ where α is given by the Wolfe line search.
end for
for $i = 1, \cdots, q$ **do**
  $S_i \leftarrow x - X_i$
  $Y_i \leftarrow \nabla f(x) - G_i$
end for

A large benefit to using this method is that we only update the inverse-Hessian every q steps, thus reducing the computational cost per iteration. However, if q is large then the later steps taken in the $TakeSteps$ algorithm could be based on curvature information far away from the current iterate, which could be detrimental. This leads us on to the next method we will consider.

3.4.2 Rolling-Block-BFGS

This follows the same idea as the Rolling-Block-BFGS method considered by Gao and Goldfarb [11], in which an inverse-Hessian update is performed at every iteration, thus using new information at every step, at extra computational expense of course. We shall explore in Chapter 4 whether this reduces the total iterations required for convergence significantly enough to reduce the overall time to convergence. *TakeSteps* for this method is given by the following algorithm.

Algorithm 6 Rolling-Block-BFGS *TakeSteps*

Input: Iterate x ; inverse-Hessian H ; Old iterates X ; Old gradients G
Output: New iterate x ; New S ; New Y

$X_{>1} \leftarrow X_{<q}$ {Shift old iterates along}
$G_{>1} \leftarrow G_{<q}$ {Shift old gradients along}
$X_1 \leftarrow x$ {Append new iterate}
$G_1 \leftarrow \nabla f(x)$ {Append new gradient}
$x \leftarrow x - \alpha H \nabla f(x)$ where α is given by the Wolfe line search.
for $i = 1, \cdots, q$ **do**
    $S_i \leftarrow x - X_i$
    $Y_i \leftarrow \nabla f(x) - G_i$
end for

3.4.3 Sampled-Block-BFGS

The next method we consider is based on the ideas from Berahas, Jahani and Takáč [1], and it aims to overcome the problem of using old curvature information by sampling the gradient of the objective at q *random* points around the current iterate and interpolating ∇f at these points to obtain a more relevant local model for the objective. Of course, this comes at the added expense of *greatly* increasing the number of gradient evaluations the algorithm must perform. For this reason the update is only performed every q iterations, as in the standard Block-BFGS method, rather than in every iteration as in Rolling-Block-BFGS. This balances the average number of gradient evaluations to two per step; one for sampling and one for taking the step.

Sampled-Block-BFGS may also reduce the effects of ill-conditioning in the inverse-Hessian approximate. In both Block-BFGS and Rolling-Block-BFGS the search directions are used in S. However, often these will be *nearly* colinear as the algorithms will often take steps in similar directions, with this problem becoming particularly detrimental when we use large q. As the columns of S approach linear-dependence, the matrix $Y^T S$ approaches singularity, thus causing ill-conditioning in the inverse-Hessian update. By sampling orthogonal directions randomly to form S, this ceases to be a problem. *TakeSteps* for this method is given by the following algorithm.

Algorithm 7 Sampled-Block-BFGS *TakeSteps*

Input: Iterate x ; inverse-Hessian H
Output: New iterate x ; New S ; New Y
for $i = 1, \cdots, q$ **do**
  $x \leftarrow x - \alpha H \nabla f(x)$ where α is given by the Wolfe line search.
end for
Sample orthonormal directions $\sigma_1, \cdots, \sigma_q$
for $i = 1, \cdots, q$ **do**
  $S_i \leftarrow r\sigma_i$
  $Y_i \leftarrow \nabla f(x + r\sigma_i) - \nabla f(x)$
end for

There remains a question of how to choose the sampling radius r in this algorithm. In our implementation we will choose r to be the average of the previous q step-lengths. In this way the Frobenius norm of S is the same as in the Block-BFGS algorithm.

While this may seem like a reasonable way to overcome colinearity in S, it does seem to go against the guiding principle that was provided by Lemma 2.3.2. This lemma told us that in order to expect superlinear convergence, our inverse-Hessian approximation ought to behave like the true inverse-Hessian along the search directions asymptotically. By sampling randomly we are likely to gain curvature information that is not relevant to the search directions we are using, which could be detrimental to the convergence of the method.

3.4.4 Orthogonalised-Block-BFGS

In order to strike a balance between using sampling to reduce colinearity and using only curvature information that is *relevant* to our optimisation, we consider obtaining the columns of S by orthogonalising the q previous search directions, and sampling for gradients in these directions to obtain Y. We may hope that this method will reduce the ill-conditioning caused by colinearity, while using more relevant curvature information. *TakeSteps* for this method is given by the following algorithm.

Algorithm 8 Orthogonalised-Block-BFGS *TakeSteps*

Input: Iterate x ; inverse-Hessian H
Output: New iterate x ; New S ; New Y

$S \in \mathbb{R}^{n \times q}$ empty
for $i = q, \cdots, 1$ **do**
  $S_i \leftarrow H \nabla f(x)$
  $x \leftarrow x - \alpha H \nabla f(x)$ where α is given by the Wolfe line search.
end for
Orthogonalise and normalise columns of S
$S \leftarrow rS$
for $i = 1, \cdots, q$ **do**
  $Y_i \leftarrow \nabla f(x + S_i) - \nabla f(x)$
end for

One should note that once again a sampling radius r must be selected, and we use the same heuristic as in Sampled-Block-BFGS, that the average norm of the vectors in S should be the same as the average of the last q step lengths.

3.4.5 Orthogonalised-Rolling-Block-BFGS

This method uses the same idea of sampling gradients at orthogonal directions in the span of q previous search directions, but performs the inverse-Hessian update at every step, as in Rolling-Block-BFGS. It is worth noting that this method uses vastly more gradient evaluations, and is unlikely to generalise well to problems where gradient evaluation is particularly expensive. *TakeSteps* for this method is given by the following algorithm.

Algorithm 9 Orthogonalised-Rolling-Block-BFGS *TakeSteps*

Input: Iterate x ; inverse-Hessian H
Output: New iterate x ; New S ; New Y

$S_{>1} \leftarrow S_{<q}$ {Shift old search directions along}
$S_1 \leftarrow H \nabla f(x)$ {Append new search direction}
Orthogonalise and normalise columns of S
$x \leftarrow x - \alpha H \nabla f(x)$ where α is given by the Wolfe line search.
$S \leftarrow rS$
for $i = 1, \cdots, q$ **do**
  $Y_i \leftarrow \nabla f(x + S_i) - \nabla f(x)$
end for

3.5 Analysis of Convergence Properties

We will now turn our attention to the convergence of these methods. Since the BFGS method can be shown to converge superlinearly in a neighbourhood of a local minimum [2], we wish to obtain an equally strong result for Block-BFGS.

This is provided by Schnabel [18], under certain conditions on the choices of S and Y. In the following analysis we use the ℓ_2 condition number

$$K(S) = \|S(S^TS)^{-1}\|_2\|S\|_2$$

and we use the equivalence of the ℓ_2 and Frobenius norms given by

$$\|S\|_2 \leq \|S\|_F \leq \sqrt{r}\|S\|_2$$

where r is the rank of S.

Theorem 3.5.1. *Suppose f is twice continuously differentiable and there exists x^* satisfying $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is symmetric, positive-definite, and $\nabla^2 f$ Lipschitz continuous with Lipschitz constant γ on some open neighbourhood $\mathcal{N}$ containing x^*. Consider sequences $\{x_k\}$ and $\{H_k\}$ generated from $x_0 \in \mathbb{R}^n$ and some symmetric, positive-definite H_0 by the Quasi-Newton update $x_{k+1} = x_k - H_k\nabla f(x_k)$ and the Block-BFGS update (3.1.4) for some choice of sequences $\{S_k\}$ and $\{Y_k\}$ where $S_k, Y_k \in \mathbb{R}^{n\times p_k}$ are such that $Y_k^TS_k$ is symmetric and positive-definite for all k, and each $p_k \in \{1, \cdots, n\}$.*

Suppose there exist $\alpha_1 \geq 0, \alpha_2 \geq 1$ and $q \in \mathbb{Z}_{>0}$ such that for all k we have

$$\left\|Y_k - \nabla^2 f(x^*)S_k\right\|_F \leq \alpha_1\|S_k\|_2\mu_k, \tag{3.5.1}$$

where

$$\mu_k = max\left\{\left\|x^* - \left(x_{k+1} + S_k^{(i)}\right)\right\|_2 : i = 0, \cdots, q_k\right\},$$

where $S_k^{(i)}$ denotes the i^{th} column of S_k for $i > 0$, and is a vector of zeros for $i = 0$, and such that the ℓ_2 condition number of S_k satisfies

$$K(S_k) \leq \alpha_2, \tag{3.5.2}$$

where each $q_k \leq min\{k, q\}$.

Then there exist $\epsilon > 0, \delta > 0$ such that if $\|x_0 - x^*\|_2 < \epsilon$ and $\|H_0 - \nabla^2 f(x^*)\|_F < \delta$, then the sequence $\{x_k\}$ is well defined, and converges q-linearly to x^* with $\{H_k\}$ and $\{H_k^{-1}\}$ uniformly bounded. If in addition we have that for each k there exists v_k such that $S_kv_k = x_{k+1} - x_k$, the rate of convergence is superlinear.

To summarise, in order to guarantee superlinear convergence we need our methods to keep the columns of S *sufficiently* linearly independent, and verify the inequality (3.5.1) for some positive constant α_1 which is independent of k. We will now show that provided we use methods that ensure that requirement (3.5.2) holds, the methods of choosing S and Y and perturbing Y that we have considered satisfy the condition (3.5.1), and we will therefore have linear convergence by Theorem 3.5.1. This will also give us superlinear convergence for any such method that always maintains the latest search direction in the column space of S. We must first note a well-known result, which can be deduced from [16, section 3.2.5].

Lemma 3.5.2. *Suppose f is twice continuously differentiable on a convex set D. Then for any $x, y, z \in D$*

$$\|\nabla f(y) - \nabla f(z) - \nabla^2 f(x)(y-z)\| \leq \sup_{t\in[0,1]} \left\|\nabla^2 f(z + t(y-z)) - \nabla^2 f(x)\right\| \|y - z\|$$

Theorem 3.5.3. *Assume the assumptions on f in Theorem 3.5.1. Let $S = [s^{(1)}, \cdots, s^{(q)}]$, $Y = [y^{(1)}, \cdots, y^{(q)}]$ where $y^{(i)} = \nabla f(x + s^{(i)}) - \nabla f(x)$ and define ΔY by any one of (3.2.4), (3.2.5), or Algorithm 2 with (3.2.7) or (3.2.8). Assume that the method of selecting S ensures that the ℓ_2 condition number of S satisifes $K(S) < \alpha_2$, where α_2 is a uniform bound. Then the requirement (3.5.1) of Theorem 3.5.1 is satisfied by $(Y + \Delta Y)$.*

Proof. We begin by showing that Y satisfies (3.5.1) itself. Let $B_* = \nabla^2 f(x^*)$. Then Lemma 3.5.2 gives us

$$\begin{aligned}
\left\|y^{(i)} - B_* s^{(i)}\right\|_2 &\leq \sup_{t\in[0,1]} \left\|\nabla^2 f(x + ts^{(i)}) - B_*\right\|_F \|s^{(i)}\|_2 \\
&\leq \sup_{t\in[0,1]} \gamma \|x + ts^{(i)} - x^*\|_2 \|s^{(i)}\|_2 \\
&\leq \gamma \max\left\{\|x - x^*\|_2, \|x + s^{(i)} - x^*\|_2\right\} \|s^{(i)}\|_2 \\
&\leq \gamma\mu \|s^{(i)}\|_2
\end{aligned}$$

where the second inequality comes from Lipschitz continuity of $\nabla^2 f$, the third holds because the point on the line connecting x to $x + s^{(i)}$ furthest from x^* must be one of the end points, and the last holds by definition of μ. We can then see

$$\begin{aligned}
\|Y - B_* S\|_F &= \sqrt{\sum_{i=1}^{q} \left\|y^{(i)} - B_* s^{(i)}\right\|_2^2} \\
&\leq \gamma\mu \sqrt{\sum_{i=1}^{q} \|s^{(i)}\|_2^2} \\
&= \gamma\mu \|S\|_F \\
&\leq \sqrt{n}\gamma\mu \|S\|_2 \text{ by equivalence of these matrix norms.} \qquad (3.5.3)
\end{aligned}$$

We therefore need only show that for each choice of ΔY we have

$$\|\Delta Y\|_F \leq c\mu \|S\|_2$$

for some constant c, and then we will attain the result by the triangle inequality.

Case 1

We first consider ΔY given by (3.2.4). This is the example that was proved by

Schnabel [18], and I will repeat it here as it is instructive in the proofs of the other cases. We first note that

$$\begin{aligned}
\|L\|_F &= \frac{1}{\sqrt{2}}\|-L + L^T\|_F \text{ as L is strictly lower triangular} \\
&= \frac{1}{\sqrt{2}}\|Y^T S - S^T Y\|_F \text{ by definition of } L \\
&= \frac{1}{\sqrt{2}}\|(Y^T S - S^T B_* S) - (S^T Y - S^T B_* S)\|_F \\
&\leq \frac{1}{\sqrt{2}}\left(\|Y^T - S^T B_*\|_F \|S\|_F + \|S^T\|_F \|Y - B_* S\|_F\right) \\
&\leq \sqrt{2n}\gamma\mu\|S\|_2^2 \text{ by (3.5.3)}
\end{aligned}$$

Then we can see that

$$\begin{aligned}
\|\Delta Y\|_F &\leq \|S(S^T S)^{-1}\|_F \|L\|_F \\
&\leq \sqrt{2}n\gamma\mu K(S)\|S\|_2 \text{ by definition of } \ell_2 \text{ condition number,} \\
&\leq \sqrt{2}n\gamma\mu\alpha_2\|S\|_2 \text{ as required.}
\end{aligned}$$

Case 2

Consider now ΔY given by (3.2.5). We can easily see

$$\begin{aligned}
\|Y(S^T Y)^{-1}\|_F &= \|(SS^T)^{-1} S(S^T Y)(S^T Y)^{-1}\|_F \\
&= \|(SS^T)^{-1} S\|_F
\end{aligned}$$

so that in the same way as in Case 1 we have

$$\begin{aligned}
\|\Delta Y\|_F &\leq \|Y(S^T Y)^{-1}\|_F \|L\|_F \\
&= \|(SS^T)^{-1} S\|_F \|L\|_F \\
&\leq \sqrt{2}n\gamma\mu\alpha_2\|S\|_F \text{ as before.}
\end{aligned}$$

Case 3

Now we consider ΔY given by Algorithm 2 with δ_j given by (3.2.7). We have

$$\delta_j = S_{<j}(S_{<j}^T S_{<j})^{-1} L_j^T$$

where

$$L_j = S_j^T (Y + \Delta Y)_{<j} - Y_j^T S_{<j}.$$

Using a similar technique as in Case 1 we can see

$$\begin{aligned}
\|Y_{<j}^T S_j - S_{<j}^T Y_j\|_F &\leq \|Y_{<j} - S_{<j} B_*\|_F \|S_j\|_F + \|S_{<j}\|_F \|Y_j - B_* S_j\|_F \\
&\leq 2n\gamma\mu\|S_{<j}\|_F \|S_j\|_F.
\end{aligned}$$

From which we can obtain

$$\begin{aligned}\|L_j^T\|_F &\leq \|Y_{<j}^T S_j - S_{<j}^T Y_j\|_F + \|\Delta Y\|_F \|S_j\|_F \\ &\leq 2n\gamma\mu\|S_{<j}\|_F\|S_j\|_F + \|S_j\|_F\sqrt{\sum_{k<j}\|\delta_k\|_F^2}.\end{aligned}$$

We will now prove by induction that for all k we have

$$\|\delta_k\|_F \leq \gamma\mu n^k q^{k/2}\|S_{<k}\|_F(1+K(S_{<k}))^{k+1}. \tag{3.5.4}$$

First note that $\|\delta_1\|_F = 0$, so (3.5.4) holds for $k = 1$. Now assume that (3.5.4) holds for all $k < j$, so that

$$\begin{aligned}\|L_j^T\|_F \leq& 2n\gamma\mu\|S_{<j}\|_F\|S_j\|_F \\ &+ \|S_j\|_F\sqrt{\sum_{k<j}\gamma^2\mu^2 n^{2k}q^k\|S_{<k}\|_F^2(1+K(S_{<k}))^{2(k+1)}} \\ \leq& 2n\gamma\mu\|S_{<j}\|_F\|S_j\|_F \\ &+ \|S_j\|_F\gamma\mu n^{j-1}q^{\frac{j-1}{2}}\|S_{<j}\|_F(1+K(S_{<j}))^j\sqrt{j-1} \\ \leq& n^{j-1}q^{\frac{j}{2}}\gamma\mu\|S_j\|_F\|S_{<j}\|_F\left(1+(1+K(S_{<j}))^j\right).\end{aligned}$$

Where the second inequality holds since for $k < j$ we have $\|S_{<k}\|_F \leq \|S_{<j}\|_F$ and $K(S_{<k}) \leq K(S_{<j})$, and the last holds as $j \leq q$. Then we can see

$$\begin{aligned}\|\delta_j\|_F &\leq \|S_{<j}(S_{<j}^T S_{<j})^{-1}\|_F\|L_j^T\|_F &(3.5.5)\\ &\leq n^j q^{\frac{j}{2}}\gamma\mu\|S_{<j}\|_F K(S_{<j})\left(1+(1+K(S_{<j}))^j\right) &(3.5.6)\\ &\leq n^j q^{\frac{j}{2}}\gamma\mu\|S_{<j}\|_F(1+K(S_{<j}))^{j+1} \text{ as required.} &(3.5.7)\end{aligned}$$

Thus we have

$$\begin{aligned}\|\Delta Y\|_F &= \sqrt{\sum_{j=1}^{q}\|\delta_j\|_F^2} \\ &\leq \alpha_1\mu\sqrt{\sum_{j=1}^{q}\|S_{<j}\|_F^2} \\ &\leq q\alpha_1\mu\|S\|_F \text{ as required.}\end{aligned}$$

Case 4

The case of ΔY obtained from Algorithm 2 with (3.2.8) follows as an easy consequence of Case 3, using the same tweak that was used to obtain the proof for Case 2 from Case 1.

□

We have now proved that provided we ensure the assumption 3.5.2 holds, the methods Block-BFGS and Rolling-Block-BFGS will converge superlinearly in some neighbourhood of a local minimum since the first column of S in these methods is $x_{k+1} - x_k$, so that the requirement for superlinear convergence is satisfied. This is a consequence of implementing the Wolfe line search, which guarantees that the first column of S will always be accepted by the Modified Cholesky algorithm. No such guarantee exists for the Sampled-Block-BFGS method, however, since the columns of S are sampled randomly. Thus we can guarantee only linear convergence for this method. We do, however, see the promise of the Orthogonalised-Block-BFGS and Orthogonalised-Rolling-Block-BFGS methods as they sample the columns of S from the span of the previous q search directions, so that the requirement (3.5.1) is certainly satisfied. Additionally we need not assume that (3.5.2) holds for these methods, as the ℓ_2 condition number of S is clearly bounded as its columns are orthogonal.

Chapter 4

Numerical Results

4.1 Preliminaries

4.1.1 Implementation

Our optimisation routines are implemented in Python 3.7 with Wolfe line search given by the **line_search** function from SciPy [22] with default parameters $c_1 = 10^{-4}$, and $c_2 = 0.9$, the same values suggested by Nocedal [15] for the BFGS algorithm. This code is adapted to satisfy the Wolfe line search conditions rather than the Strong Wolfe conditions. Where the line search fails to converge, we then try resetting the inverse-Hessian estimate to the identity matrix, and try the Wolfe line search again. If this fails, a back-tracking Armijo line search is performed, and the inverse-Hessian is not updated. The Concurrent.Futures module is utilised for multiprocessing in order to sample gradient values in parallel for the Sampled, Orthogonalised, and Orthogonalised-Rolling Block BFGS methods. A sample of the code used is provided in Appendix C.

4.1.2 Test Problems

Generalised Rosenbrock

We utilise the Generalised Rosenbrock function to provide a source of problems for our optimisers to solve. For some chosen dimension d, the generalised Rosenbrock function is given by

$$R_d(x) = \sum_{i=1}^{d-1} \left(100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right).$$

Such functions have a global minimum at $(1, \cdots, 1) \in \mathbb{R}^d$. We will vary both the dimension and starting point to create sets of problems on which to compare the performance of our optimisers.

CUTEst

We also consider a subset of problems provided by the CUTEst problem set. We use 25 problems of dimension 50, and 18 of dimension 100, totalling 43 problems. These are listed in Appendix A. These problems were chosen by running the BFGS algorithm on the whole set of problems which were available with dimension 50 and 100, and selecting those for which the algorithm converged within 10,000 iterations. The default starting point is used for these problems. We refer to this problem set as the CUTEst problem set.

Neural Network

We will also evaluate the performance of our methods on the task of training a neural network. We train a wide-and-shallow dense network with 1 hidden layer with 150 neurons, each using a sigmoid non-linear activation, for a linear binary classification problem. A 10 dimensional data set of size 100 is generated around the origin, and a separating hyperplane is selected to separate the data for classification. This results in an optimisation problem with dimension 1650. By initialising the network with different random seeds, we create a set of problems on which to compare our optimisers. We use only a small data set to ensure that the sum of squares loss function can be reduced to zero easily and we may observe the asymptotic behaviour of our optimisers. The code used to obtain the function values and gradients of this neural network is provided in Appendix D.

4.1.3 Methodology

In order to compare the performance of a set of optimisers, $\mathcal{S}$, on a set of problems, $\mathcal{P}$, we use the idea of performance profiles [8]. For $s \in \mathcal{S}$ and $p \in \mathcal{P}$, we let $t_{s,p}$ be the cost of optimiser s to converge to a solution on problem p. For the purposes of this paper, the cost will either be the number of iterations required for convergence, or the total CPU time. The algorithm is deemed to have converged when $\|\nabla f(x_k)\| \leq 10^{-5}$. For each problem p we define $m_p = \min_{s \in \mathcal{S}} t_{s,p}$, the minimum cost to solve that problem. For each optimiser s we define

$$\rho_s(r) = \frac{\left|\left\{p \in \mathcal{P} : \frac{t_{s,p}}{m_p} \leq r\right\}\right|}{|\mathcal{P}|},$$

so that $\rho_s(r)$ is the proportion of problems that optimiser s solved within a factor r of the minimum for each problem. Then the performance profile is a plot of these functions for each optimiser, and the quicker a curve rises to 1, the better the corresponding optimiser performed on the problem set $\mathcal{P}$ with respect to the chosen cost.

4.1.4 Sampled-Block-BFGS

First we will see an example that demonstrates the poor performance, which was expected, of the Sampled-Block-BFGS method. In figure 4.1 we see how this method compares to the BFGS method on the classical generalised Rosenbrock problem for dimensions 10, 30 and 50, initialised at $(-1, \cdots, -1)^T$. The curves for the Sampled-Block-BFGS results are averages of the results obtained for each of the four methods of maintaining symmetry of $Y^T S$. Even on such small problems we begin to see poor performance, and this trend continues in all further examples we have tried. We therefore discount it from all further results.

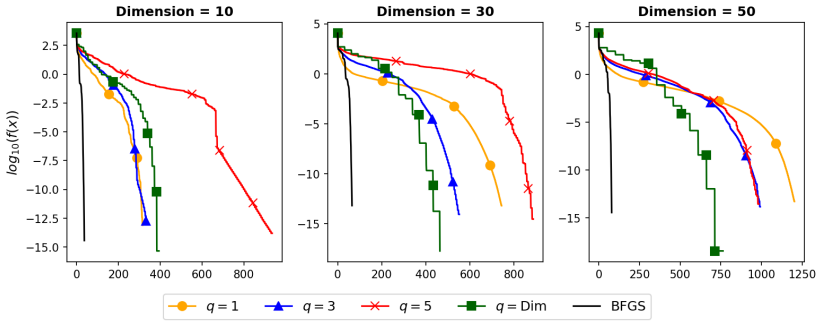

Figure 4.1: **Rosenbrock** Log plots of function value by iteration comparing Sampled-Block-BFGS and BFGS

4.2 Enforcing Symmetry - A Numerical Study

Recall that we gave four methods of enforcing the symmetry of $Y^T S$ in each of our Block BFGS methods. We will first compare the performance of each algorithm separately to explore the differences in performance that these four methods induce. In the following plots, 'Weighted' refers to whether the method comes from using the weighted Frobenius norm or the regular Frobenius norm. 'Prioritised' refers to whether the method used Algorithm 2 to prioritise more recent information.

Figures 4.2 and 4.3 show the results of testing the Block BFGS optimisers for each of the four methods of enforcing symmetry for q-values 3 and 5 on a problem set generated from the Generalised Rosenbrock function, comprised of R_d for $d = 200, 210, \cdots, 600$, initialised at $(-1, \cdots, -1)^T$, totalling 41 problems.

These figures show a clear benefit to using the methods given by Algorithm 2, particularly when using the larger $q = 5$, in which these methods outperformed their counterparts in every example. The difference certainly seems less significant for the Orthogonalised methods, which is not necessarily unexpected since these methods have no ordering in how relevant the columns of S are to the current iteration. With this in mind it is interesting that these methods do

still seem to perform better. We see little difference between using weighted or unweighted norms in the prioritised information methods, however we do see the unweighted norm coming out on top in most cases for the other two methods.

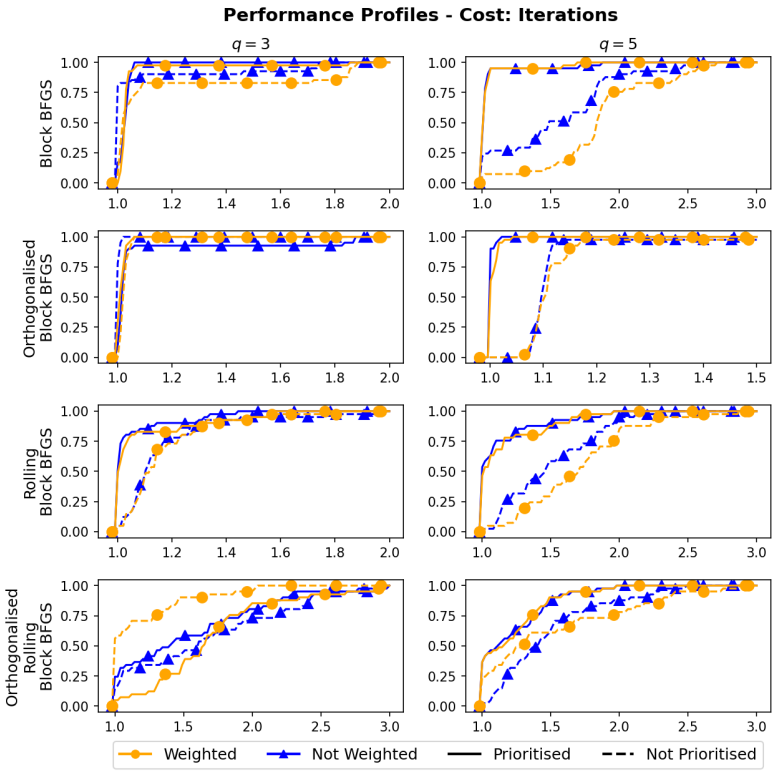

Figure 4.2: **Rosenbrock** Performance profiles comparing methods of enforcing symmetry by *Iterations*.

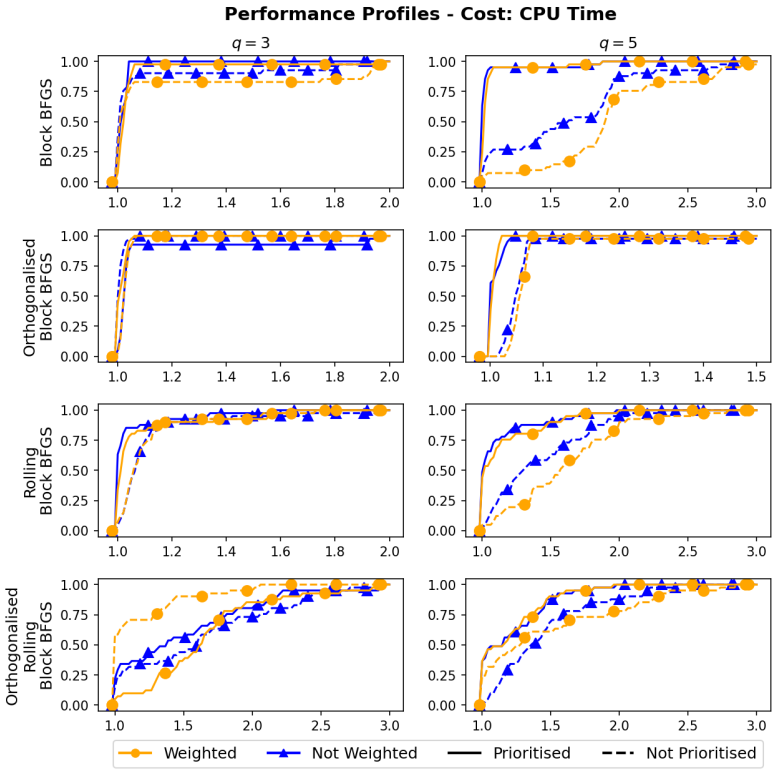

Figure 4.3: **Rosenbrock** Performance profiles comparing methods of enforcing symmetry by *CPU Time*.

Figures 4.4 and 4.5 show the corresponding results from the CUTEst problem set, and they demonstrate the same benefit of using Algorithm 2, with this method coming out on top again across the board in terms of both iterations and CPU time. We once again see a smaller, though still present, difference in the Orthogonalised methods. Here we tested for q-values 2, 3 and 4, and ommit the plots for $q = 3$ for space. They show the same patterns as we have observed above.

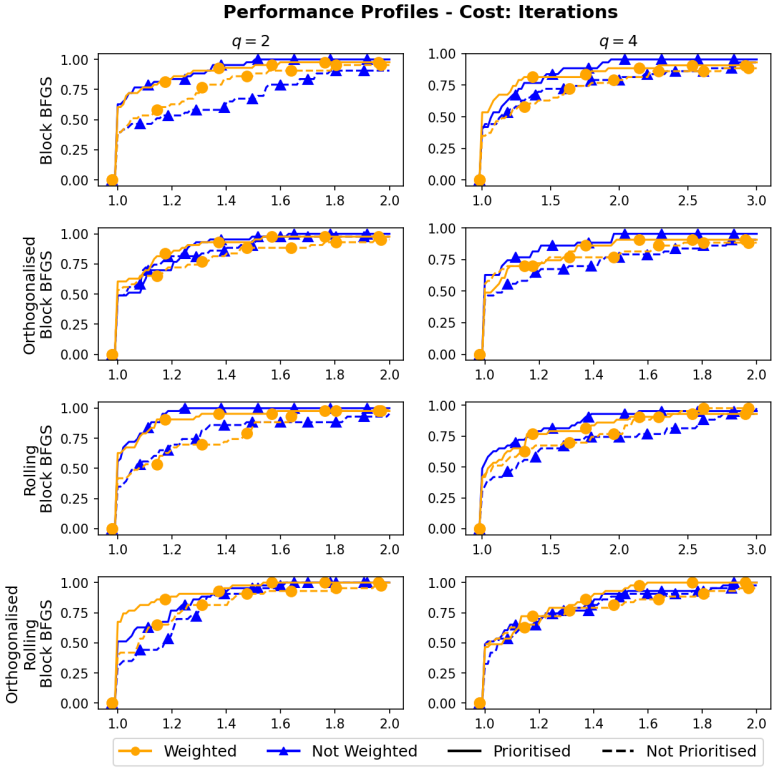

Figure 4.4: **CUTEst** Performance profiles comparing methods of enforcing symmetry by *Iterations*.

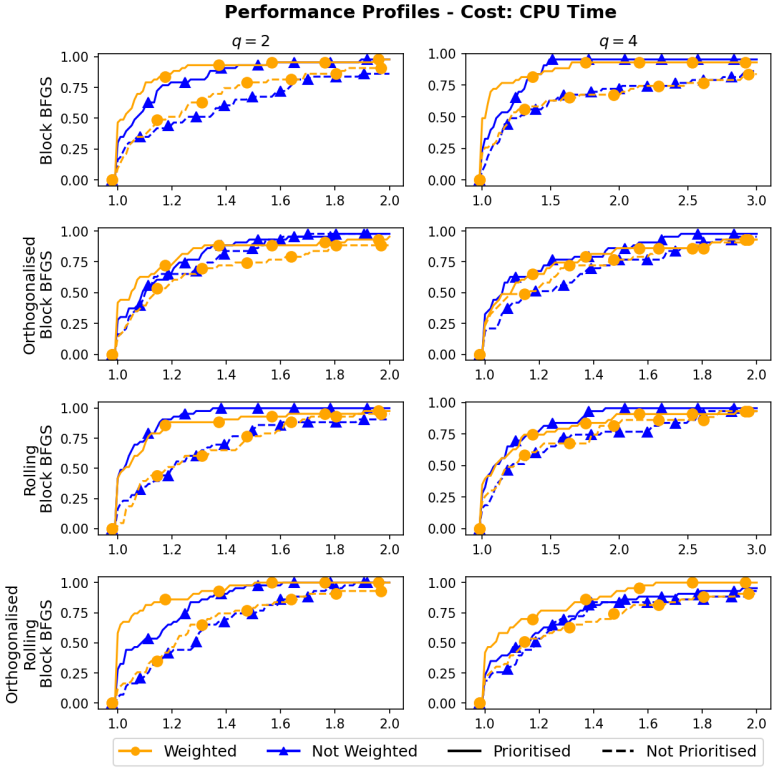

Figure 4.5: **CUTEst** Performance profiles comparing methods of enforcing symmetry by *CPU Time*.

4.3 A Comparison of Block BFGS Methods

For the reasons given previously we now use Algorithm 2 with un-weighted norm as our standard method for maintaining symmetry of $Y^T S$, and we compare the performance of our Block-BFGS variants.

4.3.1 Small Dimension Problems

First we create a set of problems using the Generalised Rosenbrock function with dimensions 100, 200, 300 and 400, running each of our Block BFGS optimisers for q-values 2, 4 and 6 starting from ten randomly generated starting points, centred around the optimum, for each dimension. This results in a set of 40 problems. The larger q-values proved detrimental to the convergence of our methods, with the best results being obtained by $q = 2$ across the board. These results are presented in Figure 4.6. We observe the Rolling-Block-BFGS and Orthogonalised-Rolling-Block-BFGS methods performing similarly to the BFGS method, and Block-BFGS and Orthogonalised-Block-BFGS show poorer

performance. All methods outperform Steepest Descent, which takes over 60 times more CPU time than the minimum to converge in the worst case.

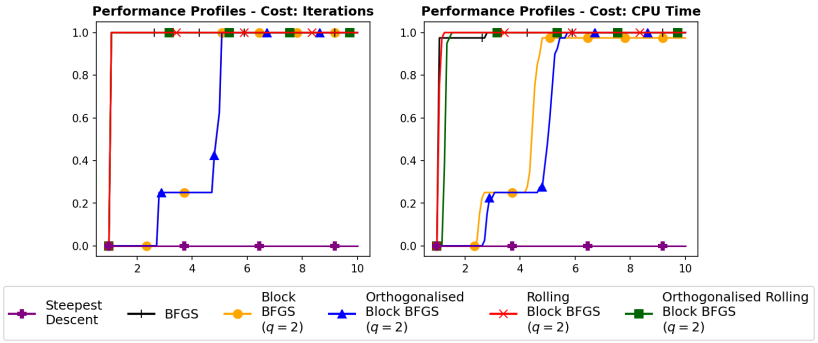

Figure 4.6: **Rosenbrock** Performance profiles comparing optimisers.

Figure 4.7 shows the results of testing the optimisers for q-values 2, 3 and 4 on the CUTEst problem set, and once again the results from the best-performance are presented for each. We can immediately see that the Block BFGS optimisers show much more similar performance on this problem set, and we see Block-BFGS and Orthogonalised-Block-BFGS slightly outperforming both their Rolling counterparts and the BFGS method in terms of CPU time. Once again, all of our methods outperform the Steepest Descent method, which failed to converge within the allowed number of iterations $(10,000)$ on approximately 15% of the problems.

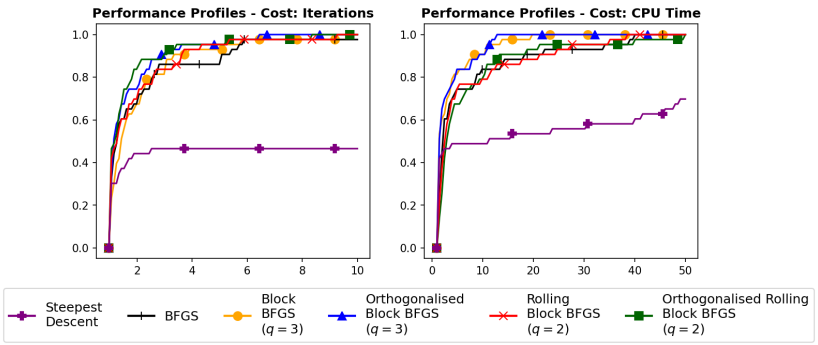

Figure 4.7: **CUTEst** Performance profiles comparing optimisers.

4.3.2 Larger Dimension Problems

We will now evaluate the performance of our methods on some larger dimension problems. First we use the generalised Rosenbrock function with dimension

1000, and create 15 problems by using 15 randomly generated starting points, centered around the point $(-1, \cdots, -1)^T$. We test our methods for q-values 4, 7 and 10, and present the best performing in Figure 4.8. Note that due to the computation time required for convergence of Steepest Descent on these problems, it is not included in the tests for this problem set.

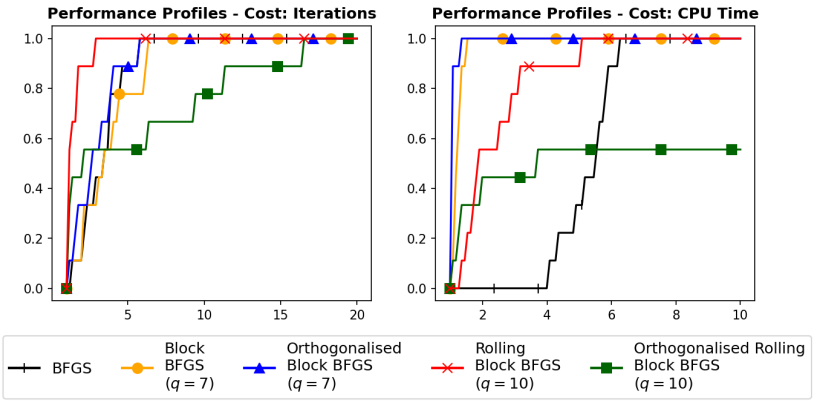

Figure 4.8: **Rosenbrock: Dimension 1000** Performance profiles comparing optimisers.

The first thing we can note from Figure 4.8 is how Rolling-Block-BFGS shows the best performance in terms of iterations, but falls behind in terms of CPU time, as we had anticipated may happen. We can also note the interesting behaviour of Orthogonalised-Rolling-Block-BFGS, which despite performing well on a large proportion ($\approx 40\%$) of the problems, also performs very poorly on another large proportion of problems. We will explore this some more later. We see Block-BFGS, Othogonalised-Block-BFGS and BFGS performing similarly in terms of Iterations, with BFGS falling behind when it comes to CPU time.

Figure 4.9 shows the corresponding results for similar tests we perform using the DQDRTIC problem from the CUTEst problem set. Here we create 20 problems by sampling starting points around the default point for the problem. We test our optimisers for q-values 3, 6 and 9, and present the results from the best performance for each. We once again see Block-BFGS and Orthogonalised-Block-BFGS outperforming the other optimisers similarly in terms of CPU time, despite not performing as well in terms of iterations.

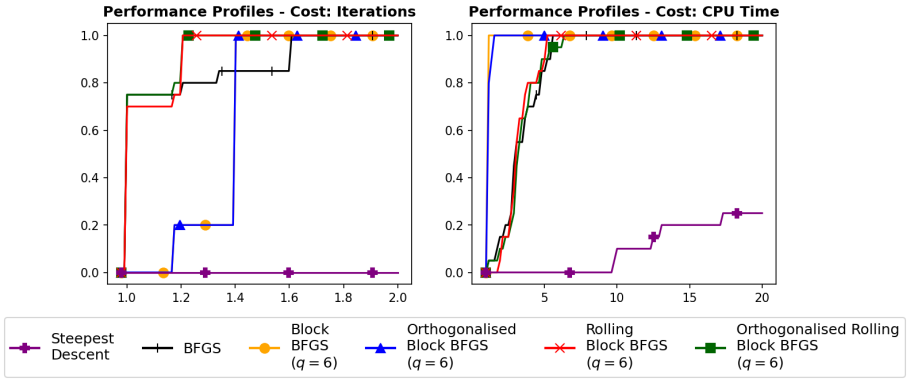

Figure 4.9: **DQDRTIC: Dimension 1000** Performance profiles comparing optimisers.

We now consider a problem set created by initialising the neural network described in Subsection 4.1.2 20 times. We have tested our optimisers for q-values 2, 3 and 4, and presented the best performing optimiser for each in Figure 4.10. During early experimentation large values of q demonstrated poor behaviour for this problem, finding sub-optimal local minima, so we are focusing only on these small q-values. We observe all of our methods performing well compared with Steepest Descent, and we once again see Block-BFGS and Orthogonalised-Block-BFGS outperforming the BFGS method. We also see Orthogonalised-Rolling-Block-BFGS showing extremely varied performance once again.

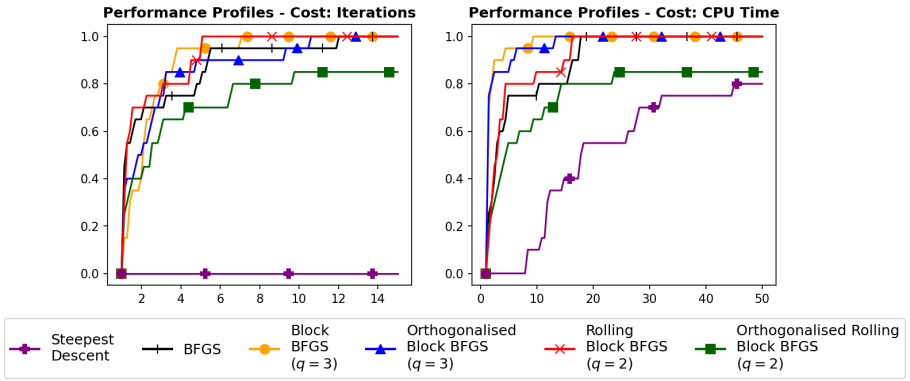

Figure 4.10: **Neural Network: Dimension 1650** Performance profiles comparing optimisers.

We will now take a closer look at the convergence of each of our methods, and look to explain the odd behaviour of Orthogonalised-Rolling-Block-BFGS on the Rosenbrock and Neural Network problem sets. Let us look first at Figure 4.11, comprised of log-plots of the function value by iteration for each problem

and algorithm. We can see a clear difference in convergence between two subsets of problems for Orthogonalised-Rolling-Block-BFGS. Call the subset on which it shows poor performance the red set (the red, dashed curves), and the other subset the blue set (the blue, dotted curves). We can see that Rolling-Block-BFGS shows similar behaviour to its orthogonalised counterpart, albeit to a lesser extent, where it tends to perform worst on problems from the red set. Both Block-BFGS and its orthogonalised counterpart show no difference in behaviour on these sets. Interestingly, BFGS performs better on the red set than the blue. What could cause these differences is not clear, it seems that Rolling-Block-BFGS and Orthogonalised-Rolling-Block-BFGS are both susceptible to getting stuck, and their performance is highly dependent on their initialisation.

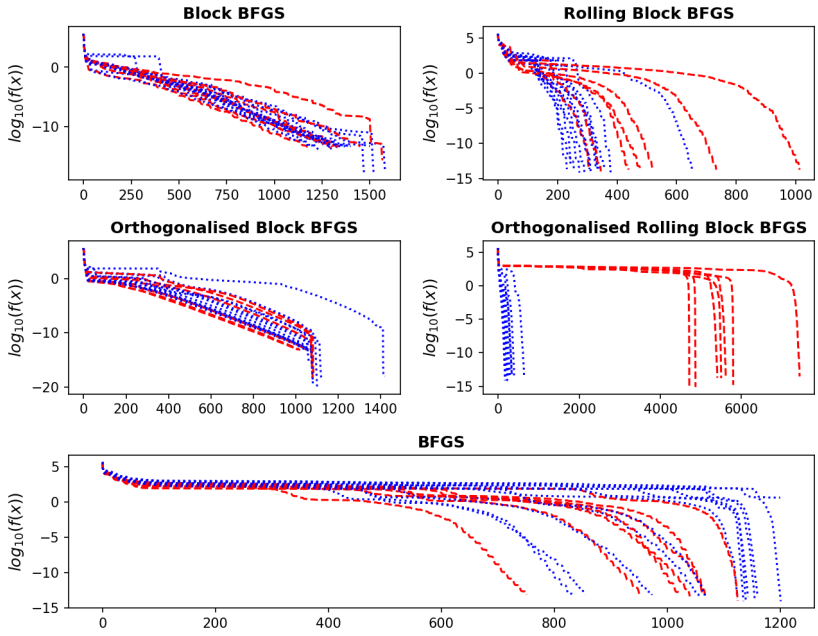

Figure 4.11: **Rosenbrock: Dimension 1000** Log plots of function values by iteration.

We can also see some of the differences between our methods highlighted by how the condition number of the inverse-Hessian estimate evolves throughout the optimisation routine. Figure 4.12 displays this information, though due to the erratic behaviour of the plots, some smoothing was required in order to see the patterns. Each of the plots is smoothed by taking the mean value of the true curve for some window around each point. The radius of this window is given in brackets in the title for each plot. The curves are coloured in the same way as in Figure 4.11.

We first note that the plot for BFGS shows the condition number remains fairly stable throughout optimisation. In contrast we see that Block-BFGS and

Rolling-Block-BFGS both show more erratic behaviour with large peaks. We can also see that the plots for Block-BFGS tend to decrease to a lower value as the algorithm approaches convergence compared to Rolling-Block-BFGS which seems to remain high, especially for problems from the red set. Perhaps the most noticeable difference in these plots, however, is how smooth the plots for Orthogonalised-Block-BFGS are. With very little smoothing we see the curves rising to a value and then settling there with few peaks. The plots for Orthogonalised-Rolling-Block-BFGS, meanwhile, are very erratic, though generally quite low.

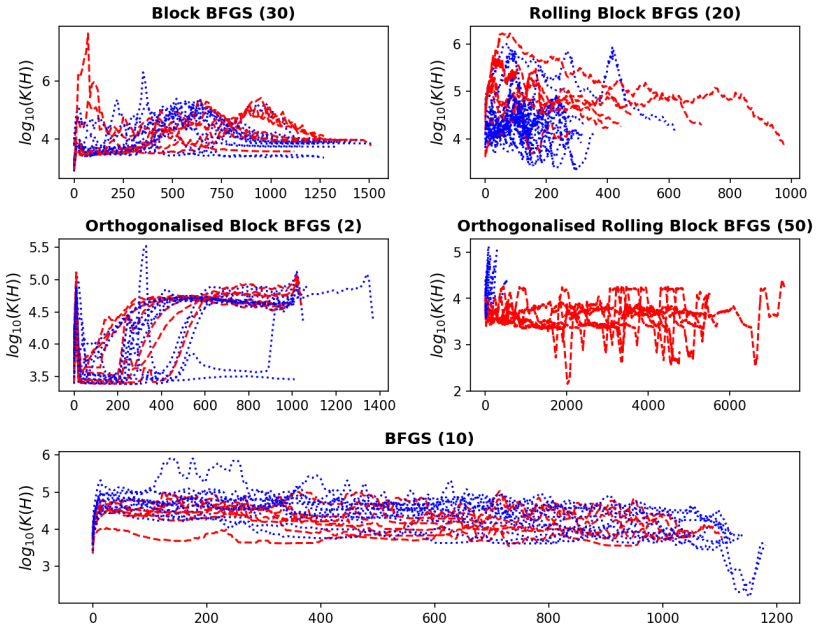

Figure 4.12: **Rosenbrock: Dimension 1000** Log plots of condition number of inverse-Hessian estimate by iteration.

Figure 4.13 shows the corresponding plots of the condition number for the DQDRTIC problem set. We again see that Block-BFGS tends to have larger condition numbers for the inverse-Hessian estimate compared to BFGS. The plots for Block-BFGS show many sharp jumps to extremely high values, so this method is often acting with low accuracy. By comparison the Orthogonalised methods result in much lower condition numbers, though still higher than that of BFGS. This may be a contributing factor to its slightly better performance compared to Block-BFGS in Figure 4.8. The log-plots of function values for tests conducted on the DQDRTIC and Neural Network problem sets are included in Appendix B.

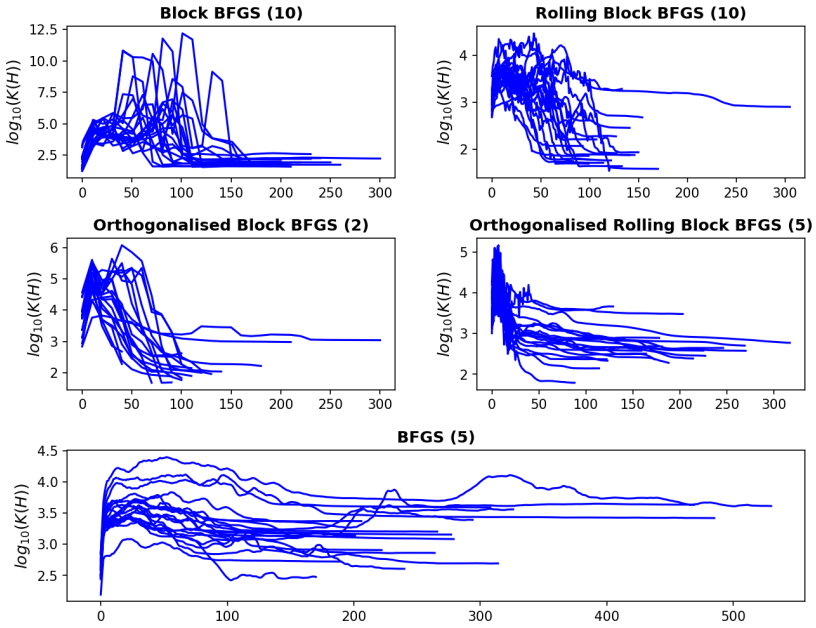

Figure 4.13: **DQDRTIC: Dimension 1000** Log plots of condition number of inverse-Hessian estimate by iteration.

4.4 Overview of Numerical Results

We have observed in Figures 4.7, 4.8 and 4.10 that the Block-BFGS method and its orthogonalised counterpart can perform competetively with the BFGS method in terms of iterations, resulting in these methods outperforming BFGS in terms of the CPU time required to reach convergence. We have observed throughout our experiments that the Rolling-Block-BFGS method performs well in terms of iterations, however the higher computational cost per iteration results in poorer performance than Block-BFGS, as in Figures 4.8, 4.9 and 4.10. We do see this method performing as well as BFGS, however. We have seen Orthogonalised-Rolling-Block-BFGS show unreliability in its performance, sometimes performing well, but often getting stuck, as seen in Figure 4.11.

The motivation behind our orthogonalised methods was to mitigate the effects of ill-conditioning that are introduced in the Block BFGS methods. We have observed in Figures 4.12 and 4.13 that the Orthogonalised-Block-BFGS method does not exhibit the large spikes of ill-conditioning that appear for the Block-BFGS method, demonstrating that this method does indeed show some merit.

Chapter 5

Conclusion

In this work we have discussed in detail six algorithms for unconstrained continuous optimisation: BFGS, Block-BFGS, Rolling-Block-BFGS, Sampled-Block-BFGS, Orthogonalised-Block-BFGS and Orthogonalised-Rolling-Block-BFGS. We have outlined theoretical benefits and disadvantages of each method, and we have implemented them on a variety of test problems in order to explore their differences in practice. We have seen clear evidence of the potential that Block BFGS methods have to outperform the classical BFGS and Steepest Descent methods, and we have demonstrated that in some cases our own orthogonalised methods show good performance, particularly on larger problems.

A more extensive review of the performance of such methods on large scale problems would be required in order to validate the findings of this work. Further examination of the performance of Block BFGS methods on large-scale problems would be highly illuminating, and further research into methods for maintaining symmetry and positive definiteness in our algorithms could lead to further improvements. An in-depth comparison between our Block BFGS methods and the methods of Gao and Goldfarb [11] is another important next step.

Bibliography

[1] Albert S Berahas, Majid Jahani, and Martin Takáč. "Quasi-newton methods for deep learning: Forget the past, just sample". In: *arXiv preprint arXiv:1901.09997* (2019).

[2] C. G. BROYDEN, Jr. DENNIS J. E., and JORGE J. MORÉ. "On the Local and Superlinear Convergence of Quasi-Newton Methods". In: *IMA Journal of Applied Mathematics* 12.3 (Dec. 1973), pp. 223–245.

[3] Charles George Broyden. "The convergence of a class of double-rank minimization algorithms 1. general considerations". In: *IMA Journal of Applied Mathematics* 6.1 (1970), pp. 76–90.

[4] Richard H Byrd et al. "On the use of stochastic hessian information in optimization methods for machine learning". In: *SIAM Journal on Optimization* 21.3 (2011), pp. 977–995.

[5] Louis Caccetta. "Application of optimisation techniques in open pit mining". In: *Handbook of operations research in natural resources*. Springer, 2007, pp. 547–559.

[6] Coralia Cartis. *C6.2 Continuous Optimisation*. 2020.

[7] Amlan Das and Bithin Datta. "Application of optimisation techniques in groundwater quantity and quality management". In: *Sadhana* 26.4 (2001), pp. 293–316.

[8] Elizabeth D Dolan and Jorge J Moré. "Benchmarking optimization software with performance profiles". In: *Mathematical programming* 91.2 (2002), pp. 201–213.

[9] P de Fermat. "Abhandlungen uber Maxima und Minima". In: *book: Oswalds, Klassiker der Exakten Wissenschaften* 238 (1934).

[10] Roger Fletcher. "A new approach to variable metric algorithms". In: *The computer journal* 13.3 (1970), pp. 317–322.

[11] Wenbo Gao and Donald Goldfarb. "Block BFGS methods". In: *SIAM Journal on Optimization* 28.2 (2018), pp. 1205–1231.

[12] Donald Goldfarb. "A family of variable-metric methods derived by variational means". In: *Mathematics of computation* 24.109 (1970), pp. 23–26.

[13] Nicholas Gould. *An introduction to algorithms for continuous optimization*. 2006.

[14] Robert Gower, Donald Goldfarb, and Peter Richtárik. "Stochastic block BFGS: Squeezing more curvature out of data". In: *International Conference on Machine Learning*. 2016, pp. 1869–1878.

[15] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[16] James M Ortega and Werner C Rheinboldt. *Iterative solution of nonlinear equations in several variables*. Vol. 30. Siam, 1970.

[17] KB Petersen, MS Pedersen, et al. "The Matrix Cookbook, vol. 7". In: *Technical University of Denmark* 15 (2008).

[18] Robert B Schnabel. *Quasi-Newton Methods Using Multiple Secant Equations*. Tech. rep. COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE, 1983.

[19] Nicol N Schraudolph, Jin Yu, and Simon Günter. "A stochastic quasi-Newton method for online convex optimization". In: *Artificial intelligence and statistics*. 2007, pp. 436–443.

[20] David F Shanno. "Conditioning of quasi-Newton methods for function minimization". In: *Mathematics of computation* 24.111 (1970), pp. 647–656.

[21] Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. *Optimization for machine learning*. Mit Press, 2012.

[22] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: https://doi.org/10.1038/s41592-019-0686-2.

Appendix A

CUTEst Test Problems

Dimension 50:

ARGLINA	ARGTRIGLS	BROYDN3DLS	BROYDNBDLS
BRYBND	CHNROSNB	CHNRSNBM	DQDRTIC
DQRTIC	ERRINROS	ERRINRSM	HILBERTB
INDEFM	MANCINO	MOREBV	NONDIA
PENALTY1	PENALTY2	POWER	SPARSINE
SPARSQUR	TOINTGSS	TQUARTIC	TRIDIA
VARDIM			

Dimension 100:

ARGLINA	ARGTRIGLS	BROYDNBDLS	BRYBND
DQDRTIC	DQRTIC	INDEFM	MANCINO
MOREBV	NONDIA	PENALTY1	POWER
SPARSINE	SPARSQUR	TOINTGSS	TQUARTIC
TRIDIA	VARDIM		

Appendix B

Further Results

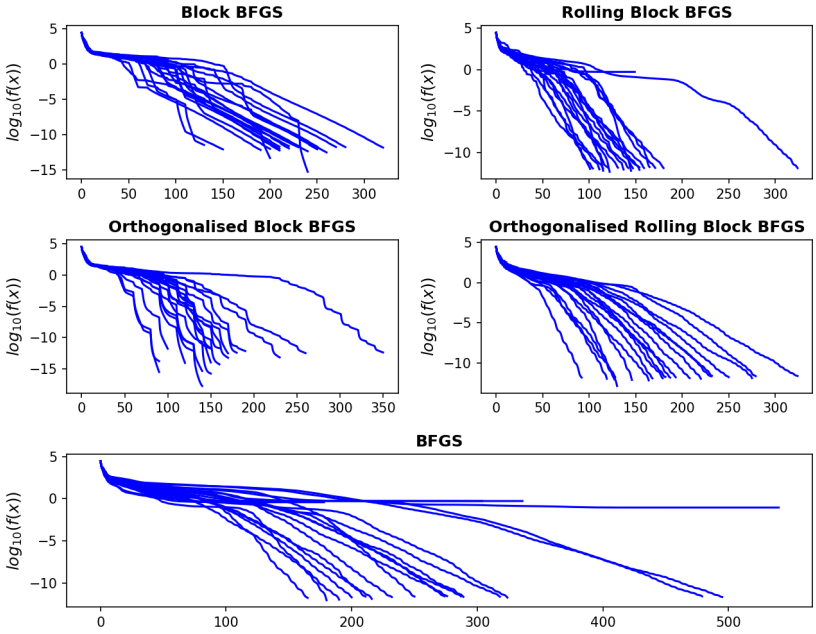

Figure B.1: **DQDRTIC: Dimension 1000** log plots of function value by iteration.

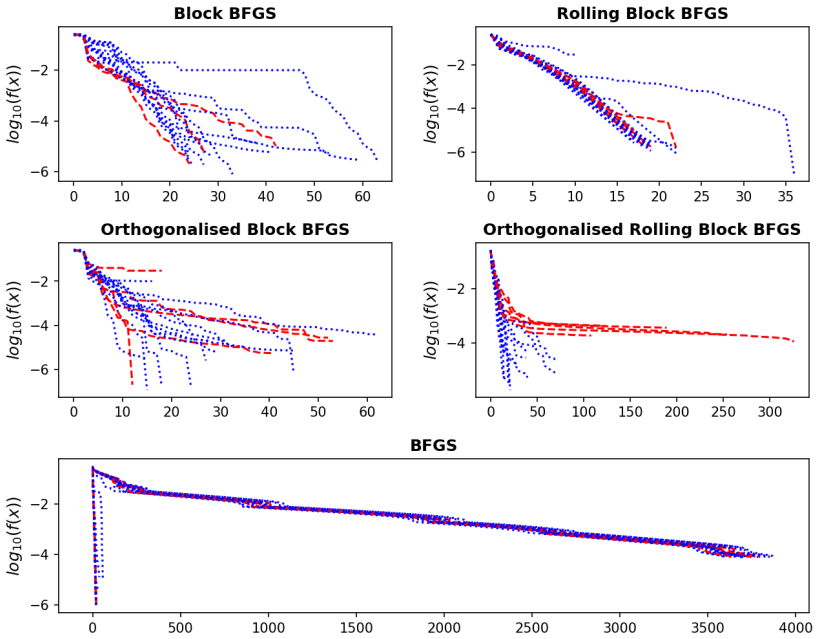

Figure B.2: **Neural Network: Dimension 1650** log plots of function value by iteration.

Appendix C

Optimisers.py

```
import numpy as np
from scipy.optimize import line_search
from numpy.linalg import multi_dot
import time
import scipy
import concurrent.futures


class Block_BFGS_Optimisers():
# Initialising variables
  def __init__(self,
               opt,
               fun,
               grad,
               d,
               q,
               maxiter=1000,
               tol=1e-8,
               which_make_symm=2,
               weighted_make_symm=False,
               conditions=False):
    opts = {'b-bfgs': self.Block_BFGS_TakeSteps,
            'o-b-bfgs': self.Orthogonalised_Block_BFGS_TakeSteps,
            'r-b-bfgs': self.Rolling_Block_BFGS_TakeSteps,
            'o-r-b-bfgs':
            ↪  self.Orthogonalised_Rolling_Block_BFGS_TakeSteps,
            's-b-bfgs': self.Sampled_Block_BFGS_TakeSteps,
            'sd': None}
    if opt == 'sd':
      self.optimiser = self.sd_optimiser
    else:
      self.optimiser = self.bfgs_optimiser
```

```
    self.opt = opt
    self.Take_Steps = opts[opt]
    self.fun = fun
    self.grad = grad
    self.d = int(d)
    self.q = int(q)
    self.tol = tol
    self.weighted_make_symm = weighted_make_symm
    if which_make_symm == 1:
        self.make_symm = self.make_symm1
    if which_make_symm == 2:
        self.make_symm = self.make_symm2
    self.conditions = conditions
    self.maxiter = maxiter
    self.done = False
    self.I = np.eye(d)
    self.grad_norms = list()

# A backtracking Armijo line search to use if Wolfe line search
↪  fails with steepest descent direction
  def bArmijo(self, fun, grad, x, p, alpha, beta, maxiter):
      g = grad(x)
      f = fun(x)
      k = 0
      armijo = False
      while k <= maxiter:
          armijo = fun(x + alpha*p) < f + 1e-4 * alpha *
          ↪  np.dot(np.transpose(p), g)
          if not armijo:
            alpha = alpha * beta
            k += 1
          else:
            return alpha
      return alpha

# Minimum perturbation method of maintaining symmetry
  def make_symm1(self, S, Y):
    X = np.dot(np.transpose(Y), S) - np.dot(np.transpose(S), Y)
    L = -np.tril(X)
    if self.weighted_make_symm:
        LU = scipy.linalg.lu_factor(np.dot(np.transpose(S), Y))
        DY = scipy.linalg.lu_solve(LU, np.transpose(L))
        DY = np.dot(Y, DY)
    else:
        LU = scipy.linalg.lu_factor(np.dot(np.transpose(S), S))
        DY = scipy.linalg.lu_solve(LU, np.transpose(L))
        DY = np.dot(S, DY)
    return DY

```

```
# Prioritising recent information method of maintaining symmetry
  def make_symm2(self, S, Y):
    DY = np.zeros_like(Y, dtype=float)
    for j in range(1, np.shape(Y)[1]):
      Slj = S[:,:j]
      Sj = S[:,j]
      Ylj = (Y + DY)[:,:j]
      Yj = Y[:,j]
      if self.weighted_make_symm:
          X = np.dot(np.transpose(Slj), Ylj)
          L = scipy.linalg.lu_factor(X)
          Z = np.dot(np.transpose(Ylj), Sj) -
          ↪  np.dot(np.transpose(Slj), Yj)
          Lambda = scipy.linalg.lu_solve(L, Z)
          DY[:,j] = np.dot(Ylj, Lambda)/2
      else:
          X = np.dot(np.transpose(Slj), Slj)
          L = scipy.linalg.lu_factor(X)
          Z = np.dot(np.transpose(Ylj), Sj) -
          ↪  np.dot(np.transpose(Slj), Yj)
          Lambda = scipy.linalg.lu_solve(L, Z)
          DY[:,j] = np.dot(Slj, Lambda)/2
    return DY

# Modified Cholesky algorithm
  def cholesky(self, A):
    bad_cols = list()
    L = [[0.0] * len(A) for _ in range(len(A))]
    for i, (Ai, Li) in enumerate(zip(A, L)):
      for j, Lj in enumerate(L[:i+1]):
        s = sum(Li[k] * Lj[k] for k in range(j))
        if Ai[i] <= s:
          Li = np.zeros_like(Li)
          bad_cols.append(i)
          break
        elif i == j:
          Li[j] = np.sqrt(Ai[i] - s)
        elif Lj[j] == 0:
          Li[j] = 0
        else:
          Li[j] = (1.0 / Lj[j] * (Ai[j] - s))
    L = np.delete(L, bad_cols, 0)
    L = np.delete(L, bad_cols, 1)
    return L, bad_cols

# A line search method to use in case Wolfe line search fails
  def line_search(self, x, p, g, H, i):
    alpha, ls_f_evals, ls_g_evals, fval, old_fval, gkp1 =
        line_search(self.fun, self.grad, x, p, g, maxiter=1000)
```

```
126
127     if alpha is None:
128       H = self.I
129       p = -g
130       alpha, ls_f_evals, ls_g_evals, fval, old_fval, gkp1 =
131         line_search(self.fun, self.grad, x, p, g, maxiter=1000)
132
133     if alpha is None:
134       alpha = self.bArmijo(self.fun, self.grad, x, p, 1, 0.1, 100)
135       ls_f_evals += 1
136       ls_g_evals += 1
137       fval = self.fun(x + alpha*p)
138       gkp1 = self.grad(x + alpha*p)
139
140     return alpha, fval, gkp1, H, p
141
142 ########################################################
143
144 # Optimiser method used by all algorithms
145   def bfgs_optimiser(self, x0):
146     x = x0
147     H = self.I
148     i = 1
149     total_steps = 0
150     while i < self.maxiter:
151       try:
152         x, H, steps = self.updater(x, H, iteration=i)
153       except Exception as e:
154         H = self.I
155         print(e)
156       i += 1
157       total_steps += steps
158       if self.done:
159         break
160     if self.done:
161       return x
162     else:
163       print('Failed to converge in maxiter')
164       return x
165
166 # Implements TakeSteps method and then updates the inverse-Hessian
    ↪  estimate
167   def updater(self, x, H, iteration):
168     x, S, Y, steps = self.Take_Steps(x, H, iteration)
169     if S is None:
170       return x, self.I, steps
171     if self.done:
172       return x, None, steps
173     try:
```

```
      DY = self.make_symm(S, Y)
      Y = Y + DY
      L, bad_cols = self.cholesky(np.dot(np.transpose(Y), S))
      S_del = np.delete(S, bad_cols, 1)
      if np.size(S_del) == 0:
        return x, self.I, steps
      Y_del = np.delete((Y+DY), bad_cols, 1)
      L = (L, True)
      DeltaST = scipy.linalg.cho_solve(L, np.transpose(S_del))
      YDeltaST = np.dot(Y_del, DeltaST)
      Z = np.dot(S_del, DeltaST)
      H = Z + multi_dot([self.I-np.transpose(YDeltaST), H,
      ↪  self.I-YDeltaST])
    except Exception as e:
      return x, self.I, steps
    return x, H, steps

# TakeSteps method for the Block-BFGS method, this is the only one
↪  included here for brevity
  def Block_BFGS_TakeSteps(self, x, H, iteration):
    steps = 0
    X = np.zeros((self.d, self.q))
    G = np.zeros((self.d, self.q))
    S = np.zeros((self.d, self.q))
    Y = np.zeros((self.d, self.q))
    i = self.q-1
    g = self.grad(x)
    while i >= 0:
      X[:, i] = x
      if self.done:
        break
      G[:, i] = g
      p = -np.dot(H,g)
      alpha, fval, gp1, H, p = self.line_search(x, p, g, H, i)
      grad_norm = np.linalg.norm(gp1, ord=2)
      self.grad_norms.append(grad_norm)
      self.done = grad_norm < self.tol
      self.fvals.append(fval)
      x = x + alpha*p
      g = gp1
      steps += 1
      i -= 1
    if not self.done:
      for i in range(self.q):
        S[:, i] = x - self.X[:, i]
        Y[:, i] = gp1 - self.G[:, i]
      return x, S, Y, steps
    else:
      return x, None, None, steps
```

Appendix D

Neural Network.py

```
import numpy as np
class NeuralNetwork:
  def __init__(self, x, y, epochs, width, n_hidden, eta):
    self.input = x
    self.y = y
    self.width = width
    self.n_hidden = n_hidden
    input_weights = [np.random.normal(size =
    ↪  (self.input.shape[1],self.width))]
    hidden_weights = [np.random.normal(size = (self.width,
    ↪  self.width)) for i in range(self.n_hidden - 1)] #Need depth
    ↪  - 1 weight matrices between input and output
    output_weights = [np.random.normal(size =
    ↪  (self.width,self.y.shape[1]))]
    self.weights = input_weights + hidden_weights + output_weights
    self.shapes = [np.shape(arr) for arr in self.weights]
    self.output = np.zeros(self.y.shape)
    self.epochs = epochs
    self.eta = eta

  def sigmoid(self, x):
    return 1.0/(1+ np.exp(-x))

  def sigmoid_derivative(self, x):
    return x * (1.0 - x)

  def flatten(self, arrays):
    flat = np.array([])
    for arr in arrays:
      flat = np.append(flat, np.reshape(arr, (-1,)))
    return flat

```

```
29    def un_flatten(self, flat, shapes):
30      arrs = []
31      i=0
32      for n, m in shapes:
33        arrs.append(np.array(flat[i:i+n*m]).reshape(n,m))
34        i += n*m
35      return arrs
36
37    def feedforward(self, weights):
38      self.weights = weights
39      ### This propagates the input through the net to give the
        ↪  output
40      self.layer_values = [self.input] #Get first layer
41      for layer in range(0, self.n_hidden+1):
42        self.layer_values +=
          ↪  [self.sigmoid(np.dot(self.layer_values[-1],
          ↪  self.weights[layer]))] #Iteratively append outputs for
          ↪  each layer
43      self.output = self.layer_values[-1]
44
45    # This is the square loss
46    def loss(self, weights):
47      self.feedforward(weights)
48      output_error = self.y - self.output
49      return np.mean(output_error**2)
50
51    def gradients(self, weights):
52      self.feedforward(weights)
53      ### First get error, delta and adjustment for last layer
54      output_error = self.y - self.layer_values[self.n_hidden + 1]
55      output_delta = output_error *
        ↪  self.sigmoid_derivative(self.layer_values[self.n_hidden +
        ↪  1])
56      output_adjustment = -
        ↪  np.transpose(self.layer_values[self.n_hidden]).
        ↪  dot(output_delta)
57      errors = [output_error]
58      deltas = [output_delta]
59      adjustments = [output_adjustment]
60      ### Then go back through the layers, calculating the
        ↪  adjustments needed for each set of weights
61      for layer in reversed(range(1, self.n_hidden+1)):
62        layer_error =
          ↪  deltas[0].dot(np.transpose(self.weights[layer]))
63        layer_delta = layer_error *
          ↪  self.sigmoid_derivative(self.layer_values[layer])
64        layer_adjustment = - (1/np.shape(self.y)[0])*
          ↪  np.transpose(self.layer_values[layer-1]).dot(layer_delta)
65        errors = [layer_error] + errors
```

```
66        deltas = [layer_delta] + deltas
67        adjustments = [layer_adjustment] + adjustments
68        #print([np.mean(grad) for grad in adjustments])
69      return adjustments
70
71    def predict(self, x):
72      ### A function to predict the labels of new data
73      self.layer_pred = x
74      for weight_matrix in self.weights:
75        self.layer_pred = self.sigmoid(np.dot(self.layer_pred,
          ↪  weight_matrix))
76      self.output_pred = self.layer_pred
```