UNIVERSITY OF CALIFORNIA, SAN DIEGO

Projected-Search Methods for Box-Constrained Optimization

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Mathematics

by

Michael William Ferry

Committee in charge:

Professor Philip E. Gill, Chair
Professor Randolph E. Bank
Professor Thomas R. Bewley
Professor Robert R. Bitmead
Professor Michael J. Holst

2011

The dissertation of Michael William Ferry is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____
Chair

University of California, San Diego

2011

DEDICATION

To Lisa

EPIGRAPH

An inconvenience is only an adventure wrongly considered;
an adventure is an inconvenience rightly considered.
G. K. Chesterton

TABLE OF CONTENTS

Signature Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures viii

List of Tables ix

Acknowledgements x

Vita and Publications xi

Abstract of the Dissertation xii

Chapter 1 Introduction 1
1.1 Overview 1
1.2 Contributions of this thesis 5

Chapter 2 Line-Search Methods for Unconstrained Optimization 8
2.1 Newton's method 8
2.2 Quasi-Newton methods 9
2.2.1 Solving the quasi-Newton equation 11
2.2.2 Limited-memory variants 12
2.3 Line searches 12
2.3.1 Armijo line search 13
2.3.2 Wolfe line search 15
2.4 Reduced-Hessian methods 20
2.4.1 A quasi-Newton implementation 23
2.4.2 Reinitialization 26
2.4.3 Lingering 27
2.4.4 Limited-memory variants 28
2.4.5 Other reduced-Hessian methods 32

Chapter 3 Active-Set Methods for Box-Constrained Optimization 35
3.1 Definitions 36
3.2 Gradient-projection methods 39
3.2.1 Algorithm L-BFGS-B 42
3.3 Projected-search methods 43

3.3.1 Solving the quasi-Newton equation 46
3.3.2 Line searches for projected-search methods 50

Chapter 4 Line Searches on Piecewise-Differentiable Functions 51
4.1 Differentiable functions: the Wolfe step 52
4.2 Piecewise-differentiable functions: the quasi-Wolfe step . 54
4.2.1 Convergence results 60
4.3 Practical considerations 62
4.3.1 Current implementation 64
4.4 A variant of Algorithm L-BFGS-B 65

Chapter 5 Reduced-Hessian Methods for Box-Constrained Optimization . 66
5.1 Converting to a projected-search method 69
5.2 Simple implementation of a projected-search RH algorithm 72
5.3 Updating the working set 75
5.3.1 Removing a constraint from the working set . . . 76
5.3.2 Adding a constraint to the working set 84
5.3.3 Dealing with rank reduction 92
5.4 RH-B algorithms 95
5.5 Convergence results 97
5.6 Future work 97

Chapter 6 Numerical Results 99
6.1 Test problem selection 101
6.2 Explanation of Results 106
6.2.1 Performance Profiling 107
6.3 Numerical Results 108
6.3.1 Reinitialization 108
6.3.2 RH-B methods in MATLAB 113
6.3.3 Known bugs and issues 119
6.3.4 Competitive algorithms 124

Bibliography 132

LIST OF FIGURES

Figure 2.1: Armijo condition 14
Figure 2.2: Weak-Wolfe conditions 16
Figure 2.3: Strong-Wolfe conditions 17

Figure 4.1: No guarantee to update approximate Hessian 61

Figure 6.1: Performance profile for RH-B methods in MATLAB (time) . . . 117
Figure 6.2: Performance profile for RH-B methods in MATLAB (nfg) 118
Figure 6.3: Performance profile for implicit implementations of LRHB (nfg) 123
Figure 6.4: Performance profile for competitive solvers (time) 128
Figure 6.5: Performance profile for competitive solvers (nfg) 129
Figure 6.6: Performance profile for competitive solvers on full set (time) . . 130
Figure 6.7: Performance profile for competitive solvers on full set (nfg) . . . 131

LIST OF TABLES

Table 6.1: RH-B algorithms . . . 101
Table 6.2: Full test set (problems 1–37) . . . 103
Table 6.3: Full test set (problems 38–74) . . . 104
Table 6.4: Full test set (problems 75–111) . . . 105
Table 6.5: Effects of reinitialization (Problems 1–37) . . . 110
Table 6.6: Effects of reinitialization (Problems 38–74) . . . 111
Table 6.7: Effects of reinitialization (Problems 75–111) . . . 112
Table 6.8: RH-B methods in MATLAB (Sum Total) . . . 113
Table 6.9: RH-B methods in MATLAB (Problems 1–37) . . . 114
Table 6.10: RH-B methods in MATLAB (Problems 38–74) . . . 115
Table 6.11: RH-B methods in MATLAB (Problems 75–111) . . . 116
Table 6.12: Implicit implementations of LRHB (Problems 1–37) . . . 120
Table 6.13: Implicit implementations of LRHB (Problems 38–74) . . . 121
Table 6.14: Implicit implementations of LRHB (Problems 75–111) . . . 122
Table 6.15: Comparison of competitive algorithms (Sum Total) . . . 124
Table 6.16: Comparison of competitive algorithms (Problems 1–37) . . . 125
Table 6.17: Comparison of competitive algorithms (Problems 38–74) . . . 126
Table 6.18: Comparison of competitive algorithms (Problems 75–111) . . . 127

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my wife Lisa for her unwavering support and dedication throughout this whole process. Though they may not see this for a long time, I also want to thank my two sons, William and Thomas, for never failing to put a smile on my face at the end of a long day.

I owe a debt of gratitude to my advisor, Philip Gill, for pointing me in the right direction with my research, for supporting me as a Research Assistant for several quarters, and for helping me to always catch the errant split infinitive.

I wish to express my thanks, too, to the rest of my committee: Randy Bank, Mike Holst, Tom Bewley, and Bob Bitmead. Thank you for your time and your helpful comments and advice. A special thanks goes to Tom Bewley, David Zhang, Joe Cessna, Chris Colburn, Robert Krohn, and Paul Belitz in the Engineering department for the opportunity to collaborate together on a number of projects.

Last but not least, I wish to thank my mom and dad. Simply put, I would not be here today without their help.

VITA

2003	B. A., Mathematics and Philosophy, *Summa cum Laude*. University of San Francisco, San Francisco
2003-2005	Middle- and High-School Mathematics Teacher. Saint Monica Academy, Pasadena
2005-2010	Teaching Assistant. Department of Mathematics, University of California, San Diego
2007	M. A., Mathematics. University of California, San Diego
2009	C. Phil., Mathematics. University of California, San Diego
2009-2010	Associate Instructor. Department of Mathematics, University of California, San Diego
2011	Ph. D., Mathematics, University of California, San Diego

ABSTRACT OF THE DISSERTATION

Projected-Search Methods for Box-Constrained Optimization

by

Michael William Ferry

Doctor of Philosophy in Mathematics

University of California, San Diego, 2011

Professor Philip E. Gill, Chair

Many algorithms used in unconstrained minimization are line-search methods. Given an initial point x and function $f : \mathbb{R}^n \to \mathbb{R}$ to be minimized, a line-search method repeatedly solves two subproblems: the first calculates a search direction p; the second performs a line search on the function $\phi(\alpha) = f(x + \alpha p)$. Then, αp is added to x and the process is repeated until a solution is located.

Quasi-Newton methods are often used to calculate the search direction. A quasi-Newton method creates a quadratic model of f at x and defines the search direction p such that $x + p$ is the minimizer of the model. After each iteration the model is updated to more closely resemble f near x.

Line searches seek to satisfy conditions that ensure the convergence of the sequence of iterates. One step that decreases f "sufficiently" is called an Armijo

step. A Wolfe step satisfies stronger conditions that impose bounds on $\phi'(\alpha)$. Quasi-Newton methods perform significantly better when using Wolfe steps.

Recently Gill and Leonard proposed the reduced Hessian (RH) method, which is a new quasi-Newton method for unconstrained optimization. This method exploits key structures in the quadratic model so that the dimension of the search space is reduced.

Placing box constraints x leads to more complex problems. One method for solving such problems is the projected-search method. This method performs an unconstrained minimization on a changing subset of the variables and projects points that violate the constraints back into the feasible region while performing the line search. To date, projected line-search methods have been restricted to using an Armijo-like line search.

By modifying the line-search conditions, we create a new projected line search that uses a Wolfe-like step. This line search retains many of the benefits of a Wolfe line search for the unconstrained case.

Projected-search methods and RH methods share a similar structure in solving for the search direction. We exploit this similarity and merge the two ideas to create a class of RH methods for box-constrained optimization. When combined with the new line search, this new family of algorithms minimizes problems in less than 74% of the time taken by the leading comparable alternative on a collection of standard test problems.

Chapter 1

Introduction

1.1 Overview

An unconstrained minimization problem may be written in the form,

$$\min_{x \in \mathbb{R}^n} f(x),$$

where $f : \mathbb{R}^n \to \mathbb{R}$. Because maximization can be achieved by minimizing $-f$, only minimization methods need to be considered. All unconstrained optimization methods discussed in this dissertation are *line-search methods* unless otherwise noted. Given an initial point x and a continuously-differentiable function f to be minimized, a line-search method repeatedly solves two subproblems: the first calculates a search direction $p \in \mathbb{R}^n$; the second performs a line search on the univariate function $\phi(\alpha) = f(x + \alpha p)$ to compute a step length α. Once α and p have been found, αp is added to x and the process is repeated until a solution is located.

In this dissertation, we also consider box-constrained minimization problems, which can be expressed in the form,

$$\min_{x \in \mathbb{F}} f(x),$$

with $\mathbb{F} = \{x \in \mathbb{R}^n : l \leq x \leq u\}$, where l and u are the constant lower- and upper-bounds for the problem, and $\leq$ is defined component-wise. A point x is called *feasible* if $x \in \mathbb{F}$. Most box-constrained optimization methods proposed in this

dissertation (and all methods proposed in Chapter 5) are *projected-search* methods. They differ from line-search methods primarily in how the second subproblem is posed: instead of performing a line search on the differentiable function $\phi(\alpha)$, they perform a line search on the piecewise-differentiable function $\psi(\alpha) = f(P(x+\alpha p))$, where $P(x)$ is defined to be the closest feasible point to x. A line search of this type may be referred to as a projected line search.

Solving a box-constrained minimization problem can be thought of as solving two subproblems. The first subproblem seeks to identify the optimal active set. Once the optimal active set is identified, the second subproblem seeks to find the unconstrained minimizer of f on the set of "free" variables whose indices are not in the active set. In practice, it is not possible to know when the optimal active set is obtained, but many theoretical convergence results show that the active set is identified after a finite number of iterations. Thus, once the active set is identified, the asymptotic convergence rate of the problem is determined by the unconstrained method chosen in the second subproblem. For practical reasons, projected-search methods do not work with the active set. Instead, they work with a *working set*, which is a subset of the active set that approximates it. In most convergence results, once the active set is identified, the working set is identical to the active set.

If $\lim_{k\to\infty} x_k = x^*$, then the sequence $\{x_k\}$ is said to converge to x^* with *Q-order at least p* if there exists $\mu \geq 0$ and $N \geq 0$ so that, for all $k \geq N$,

$$\|x_{k+1} - x^*\| \leq \mu \|x_k - x^*\|^p,$$

with $p \geq 1$. For the cases $p = 1$ and $p = 2$, the rate of convergence is said to be at least *Q-linear* and *Q-quadratic*, respectively.

The sequence $\{x_k\}$ is said to converge to x^* with *Q-superorder at least p* if, for all $\mu > 0$, there exists $N \geq 0$ so that, for all $k \geq N$,

$$\|x_{k+1} - x^*\| \leq \mu \|x_k - x^*\|^p,$$

with $p \geq 1$. When $p = 1$, the rate of convergence is said to be at least *Q-superlinear*. When $p = 2$, the rate of convergence is said to be at least *Q-quadratic*.

All methods discussed in this dissertation are *model-based* methods. That is, given x, they compute a search direction p so that $x+p$ minimizes some quadratic model of f at x. All functions are assumed to be twice-differentiable unless otherwise noted. With the exception of Newton's method, no methods considered here use $\nabla^2 f$, the second derivative of f, to compute p.

One important model-based method is the quasi-Newton method. During iteration k, a quasi-Newton method calculates p_k so that $x_k + p_k$ minimizes the quadratic model

$$q_k(x) = f_k + g_k^T(x - x_k) + \frac{1}{2}(x - x_k)^T H_k(x - x_k),$$

where $f_k = f(x_k)$, $g_k = g(x_k) = \nabla f(x_k)$, and H_k (the *approximate Hessian*) is a positive-definite, symmetric matrix that, in some fashion, approximates $\nabla^2 f(x_k)$. The search direction p_k, called a *quasi-Newton direction*, can be obtained by solving

$$H_k p_k = -g_k.$$

The approximate Hessian H_k can be defined in many ways. One of the most widely-used definitions computes H_{k+1} from H_k and is called the BFGS update. Given H_k, the BFGS update defines H_{k+1} as:

$$H_{k+1} = H_k + \frac{1}{\gamma_k^T \delta_k}\gamma_k \gamma_k^T - \frac{1}{\delta_k^T H_k \delta_k}(H_k\delta_k)(H_k\delta_k)^T,$$

where

$$\gamma_k = g_{k+1} - g_k \;\text{ and }\; \delta_k = x_{k+1} - x_k.$$

A quasi-Newton method that uses the BFGS update is often referred to as a BFGS method. Throughout this dissertation, all BFGS methods are defined so that $H_0 = \sigma I$, where $\sigma > 0$ and I is the identity matrix of order n. One of the most important characteristics of the BFGS update is that the following equation holds:

$$\delta_k^T H_{k+1} \delta_k = \gamma_k^T \delta_k.$$

In other words, the approximate curvature, $\gamma_k^T \delta_k$, gets *installed* as the actual curvature in the new quadratic model q_{k+1}. Because of this, if $\gamma_k^T \delta_k \leq 0$, then H_{k+1}

is not positive definite. To ensure that H_k always remains positive definite, the BFGS update is applied only when the approximate curvature is positive.

In Section 2.4, we describe a relatively new quasi-Newton method, called a *reduced-Hessian* (RH) method, first proposed by Gill and Leonard [GL01]. By taking advantage of an implicit structure contained in quasi-Newton methods, an RH method is able to calculate the search direction from a much smaller search space. It is possible to implement several techniques that can make the method more efficient. The first technique, called *lingering*, is used to force the search direction p to be taken from a smaller subspace. This allows p to be obtained from a smaller system of equations. The second technique, called *reinitialization*, allows the method to change the curvature of the model q_k in a certain subspace based on improved estimates each iteration.

Additionally, RH methods can be implemented using a limited-memory framework, which stores information about only the most recent m steps. When expressed as a limited-memory algorithm, an RH method can be implemented as an *implicit method*, which stores and updates one of the relevant matrix factors implicitly, or as an *explicit method*, which stores and updates the matrix explicitly. An implicit method requires fewer computations per iteration but is only effective when $m \lesssim 6$. These techniques are described more fully in Sections 2.4.2–2.4.4.

The two most commonly-used line searches used in an unconstrained line-search method are the Armijo and the Wolfe line searches. An Armijo line search seeks to ensure that the next iterate reduces the function sufficiently relative to the directional derivative of f at x_k in direction p_k. A step that satisfies this condition is called an *Armijo step*. A Wolfe line search seeks to enforce conditions on f' as well as the Armijo condition, in order to guarantee that the BFGS update can be safely applied. Such steps are called *Wolfe steps*. Wolfe line searches are appropriate only for differentiable functions. Thus, all projected-search methods to date use an Armijo (or Armijo-like) line search.

In Chapter 3, we review the history of several types methods for box-constrained optimization. One of the most successful algorithms for such problems, first proposed by Byrd, Lu, Nocedal and Zhu [BLNZ95], is Algorithm L-BFGS-B,

which incorporates several strategies used in unconstrained and box-constrained optimization. One of the strengths of Algorithm L-BFGS-B is that it uses a Wolfe line search, which makes it more efficient than comparable algorithms in practice.

1.2 Contributions of this thesis

Because projected-search methods perform a projected line search along the piecewise-differentiable univariate function $\psi(\alpha) = P(x+\alpha p)$, it is not appropriate to employ a Wolfe line search. In Chapter 4, we introduce a new line search, called a quasi-Wolfe line search, that can be used in a projected-search method.

A quasi-Wolfe line search behaves almost identically to a Wolfe line search, except that a step is deemed acceptable (called a *quasi-Wolfe step*) under a wider range of conditions. Relaxed conditions are defined that take into consideration steps where the function is not differentiable.

We provide the theory and some of the corresponding proofs that drive a Wolfe line search. Next, we prove similar results for the quasi-Wolfe line search to show that it is a well-defined algorithm. After identifying the practical considerations needed for converting a Wolfe line search into a quasi-Wolfe line search, we describe details of the implementation.

A quasi-Wolfe line search does not retain all the theoretical benefits of a Wolfe line search. If a quasi-Newton line-search method for unconstrained optimization is used and, during some iteration, a Wolfe step is taken, it can be shown that the update to the approximate Hessian is positive definite (assuming the initial approximate Hessian is positive definite). On the other hand, if a quasi-Newton projected-search method for box-constrained optimization is used and, during some iteration, a quasi-Wolfe step is taken, the update to the approximate Hessian can no longer be guaranteed to be positive definite. In practice, the update is rarely skipped. When an algorithm using a quasi-Wolfe line search was tested on a large number of problems, the update to the approximate Hessian had to be skipped in less than half of one percent of the cases when a quasi-Wolfe step was found.

If a quasi-Wolfe step is found, the undesirable behavior described in the preceding paragraph is only possible if the piecewise-linear path $x(\alpha) = P(x + \alpha p)$ has changed direction for some α between zero and the quasi-Wolfe step. If it can be shown that a projected-search method identifies the active set after a finite number of iterations, then this behavior does not affect the long-term convergence rate in a negative way. Further, a quasi-Wolfe line search behaves like a Wolfe line search once the active set stabilizes.

In addition, we propose a modification of Algorithm L-BFGS-B, called Algorithm LBFGSB-M (for modified), that incorporates a quasi-Wolfe line search. Because we use the underlying code from L-BFGS-B to create the quasi-Wolfe line search, it allows us to compare it directly to L-BFGS-B to test the effectiveness of the new line search.

In Chapter 5, we propose a new class of projected-search methods, called RH-B methods, that generalize RH methods to box-constrained optimization. RH-B methods take advantage of structure common to both projected-search and reduced-Hessian methods to create a new way to solve for a search direction. For efficiency, most of the matrix factorizations used by RH-B method are updated when the working set is changed. A large amount of Chapter 5 is devoted to the linear algebra that describes these updates. We also describe the considerations to be made when converting a reduced-Hessian method into a projected-search method.

Due to the way factors in RH-B methods are updated, almost all RH-B methods are built on a limited-memory framework. Because of this, these RH-B methods can be implemented as implicit or explicit methods. All RH-B methods can implement lingering and reinitialization. The most effective RH-B method, as measured by the numerical results in Chapter 6, is Algorithm LRHB. Algorithm LRHB is a limited-memory RH-B method that does not implement lingering and implements reinitialization if the dimension of the problem is large enough.

Based on work done by Bertsekas [Ber82], it can be shown that under suitable circumstances and with small modifications, RH-B methods identify the active set in a finite number of iterations and, consequently, converge at a Q-

superlinear rate.

In addition to new limited-memory-based methods, we propose a simplified RH-B algorithm, Algorithm RHSB, which is a projected-search method built on a full-memory framework. The main feature of Algorithm RHSB is that it restarts each time the working set changes, to avoid having to update certain matrix factorizations. Although inefficient in many ways, if the active set is identified in a finite number of iterations, it shares the same asymptotic convergence rates as Algorithm LRHB.

All RH-B methods are implemented in MATLAB, and all but RHSB are implemented in Fortran 90. Algorithms L-BFGS-B and LBFGSB-M are implemented in Fortran 77 and called with C++ wrappers. Finally, all testing is done in MATLAB. Any algorithms that are not implemented directly as MATLAB m-files are called using a MATLAB mex interface.

In Chapter 6, we identify a large set of box-constrained optimization test problems. Through our numerical results, we support the view that reinitialization is beneficial on problems with many variables but often detrimental on problems with few variables. We also compared several RH-B implementations and explain the strengths and weaknesses of each method. Finally, we tested the Fortran implementation of LRHB against L-BFGS-B and LBFGSB-M. Based on the numerical results, we show that LBFGSB-M outperformed L-BFGS-B and that LRHB significantly outperformed LBFGSB-M (see Table 6.1 on page 101 for a list of RH-B algorithms and their descriptions). Comparing the time taken to solve each test problem and summing over all the problems for which all three algorithms converged, Algorithm LRHB took less than 74% of the time that Algorithm L-BFGS-B took.

Chapter 2

Line-Search Methods for Unconstrained Optimization

All methods discussed in Chapter 2 work by zero-finding on $g(x)$. That is, they seek x^* so that

$$g(x^*) = 0.$$

Points that satisfy this condition are called *critical points* or *stationary points*. If f is convex, then x^* is a global minimizer of f. If $\nabla^2 f(x^*)$ is positive definite, then x^* is an isolated local minimizer[1] of f. If x^* is a local minimizer of f and $\nabla^2 f$ exists at x^*, then $\nabla^2 f(x^*)$ must be positive semi-definite. With the exception of Newton's method, none of the methods discussed in this chapter make use of second-derivative information about f. Consequently, it is not possible to identify minimizers of f with these methods. Instead, such methods can only identify stationary points of f.

2.1 Newton's method

Newton's method, also known the Newton-Raphson method, is a model-based method that, given a point x_k and a twice-differentiable function f with $\nabla^2 f(x_k)$ positive definite, computes a search direction p_k so that $x_k + p_k$ is the

[1] A point x^* is an *isolated local minimizer* if there is a neighborhood $\mathcal{N}$ of x^* such that x^* is the only local minimizer in $\mathcal{N}$.

unique minimizer of the quadratic model

$$q_k^N(x) = f_k + g_k^T(x - x_k) + \frac{1}{2}(x - x_k)^T \nabla^2 f(x_k)(x - x_k).$$

Since $\nabla^2 f(x_k)$ is positive definite, p_k must satisfy the system

$$\nabla^2 f(x_k) p_k = -g_k, \tag{2.1}$$

since

$$p_k = \operatorname*{argmin}_p q_k^N(x_k + p) \iff \nabla q_k^N(x_k + p_k) = 0 \iff g_k + \nabla^2 f(x_k) p_k = 0.$$

If $\nabla^2 f(x_k)$ is not positive definite but equation (2.1) is still solvable, $x_k + p_k$ is merely a stationary point of $q_k^N(x)$. A search direction p_k obtained from equation (2.1) is called a *Newton direction*.

The following result establishes the convergence properties of Newton's method.

Theorem 2.1.1. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a twice-continuously differentiable function defined in an open set D. Assume $g(x^*) = 0$ for some $x^* \in D$ and that $\nabla^2 f(x^*)$ is nonsingular. Then there exists an open set S so that, for any $x_0 \in S$, the iterates generated by Newton's method are well defined, remain in S, and converge to x^* Q-superlinearly. If, in addition, $\nabla^2 f$ is Lipschitz continuous at x^*, then the rate of convergence is Q-quadratic.*

Proof. See Morè and Sorensen [MS84] □

2.2 Quasi-Newton methods

For many functions f, $\nabla^2 f(x)$ is unavailable: it may not exist or it may not be practical to compute. In this case it is possible to use the method of steepest descent (also known as the gradient descent method), which uses the search direction

$$p_k = -g_k.$$

Unfortunately, given appropriate conditions, the rate of convergence is only Q-linear.

A more suitable method, first described by Davidon [Dav59], is the quasi-Newton method, which, as the name suggests, shares several important features with Newton's method. Like Newton's method, a quasi-Newton method minimizes a function by forming a quadratic approximation of f at x, moving toward the minimizer of the model and repeating. During iteration k at x_k, a quasi-Newton method minimizes the quadratic model

$$q_k(x) = f_k + g_k^T(x - x_k) + \frac{1}{2}(x - x_k)^T H_k(x - x_k), \tag{2.2}$$

where H_k, called an approximate Hessian, is a positive-definite $n \times n$ matrix that—in some sense—approximates $\nabla^2 f(x_k)$. If H_k is positive definite, then $x_k + p_k$ uniquely minimizes $q_k(x)$ where p_k satisfies

$$H_k p_k = -g_k. \tag{2.3}$$

A search direction p_k that satisfies this equation is called a quasi-Newton direction.

The basic premise used to calculate H_k in all quasi-Newton methods is that some initial H_0 is defined—usually a multiple of the identity matrix—and each successive approximate Hessian is a low-rank update to the previous one. The most widely-used update is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update. Given H_k, the BFGS update defines H_{k+1} as:

$$H_{k+1} = H_k + \frac{1}{\gamma_k^T \delta_k}\gamma_k \gamma_k^T - \frac{1}{\delta_k^T H_k \delta_k}(H_k\delta_k)(H_k\delta_k)^T, \tag{2.4}$$

where

$$\gamma_k = g_{k+1} - g_k, \quad \text{and} \quad \delta_k = x_{k+1} - x_k.$$

Throughout this dissertation, we assume that $H_0 = \sigma I$, with $\sigma > 0$, when discussing BFGS updates. Note that p_0 will always be a positive multiple of the gradient descent step, with equality when $\sigma = 1$. A quasi-Newton method that uses a BFGS update is often called a *BFGS method*.

If f is twice-continuously differentiable, the approximate curvature $\gamma_k^T \delta_k$ is the first-order approximation of $\delta_k^T \nabla^2 f(x_k)\delta_k$ since

$$g(x_k + \delta_k) - g(x_k) = \nabla^2 f(x_k)\delta_k + \int_0^1 (\nabla^2 f(x_k + \xi\delta_k) - \nabla^2 f(x_k))\delta_k \, \mathrm{d}\xi.$$

The BFGS update is designed to satisfy a number of criteria:

- If H_k is symmetric, H_{k+1} is symmetric,
- If H_k is positive definite and $\gamma_k^T \delta_k > 0$, H_{k+1} is positive definite,
- $\delta_k^T H_{k+1} \delta_k = \gamma_k^T \delta_k$.

The third criterion is responsible for incorporating curvature information from the underlying problem into the approximate Hessian. The approximate curvature $\gamma_k^T \delta_k$ is said to be installed in H_{k+1}; that is, the approximate curvature of $f(x)$ at x_k in direction δ_k is the exact curvature of the quadratic model $q_{k+1}(x)$ in direction δ_k.

One consequence of this is that, if $\gamma_k^T \delta_k \leq 0$ and H_{k+1} is obtained from (2.4), then $\delta_k^T H_{k+1} \delta_k \leq 0$ and H_{k+1} is not positive definite. To avoid this, H_{k+1} is only updated using (2.4) if $\gamma_k^T \delta_k > \epsilon$, where ϵ is some positive tolerance close to 0. If $\gamma_k^T \delta_k \leq \epsilon$, H_{k+1} is defined to be H_k.

Although quasi-Newton methods converge Q-superlinearly given appropriate conditions, skipping the approximate Hessian update with any sort of regularity degrades performance significantly. In the worst case, when all updates are rejected and $H_k = \sigma I_n$ for all k, a quasi-Newton method would perform comparably to a gradient-descent method.

2.2.1 Solving the quasi-Newton equation

There are several widely-used methods to calculate p_k from (2.3) without the need to invert an $n \times n$ matrix at each step. One direct method, based on the Sherman-Morrison-Woodbury formula, stores and updates H_k^{-1} instead of H_k. Another approach uses an iterative method, such as the conjugate-gradient method, to minimize a related problem.

Because H_k is a positive-definite, symmetric matrix, p_k can also be computed using a Cholesky factorization of H_k. If M_k is a square upper-triangular matrix such that $M_k^T M_k = H_k$, then p_k can be obtained using one forward solve and one backward solve:

$$M_k p_k = q_k \text{ where } q_k \text{ satisfies } M_k^T q_k = -g_k.$$

Each forward or backward solve takes $O(n^2)$ operations, instead of the $O(n^3)$ operations required to solve (2.3).

It is prohibitively expensive to compute the Cholesky factor M_k from scratch each iteration. Fortunately, the recursive relation (2.4) defines a similar relationship between M_{k+1} and M_k, which makes this a viable method. The exact update is covered in more detail later in this chapter. It is worth noting that using a Cholesky factor is not the most efficient method for solving for p_k under normal circumstances. However, using a Cholesky factorization can provide substantial benefits, as will be demonstrated in Section 2.4.

2.2.2 Limited-memory variants

When n, the dimension of the problem, is large, storing and using H_k can become prohibitively expensive. One solution involves storing and using information not from all steps computed so far, but from only the last m steps. Such a method is called a *limited-memory method*. These variants are discussed more fully in Sections 3.4.4 and in Chapter 5.

2.3 Line searches

Line-search methods can be summed up as methods that repeatedly solve two subproblems: given x_k, the first calculates a search direction p_k; the second performs a line search on the function

$$\phi_k(\alpha) = f(x_k + \alpha p_k) \tag{2.5}$$

with $\alpha \geq 0$ to compute a step length α_k. Once α_k and p_k have been found, the next iterate is defined to be $x_{k+1} = x_k + \alpha_k p_k$ and the process is repeated until a solution is found. This section is concerned with the second subproblem. Throughout this section, iteration subscripts are suppressed when x and p are fixed throughout the length of a discussion and used if discussing values from more than one iteration with respect to the main optimization routine. When iteration subscripts are suppressed, the symbol $\bar{\alpha}$ is used to denote the output of a line search.

Performing a line search lessens the negative consequences when p_k is a poor step. This might happen, for instance, when p_k is a gradient descent step or when it is a Newton or quasi-Newton step and the model that $x_k + p_k$ minimizes does not adequately approximate f near x_k. It is usually not feasible to find the global minimizer of $\phi_k(\alpha)$, so line search algorithms seek to satisfy other conditions that ensure the convergence of the sequence $\{x_k\}$. When p_k is a quasi-Newton step, for example, one desirable property is that α_k should be 1 as often as possible in order to take advantage of the convergence properties near a solution. In contrast, when p_k is obtained from a conjugate-gradient method, the line search should be more accurate in order to compensate for poor scaling in the search direction.

2.3.1 Armijo line search

One desirable property is that any accepted step should satisfy the inequality

$$\phi(\alpha) < \phi(0).$$

This is not a strong enough condition to make a viable algorithm, however, so this condition is usually replaced with a stronger one, often called the Armijo condition:

$$\phi(\alpha) \leq \phi(0) + c_A \alpha \phi'(0), \tag{2.6}$$

or

$$f(x + \alpha p) \leq f(x) + c_A \alpha g^T p,$$

where $0 < c_A < \frac{1}{2}$.

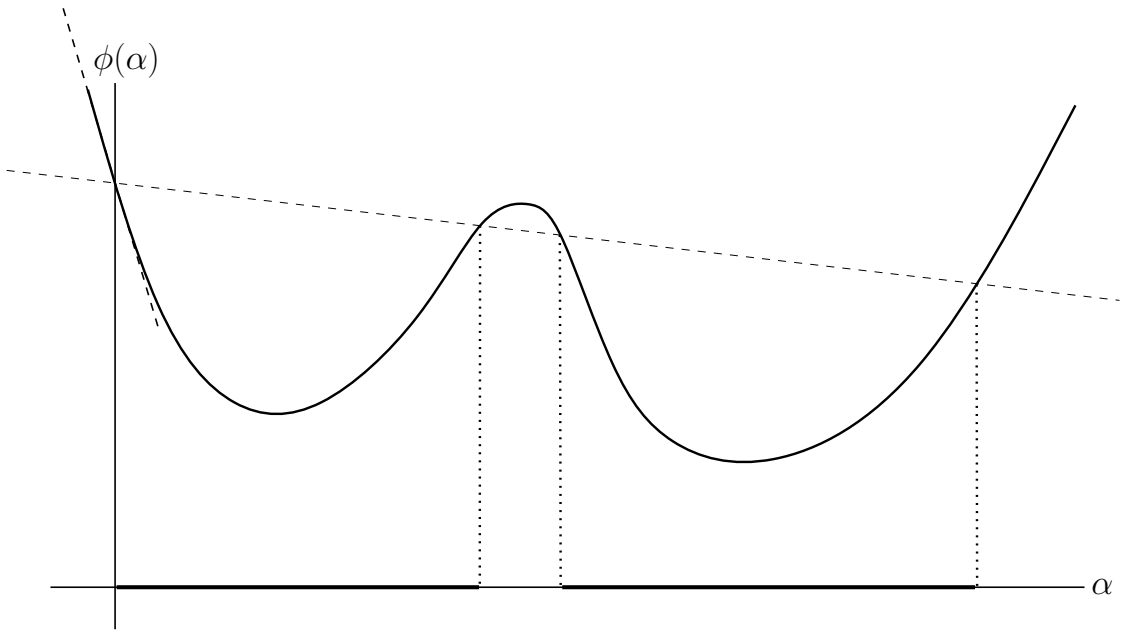

Figure 2.1: Armijo condition

If $\bar{\alpha}$ satisfies this equation for α, then it is said to sufficiently decrease the function and is called an *Armijo step*. The Armijo condition ensures that progress does not stall when step lengths are not decreasing to zero.

One widely-used algorithm that uses the Armijo condition as its termination criterion is the backtracking Armijo line search, often referred to as a backtracking line search or an Armijo line search. Such a line search algorithm starts with an initial step length, usually $\alpha = 1$, and decreases it toward 0 until (2.6) is satisfied. In a simple implementation, this is often done by contracting α by a constant factor, usually $\frac{1}{2}$. Assuming p is a descent direction and f is continuously differentiable, a backtracking line search is guaranteed to terminate after a finite number of iterations.

The following algorithm is an example of a simplified backtracking Armijo line search.

Algorithm 2.1 Simplified backtracking Armijo line search

choose $c_A \in (0, \frac{1}{2})$, $\sigma \in (0, 1)$

$\alpha \leftarrow 1$

while $f(x + \alpha p) > f(x) + c_A \alpha g^T p$

    $\alpha \leftarrow \sigma \alpha$

end

return α

Because the step length α goes to zero, it is possible for a backtracking line search not to make reasonable progress in reducing $f(x)$.

There is a more important problem that can occur when using an Armijo line search with a quasi-Newton method, as was hinted at in the previous section. To illustrate the problem, a few observations are needed first. Given an Armijo step $\alpha_k > 0$, the newly-obtained approximate curvature can be written as

$$\gamma_k^T \delta_k = (g_{k+1} - g_k)^T (\alpha_k p_k) = \alpha_k (g_{k+1}^T p_k - g_k^T p_k) = \alpha_k (\phi_k'(\alpha_k) - \phi_k'(0)).$$

Therefore, for all $\alpha > 0$:

$$\gamma_k^T \delta_k > 0 \iff \phi_k'(\alpha) > \phi_k'(0).$$

Recall from the previous section that the approximate Hessian is only updated when $\gamma_k^T \delta_k > 0$. Since the Armijo condition does not put any conditions on $\phi_k'(\alpha)$, it is possible (depending on the underlying function) that the newly obtained approximate curvature is nonpositive frequently. If the approximate Hessian cannot be updated regularly, performance can be poor.

2.3.2 Wolfe line search

To ensure that the approximate Hessian is updated as frequently as possible, another set of conditions— first described by Wolfe [Wol69]—are usually used with a quasi-Newton method. A step α is said to be a weak-Wolfe step if it satisfies the weak Wolfe conditions:

$$\phi'(\alpha) \geq c_W \phi'(0) \text{ and } \alpha \text{ is an Armijo step,} \tag{2.7}$$

where $c_W \in (c_A, 1)$.

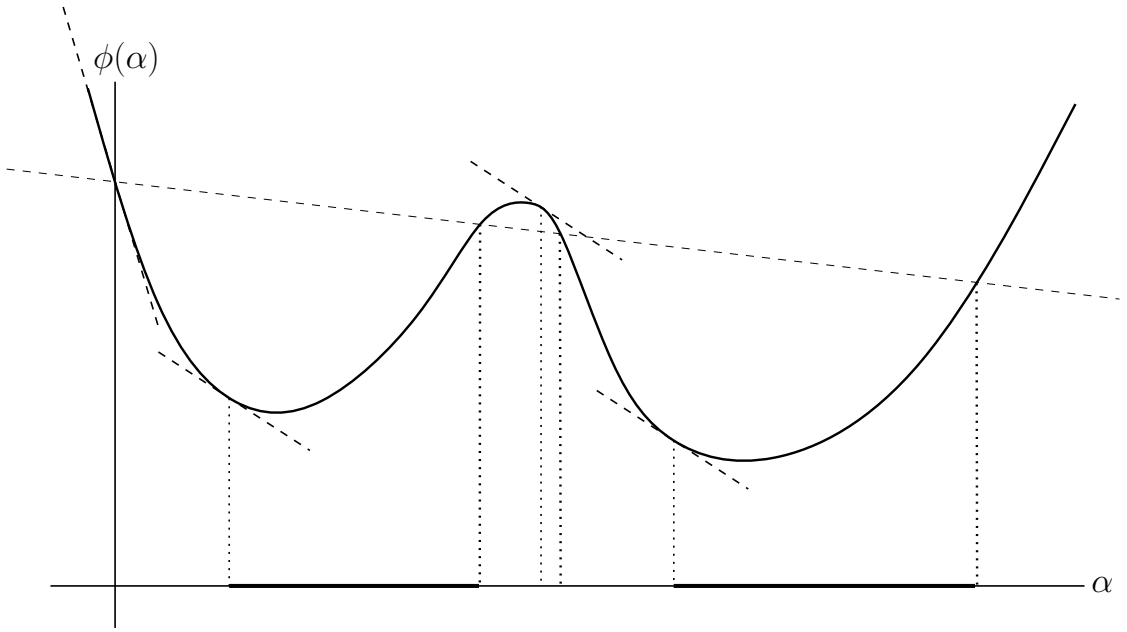

Figure 2.2: Weak-Wolfe conditions

Similarly, a step α is said to be a strong-Wolfe step—or simply a Wolfe step—if it satisfies the strong Wolfe conditions:

$$|\phi'(\alpha)| \leq c_W |\phi'(0)| \text{ and } \alpha \text{ is an Armijo step,} \tag{2.8}$$

where $c_W \in (c_A, 1)$ as before.

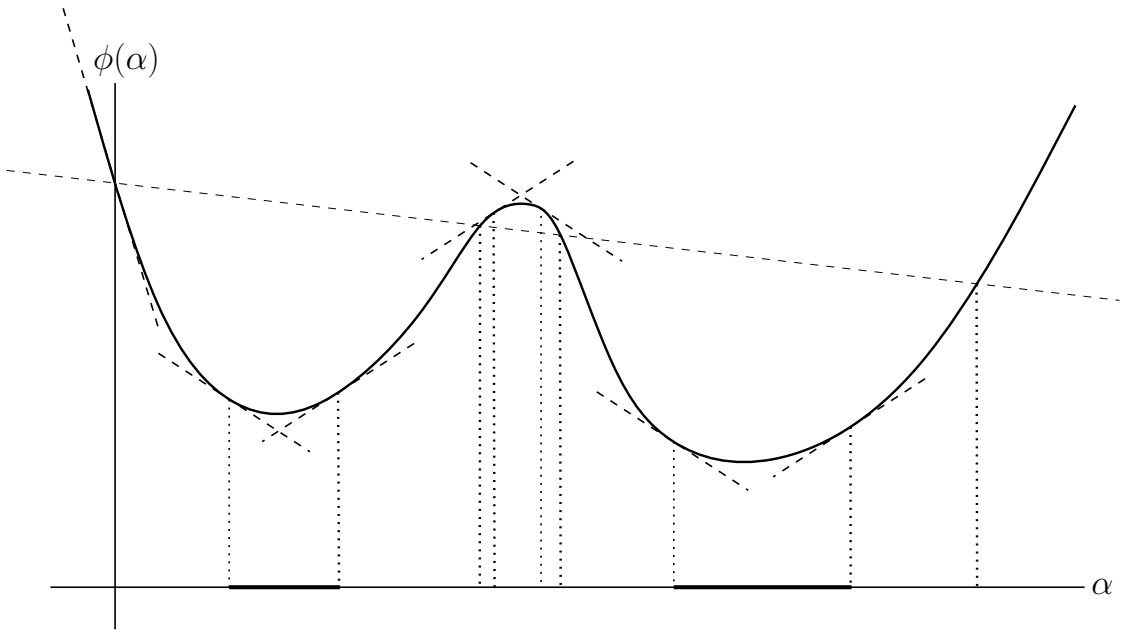

Figure 2.3: Strong-Wolfe conditions

Note that, by definition, a strong-Wolfe step is also a weak-Wolfe step. For the remainder of this dissertation, c_A is taken to be 10^{-4}.

If α_k is a weak-Wolfe step, it must hold that

$$\phi'_k(\alpha_k) \geq c_W \phi'_k(0) \implies \phi'_k(\alpha) > \phi'_k(0) \implies \gamma_k^T \delta_k > 0,$$

as $\phi'_k(0) < 0$. Therefore, if α_k satisfies the weak-Wolfe conditions, the approximate Hessian H_k can be updated.

Although a weak-Wolfe line search is enough to guarantee that H remains positive definite (assuming a weak-Wolfe step can be found), a strong-Wolfe line search places an upper bound on $\phi'(\alpha)$, which has the effect of forcing $\bar{\alpha}$ to be near a critical point of $\phi(\alpha)$. When using a Wolfe line search with a quasi-Newton method, a typical value of c_W is .9; in contrast, c_W is typically .1 when using a conjugate-gradient method. Roughly speaking, a value of c_W closer to 1 results in a "looser" or more approximate solution and a value closer to 0 gives a "tighter" or more accurate answer with respect to closeness to a critical point of $\phi(\alpha)$. All line-search methods that use a Wolfe line search in this dissertation use a strong-Wolfe line search, though there is no reason a weak-Wolfe line search could not be

used instead.

There are two related propositions that drive a Wolfe line search. The first is:

Proposition 2.3.1. *Let $\{\alpha_i\}$ be a strictly monotonically increasing sequence where $\alpha_0 = 0$, and let $\phi : \mathbb{R} \to \mathbb{R}$ be a continuously differentiable function. If it exists, let j be the smallest index where at least one of the following conditions (collectively called stage-one conditions) is true:*

1. *α_j is a Wolfe step, or*
2. *α_j is not an Armijo step, or*
3. *$\phi(\alpha_j) \geq \phi(\alpha_{j-1})$, or*
4. *$\phi'(\alpha_j) \geq 0$.*

If such a j exists, then there exists a Wolfe step $\alpha^* \in [\alpha_{j-1}, \alpha_j]$.

The proof is considered in Chapter 4. Note that the converse is not true: it is possible, for instance, that none of the stage-one conditions are satisfied for $j = 1$, but for there to be a Wolfe step in the interval $[0, \alpha_1]$. The second proposition is:

Proposition 2.3.2. *Let $\mathcal{I}$ be an interval whose endpoints are α_l and α_u (α_l not necessarily less than α_u) and let $\phi : \mathbb{R} \to \mathbb{R}$ be a continuously differentiable function. Assume α_l and α_u satisfy all of the following conditions (collectively called stage-two conditions):*

1. *α_l is an Armijo step, and*
2. *$\phi(\alpha_l) \leq \phi(\alpha_u)$ if α_u is an Armijo step, and*
3. *$\phi'(\alpha_l)(\alpha_u - \alpha_l) < 0$.*

Then there exists a Wolfe step $\alpha^* \in \mathcal{I}$.

From a geometric point of view, condition (3) is equivalent to saying that the vector $(\alpha_u - \alpha_l)$ is a descent direction of ϕ at α_l.

A typical Wolfe line search is a two-stage process. The notation and structure used here borrows heavily from Nocedal and Wright [NW99]. Assuming the initial step α_1 is not a Wolfe step—here subscripts will refer to iterations within the first stage of the line search algorithm and $\alpha_0 = 0$ by definition—pick successively larger step lengths, $\alpha_2, \alpha_3, \ldots, \alpha_{j-1}, \alpha_j, \alpha_{j+1}, \ldots, \alpha_{j_{\max}} = \alpha_{\max}$ until one of the stage-one conditions is satisfied.

By proposition (2.3.1), if the first stage-one condition is satisfied during iteration j, then the interval $[\alpha_{j-1}, \alpha_j]$ must contain a Wolfe step. At this point, the line search algorithm moves on to the second stage (or terminates successfully if condition (1) is satisfied).

If, after a finite number of iterations, the algorithm reaches $\alpha_{j_{\max}} = \alpha_{\max}$ and none of the conditions have been satisfied, it terminates with the Armijo step that gave the lowest function evaluation. An Armijo step can always be found, since not satisfying the conditions at j implies α_j is an Armijo step.

Assuming the first stage finds an interval that contains a Wolfe step, the first-stage function passes the endpoints α_{j-1} and α_j to the second-stage function, which labels them α_l and α_u such that the stage-two conditions hold. Next, the second-stage function interpolates the endpoints to calculate a best-guess step in the interval, α_{new}. The second-stage function recursively calls itself using α_{new} and an existing endpoint, labeling them so that the stage-two conditions hold again. This is repeated until α_{new} is a Wolfe step or until the function has called itself a set number of times. In practice, it rarely takes more than 1 or 2 interpolations to find a Wolfe step.

A practical implementation of a Wolfe line search is very complex. There are many ways to interpolate to obtain a new point in the second stage. Finite precision usually forces some sort of safeguarding during interpolation and gives rise to a whole host of issues, including how to handle cases when the function or step length are changing by a value near or less than machine precision. See Moré and Thuente [MT94] for a more detailed account.

A simplified Wolfe line search might be implemented in the following way.

Algorithm 2.2 Simplified Wolfe line search

choose $c_A \in (0,1)$, $c_W \in (c_A,1)$, $\alpha_{\max}$

$\alpha \leftarrow 1$, $\alpha_{\text{old}} \leftarrow 0$

while α is not a Wolfe step

    if α is not an Armijo step or $\phi(\alpha) \geq \phi(\alpha_{\text{old}})$

        $\alpha \leftarrow$ StageTwo(α_{old},α); break

    if $\phi'(\alpha) \geq 0$

        $\alpha \leftarrow$ StageTwo(α,α_{old}); break

    $\alpha_{\text{old}} \leftarrow \alpha$

    increase α towards $\alpha_{\max}$

end

return α

function StageTwo(α_l,α_u)

    choose α_{new} in interval defined by α_l and α_u using interpolation

    if α_{new} is a Wolfe step

        return α_{new}

    if α_{new} is not an Armijo step or $\phi(\alpha_{\text{new}}) \geq \phi(\alpha_l)$

        return StageTwo(α_l,α_{new})

    if $\phi'(\alpha_{\text{new}})(\alpha_u - \alpha_l) < 0$

        return StageTwo(α_{new},α_u)

    otherwise

        return StageTwo(α_{new},α_l)

end function

2.4 Reduced-Hessian methods

This section is meant to provide a brief summary of the work done by Gill and Leonard [GL01, GL03] on a family of optimization methods called *reduced-Hessian methods* (definition provided below). A more thorough account can be found in the original papers and Leonard's dissertation, *Reduced Hessian Quasi-*

Newton Methods for Optimization [Leo95].

Let $\mathcal{G}_k$ denote span$(g_0, g_1, \ldots, g_k)$, called the *gradient subspace*, and let $\mathcal{G}_k^{\perp}$ be the orthogonal complement of $\mathcal{G}_k$ in $\mathbb{R}^n$. Let the *column space of a* $\mu_1 \times \mu_2$ *matrix* M be denoted by col(M) so that

$$\text{col}(M) = \{Mx : x \in \mathbb{R}^{\mu_2}\}.$$

Fenelon [Fen81] proves the following important result.

Lemma 2.4.1. *Consider the BFGS method applied to a general nonlinear function. If* $H_0 = \sigma I$ *with* $\sigma > 0$*, then* $p_k \in \mathcal{G}_k$ *for all* k*. Further, if* $z \in \mathcal{G}_k$ *and* $w \in \mathcal{G}_k^{\perp}$*, then* $H_k z \in \mathcal{G}_k$ *and* $H_k w = \sigma w$*.* □

To simplify notation and provide better readability, iteration subscripts are suppressed for the remainder of the chapter except where otherwise noted. One exception that is made is for initial quantities. Initial quantities, i.e., quantities defined for iteration 0, are denoted with a 0 subscript. Bars above quantities are used to denote fully-updated values with respect to the current iteration. Subscripts are most commonly used instead to denote intermediate updates within an iteration.

Let B be a matrix whose columns form a basis for $\mathcal{G}$ and let Z be the orthonormal factor of the QR decomposition $B = ZT$, where T is a nonsingular, upper-triangular matrix, so that col(B) = col(Z). Let W be a matrix whose orthonormal columns span $\mathcal{G}^{\perp}$ and let $Q = \begin{pmatrix} Z & W \end{pmatrix}$. Then, solving the system

$$Hp = -g$$

is equivalent to solving the system

$$(Q^T H Q) Q^T p = -Q^T g,$$

for p.

If $H_0 = \sigma I$, then by the lemma above,

$$Q^T H Q = \begin{pmatrix} Z^T H Z & Z^T H W \\ W^T H Z & W^T H W \end{pmatrix} = \begin{pmatrix} Z^T H Z & 0 \\ 0 & \sigma I_{n-r} \end{pmatrix},$$

and

$$Q^T g = \begin{pmatrix} Z^T g \\ 0 \end{pmatrix},$$

where $r = \dim(\mathcal{G})$, and I_{n-r} is the identity matrix of order $n - r$. The matrix $Z^T H Z$ is called a *reduced Hessian* and $Z^T g$ is a *reduced gradient*.

Since $p \in \mathcal{G}$, it must hold that $p = ZZ^T p$, since $ZZ^T p$ is the projection of p onto $\text{col}(Z) = \mathcal{G}$. Therefore, the quasi-Newton step p can be calculated as

$$p = Zq, \text{ where } q \text{ satisfies } Z^T H Z q = -Z^T g. \tag{2.9}$$

Lemma 2.4.1 can be used to describe a quasi-Newton method from a geometric point of view. The quadratic model used in a quasi-Newton method builds up curvature information about a sequence of expanding gradient subspaces and fixes the curvature as σ in any direction orthogonal to the current gradient subspace. Since $p \in \mathcal{G}$, the algorithm can explicitly restrict the search direction to be in the column space of Z (or B), which is effectively what equation (2.9) does. From a practical point of view, it means that p can be obtained by solving a linear system of r variables instead of n. Methods that exploit this structure and solve p using (2.9) are known as *reduced-Hessian (RH) methods*. Also included in this description are algorithms that solve (2.9) for other choices of Z, which is more fully discussed later in the section.

As with regular quasi-Newton step calculations, p can be obtained in a variety of ways. Some of the more common ways are to store and update

- the inverse reduced Hessian,
- the vector pairs representing updates to an initial approximate Hessian that are used in conjunction with the Sherman-Morrison-Woodbury formula,
- or an upper-triangular Cholesky factor R where

$$R^T R = Z^T H Z. \tag{2.10}$$

When storing R, p can be obtained using one forward- and one backward-solve. All methods described in the remainder of this chapter use a Cholesky factor to calculate p.

2.4.1 A quasi-Newton implementation

In order to express a typical quasi-Newton method as a reduced-Hessian method, search direction p is obtained from equation (2.9) instead of (2.3). Using Cholesky factor R (2.10), p can be calculated using two triangular systems. In particular, $p = Zq$ where

$$Rq = d$$

and

$$R^T d = -Z^T g.$$

Since Z and R can change after each iteration, an effective algorithm needs to be able to update these two quantities in order to avoid having to orthogonalize B or to factor R from scratch at each step.

Note that H is not stored explicitly but can be reconstructed from Z and R as

$$\begin{aligned} H &= QQ^T HQQ^T \\ &= \begin{pmatrix} Z & W \end{pmatrix} \begin{pmatrix} Z^T HZ & 0 \\ 0 & \sigma I_{n-r} \end{pmatrix} \begin{pmatrix} Z^T \\ W^T \end{pmatrix} \\ &= ZZ^T HZZ^T + \sigma WW^T \\ &= ZR^T RZ^T + \sigma(I_n - ZZ^T). \end{aligned} \tag{2.11}$$

Updates to many quantities depends on whether $\bar{g} \in \mathcal{G}$. If $\bar{g} \in \mathcal{G}$, the gradient $\bar{g}$ is said to be *rejected*, otherwise it is said to be *accepted*. In exact arithmetic, $\bar{g}$ is rejected if $\bar{\rho} := \|(I - ZZ^T)\bar{g}\| = 0$.

The basis matrix B can be defined iteratively in the following way. Let $B_0 = g_0$. Then, given B, let

$$\bar{B} = \begin{cases} B & \text{if } \bar{g} \in \mathcal{G} \\ \begin{pmatrix} B & \bar{g} \end{pmatrix} & \text{otherwise.} \end{cases}$$

Other definitions for B are considered later in the chapter.

Similarly, let $Z_0 = g_0/\|g_0\|$ and

$$\bar{Z} = \begin{cases} Z & \text{if } \bar{\rho} = 0 \\ \begin{pmatrix} Z & \bar{z} \end{pmatrix} & \text{otherwise,} \end{cases}$$

where $\bar{z}$ satisfies $\bar{\rho}\bar{z} = (I - ZZ^T)\bar{g}$ and $\|\bar{z}\|_2 = 1$. Often some form of reorthogonalization is employed to prevent problems from round-off errors. It is worth noting that all the components of $\bar{Z}^T\bar{g}$—the reduced gradient used in the next iteration—are computed when updating Z. To capitalize on this, the intermediate quantity $w = Z^T\bar{g}$ is stored. Thus,

$$\bar{w} = \bar{Z}^T\bar{g} = \begin{cases} \begin{pmatrix} w \\ \bar{\rho} \end{pmatrix} & \text{if } \bar{g} \text{ is accepted,} \\ w & \text{if } \bar{g} \text{ is rejected.} \end{cases}$$

Since R is dependent on Z and H, it may have to be updated twice per iteration: once if the gradient $\bar{g}$ is accepted and again when the BFGS update is applied to the Cholesky factor.

The first update to R, denoted R_1, is applied only when $\bar{g}$ is accepted. The Cholesky factor R_1 must satisfy

$$R_1^T R_1 = \bar{Z}^T H \bar{Z} = \begin{pmatrix} Z^T H Z & Z^T H \bar{z} \\ \bar{z}^T H Z & \bar{z}^T H \bar{z} \end{pmatrix} = \begin{pmatrix} Z^T H Z & 0 \\ 0 & \sigma \end{pmatrix}.$$

Therefore,

$$R_1 = \begin{pmatrix} R & 0 \\ 0 & \sqrt{\sigma} \end{pmatrix},$$

if $\bar{g}$ is accepted. If $\bar{g}$ is rejected, $R_1 = R$.

Recall that the BFGS update to obtain $\bar{H}$ is

$$\bar{H} = H + \frac{1}{\gamma^T\delta}\gamma\gamma^T - \frac{1}{\delta^T H\delta}(H\delta)(H\delta)^T,$$

where $\delta = \bar{x} - x$ and $\gamma = \bar{g} - g$. Let

$$s = \bar{Z}^T\delta \text{ and } y = \bar{Z}^T\gamma,$$

and let $R_2 = R_1 + w_1 w_2^T$ so that

$$R_2^T R_2 = (R_1 + w_1 w_2^T)^T (R_1 + w_1 w_2^T) = \bar{Z}^T \bar{H} \bar{Z}, \tag{2.12}$$

where

$$w_1 = \frac{1}{\|R_1 s\|} R_1 s \;\text{ and }\; w_2 = \frac{1}{\sqrt{y^T s}} y - R_1^T w_1.$$

Note that w_1 and w_2 have no relation to the reduced vector $w = Z^T \bar{g}$. Since R_2 is not upper triangular, $\bar{R}$ is defined to be the Cholesky factor of $R_2^T R_2$.

One important note, which has a significant impact on the algorithms in Chapter 5, is that equation (2.12) only holds if $\delta = \bar{Z}\bar{Z}^T\delta$, i.e., $\delta \in \text{col}(\bar{Z})$. Since $\delta = \bar{x} - x = \alpha p$ and $p \in \text{col}(\bar{Z})$, equation (2.12) holds.

To prevent having to compute matrix-vector products with Z from scratch, when possible, intermediate values are stored and updated. These values include: $q = Z^T p$, $w = Z^T \bar{g}$, and $v = Z^T g$. We will make use of the same variable descriptions in the sections and chapters that follow except where otherwise noted.

A basic implementation of a reduced-Hessian quasi-Newton method is shown below.

Algorithm 2.3 Algorithm RH

Choose $\sigma > 0$ and x

$g \leftarrow \nabla f(x)$

$Z \leftarrow g/\|g\|$, $R \leftarrow \sqrt{\sigma}$, $v \leftarrow \|g\|$

while not converged

$d \leftarrow -R^{-T}v$

$q \leftarrow R^{-1}d$

$p \leftarrow Zq$

Compute Wolfe step α

$x \leftarrow x + \alpha p$, $g \leftarrow \nabla f(x)$

$w \leftarrow Z^T g$

Update Z, R, w, v, q based on whether gradient is accepted

$s \leftarrow \alpha q$, $y \leftarrow u - v$

Apply BFGS update to R if $y^T s > 0$

$v \leftarrow w$

end

2.4.2 Reinitialization

For some problems, particularly when $\nabla^2 f(x^*)$ is ill-conditioned, using a BFGS method with $H_0 = \sigma I$ often results in poor convergence [GL01]. To combat this, many algorithms implement some sort of Hessian scaling. One major advantage to using a reduced-Hessian method with a Cholesky factorization is that σ, which represents the curvature in $\mathcal{G}^{\perp}$, is made explicit.

Given a vector $z \in \mathcal{G}$ and equation (2.11), the approximate curvature in direction z is

$$z^T H z = (Z^T z)^T R^T R (Z^T z). \tag{2.13}$$

The second term in the right-hand-side of (2.11) vanishes since $z \in \mathcal{G}$. Likewise, if $w \in \mathcal{G}^{\perp}$, then

$$w^T H_k w = \sigma w^T w. \tag{2.14}$$

If a better estimate exists for the curvature information in $\mathcal{G}^{\perp}$, denoted as $\bar{\sigma}$, the approximate Hessian can be rescaled easily when computing the first partial

update to R, R_1. The new first partial update becomes

$$R_1 = \begin{pmatrix} R & 0 \\ 0 & \sqrt{\bar{\sigma}} \end{pmatrix},$$

if $\bar{g}$ is accepted. If $\bar{g}$ is rejected, $R_1 = R$.

This process is called *reinitialization*, since it reinitializes the curvature of the quadratic model in $\mathcal{G}^\perp$ to $\bar{\sigma}$. Provided that $\bar{\sigma} > 0$ and that $R^T R$ is positive definite, then $R_1^T R_1$ and H are positive definite, too.

As with other rescaling techniques, a reduced-Hessian algorithm with reinitialization will not produce the same iterates as a standard quasi-Newton method in exact arithmetic. Equations (2.13) and (2.14) imply that reinitializing affects curvature in all of $\mathcal{G}^\perp$, but leaves the already-computed curvature in $\mathcal{G}$ unchanged.

Gill and Leonard present several suggested values for $\bar{\sigma}$. We use the same value they use in their numerical tests [GL03], namely,

$$\bar{\sigma} = \frac{\gamma^T \gamma}{\gamma^T \delta}.$$

One important factor in determining whether to use the technique of reinitialization is the dimension of the problem. Shanno and Phua [SP78] observe that using such a technique improves performance when n is large but results in mixed performance when n is small.

For more information about alternative choices for $\bar{\sigma}$ and more about reinitialization, see Gill and Leonard [GL01, GL03].

2.4.3 Lingering

The following lemma, proved by Gill and Leonard [GL01], motivates a technique known as "lingering". Throughout Section 2.4.3, subscripts are used to denote iterations.

Lemma 2.4.2. *Suppose that the BFGS method with an exact line search is applied to a strictly convex quadratic function $f(x)$. If $H_0 = \sigma I$, then at the start of iteration k,*

(a) *x_k minimizes $f(x)$ on the linear manifold $\mathcal{M}(\mathcal{G}_k)$, with*

$$\mathcal{M}(\mathcal{G}_k) = \{x_0 + z : z \in \mathcal{G}_k\},$$

(b) *the curvature of the quadratic model is exact on the k-dimensional subspace $\mathcal{G}_{k-1}$. Thus, $z^T H_k z = z^T \nabla^2 f(x) z$ for all $z \in \mathcal{G}_{k-1}$.* □

If $f(x)$ is a strictly convex quadratic function, then $\mathcal{M}(\mathcal{G}_{k-1}) \subset \mathcal{M}(\mathcal{G}_k)$, and moving from x_k to x_{k+1} can be thought of as "stepping onto" the larger manifold $\mathcal{M}(\mathcal{G}_k)$ from the smaller manifold $\mathcal{M}(\mathcal{G}_{k-1})$.

This idea can be extended to the case when $f(x)$ is a nonlinear function and the gradient $g_k \notin \mathcal{G}_{k-1}$. In this case, $\mathcal{M}(\mathcal{G}_{k-1}) \subset \mathcal{M}(\mathcal{G}_k)$, and x_{k+1} also moves onto the larger manifold $\mathcal{M}(\mathcal{G}_k)$. Unlike in the quadratic case, it is unlikely that x_{k+1} minimizes $f(x)$ on the manifold $\mathcal{M}(\mathcal{G}_k)$.

On the other hand, if $g_k \in \mathcal{G}_{k-1}$, then $\mathcal{M}(\mathcal{G}_{k-1}) = \mathcal{M}(\mathcal{G}_k)$ and moving from x_k to x_{k+1} can be thought of as continuing to minimize on the same manifold. Thus, the manifold over which a BFGS method minimizes a function only expands when new gradients are accepted, i.e., when new gradients are not in the current gradient subspace.

Gill and Leonard use this observation to propose a new algorithm. The algorithm is driven by one main idea: let iterates remain—or "linger"—on a manifold so long as a good reduction in $f(x)$ is being achieved, even if new gradients are accepted. Once $f(x)$ is no longer being reduced by a satisfactory amount, the manifold is allowed to expand by one dimension and the process is repeated.

A more thorough discussion of lingering and its implementation in an RH method can be found in Gill and Leonard [GL01]. Although lingering can be added to any proposed algorithm in Chapter 5, its contributions are minimal in creating a more effective algorithm.

2.4.4 Limited-memory variants

Like full-memory quasi-Newton methods, full-memory reduced-Hessian methods are effective only on problems with relatively few variables, due to storage and computational requirements. Since the reduced-Hessian matrix grows to

dimension $n \times n$ after enough iterations, working with this matrix can become infeasible if n is large enough. One solution to this problem is to implement a limited-memory variant that can be applied to all RH algorithms.

Instead of storing Z as an orthonormal matrix whose columns span all of $\mathcal{G}$, the number of columns of Z is restricted to be at most m. The most straightforward approach is to let the orthonormal columns of Z correspond to the m most-recent accepted gradients. This approach tends to result in poor convergence. Several papers [GL03, Sie94] suggest that poor performance is caused by the fact that discarding the oldest gradient from the basis removes the property of finite termination on a quadratic.

Since the subspace generated by search directions is the same as the subspace generated by gradients when using a quasi-Newton method with the BFGS update, Siegel [Sie94] and Gill and Leonard [GL03] suggest taking the columns of B to be search directions instead of gradients. A new orthonormal matrix Z and triangular matrix T are defined from the QR factorization $B = ZT$. Discarding the oldest search direction to maintain at most m columns in Z (or B) preserves the finite termination property described above.

Implementing a limited-memory RH algorithm requires that the triangular factor T be stored, as well as either B or Z. Computationally, less work needs to be done when B and T are stored [GL03]. An algorithm is called an *implicit method* if B and T are stored, since Z is only computed implicitly. An algorithm is called an *explicit method* if Z and T are stored. Although less work is done per iteration when using an implicit method, it is not practical to reorthgonalize new columns being added to Z. As a consequence, m should be relatively small (6 or less) when using an implicit method. An explicit method is more appropriate when m is larger. The focus of the next subsection is on implicit methods, but more information can be found about both methods in Gill and Leonard [GL03].

Limited-memory variations exist for all RH methods described previously in this section. A prefix "L-" or "L" will be used on any RH method to describe the limited-memory variant, e.g., Algorithm L-RHR is the limited-memory equivalent to Algorithm RHR. While it is possible to have a limited-memory RH algorithm

with lingering, in practice it is not a useful combination. To more fully describe how a limited-memory RH method is implemented, the remainder of the chapter is used to address the additional practical steps necessary to create Algorithm L-RHR from Algorithm RHR using an implicit method.

When transitioning from a full-memory to a limited-memory algorithm, much of an RH algorithm remains the same except in several key areas: (i) storing T; (ii) potentially accessing Z implicitly; (iii) forming B from search directions instead of gradients; and (iv) dropping columns from B when necessary.

Forming a basis of search directions

The only way that Z and B gain new columns is when a new gradient, $\bar{g}$, is accepted. Since the next search direction $\bar{p}$ is unavailable when $\bar{g}$ is computed, $\bar{g}$ is appended to B temporarily until $\bar{p}$ can be "swapped in" near the beginning of the next iteration. Let B_1 denote the basis matrix after $\bar{g}$ is accepted, i.e.,

$$B_1 = \begin{pmatrix} B & \bar{g} \end{pmatrix}.$$

Then, after $\bar{p}$ is computed near the beginning of the next iteration, the final basis matrix is given by

$$\bar{B} = \begin{pmatrix} B & \bar{p} \end{pmatrix}.$$

Updates to obtain B_1 and $\bar{B}$ necessitate similar updates to T. Namely,

$$T_1 = \begin{pmatrix} T & w \\ 0 & \bar{\rho} \end{pmatrix},$$

and

$$\bar{T} = \begin{pmatrix} \begin{pmatrix} T \\ 0 \end{pmatrix} & \bar{q} \end{pmatrix}.$$

where $w = Z^T\bar{g}$, $\bar{\rho} = \|(I - ZZ^T)\bar{g}\|_2$, and $\bar{q} = \bar{Z}^T\bar{p}$. These updates to B and T are well-defined, since $\bar{p}^T\bar{g} \neq 0$ [GL03, Leo95]. If $\bar{g}$ is rejected, then $\bar{B} = B_1 = B$ and $\bar{T} = T_1 = T$.

Given that $B_1 = Z_1T_1$ and $\bar{B} = \bar{Z}\bar{T}$, then $Z_1 = \bar{Z}$. This is important because $\bar{p}$ should be calculated using a value identical to $\bar{Z}$. Since $Z_1 = \bar{Z}$, no additional updates to the Cholesky factor R are needed beyond the ones already required for the full-memory RH method.

Discarding the oldest search direction from the basis matrix

Since the columns of B are linearly independent, rank(B) can be used to denote the number of columns in B. The same holds true for Z. If gradient $\bar{g}$ is accepted and rank(B_1) $> m$, then a limited-memory RH method drops the first column from B_1. If rank(B_1) $> m$, and p_0 is the first column of B_1, then B_2 is defined to be the $n \times m$ matrix that satisfies the equation

$$B_1 = \begin{pmatrix} p_0 & B_2 \end{pmatrix}.$$

If $\bar{g}$ is rejected or rank(B_1) $\leq m$, then $B_2 = B_1$. Otherwise, computing B_2 requires that T_2, Z_2, R_2, and any reduced vector (such as $w_2 = Z_2^T \bar{g}$) be calculated, where Z_2 and T_2 are QR factors of B_2, and R_2 is the Cholesky factor of $Z_2^T H Z_2$. For more information about how these quantities are updated, see Section 5.3.3, which describes updates for a broader class of problems.

An example of how the basis matrix is updated is given here. Using iteration subscripts for the gradients, consider the case where the current iteration is $k = m - 1$ and all gradients computed have been accepted. When adding the next gradient, g_m, the basis matrix is updated in the following way.

$$\begin{aligned} B &= \begin{pmatrix} p_0 & p_1 & \cdots & p_{m-1} \end{pmatrix}, \\ B_1 &= \begin{pmatrix} p_0 & p_1 & \cdots & p_{m-1} & g_m \end{pmatrix}, \\ B_2 &= \begin{pmatrix} p_1 & \cdots & p_{m-1} & g_m \end{pmatrix}, \\ \bar{B} &= \begin{pmatrix} p_1 & \cdots & p_{m-1} & p_m \end{pmatrix}. \end{aligned}$$

Note that, if g_i is rejected for some $i \in [1, m]$, then rank(B_1) $\leq m$, and $B = B_1 = B_2 = \bar{B}$.

A limited-memory, implicit implementation of a reduced-Hessian quasi-Newton method with reinitialization is shown below.

Algorithm 2.4 Algorithm LRHR

Choose m, $\sigma > 0$ and x

$g \leftarrow \nabla f(x)$

$B \leftarrow g$, $T \leftarrow \|g\|$, $R \leftarrow \sqrt{\sigma}$, $v \leftarrow \|g\|$

while not converged

$d \leftarrow -R^{-T}v$

$q \leftarrow R^{-1}d$

$p \leftarrow B(T^{-1}q)$

if last gradient was accepted,

Swap last column of B with p and update T

Compute Wolfe step α

$x \leftarrow x + \alpha p$, $g \leftarrow \nabla f(x)$

$w \leftarrow Z^T g$

Update B, T, R, w, v, q based on whether gradient is accepted

$s \leftarrow \alpha q$, $y \leftarrow u - v$

Apply BFGS update to R if $y^T s > 0$

Compute new σ and reinitialize R

if rank$(B) > m$,

Drop oldest basis vector in B and update T, R, and w

$v \leftarrow w$

end

2.4.5 Other reduced-Hessian methods

If m_k is defined to be the affine model $f(x_k) + g_k^T(x - x_k)$, then, given x_k and p_k, the Armijo condition expressed in terms of α,

$$f(x_k + \alpha p_k) \leq f(x_k) + c_A \alpha g_k^T p_k,$$

can be expressed in the more general form,

$$f(x_k) - f(x_k + d_k) \geq \eta(m_k(x_k) - m_k(x_k + d_k)), \tag{2.15}$$

where $d_k = \alpha p_k$ is a function of α and $0 \leq \eta = c_A \leq \frac{1}{2}$.

Thus, a quasi-Newton line-search method can be viewed in a more general way. Given an iterate x_k, a direction p_k is obtained as the solution to unconstrained optimization problem

$$\underset{p\in\mathbb{R}^n}{\operatorname{argmin}}\, g_k^T p + \frac{1}{2} p^T H_k p,$$

where H_k is a positive-definite symmetric approximate Hessian. Then, a sequence of step lengths are tested (starting with a step length of one) using condition (2.15) and α_k is chosen to be the first step that satisfies the condition. Finally, the next iterate is defined to be $x_{k+1} = x_k + d_k = x_k + \alpha_k p_k$.

In contrast to a quasi-Newton line-search method, a quasi-Newton *trust-region method* obtains the step to the next iterate, d_k, from the constrained optimization problem,

$$\underset{d\in\mathbb{R}^n}{\operatorname{argmin}}\, g_k^T d + \frac{1}{2} d^T H_k d \;\text{ such that }\; \|d\| \leq \delta_k, \tag{2.16}$$

where H_k is an approximate Hessian (not necessarily positive-definite) and δ_k, called the *trust-region radius*, is chosen to be small enough so that d_k satisfies condition (2.15). In this case, $m_k(x)$ can be defined more generally as some affine or quadratic model that approximates f at x_k. δ_{k+1} is initially estimated by δ_k and can grow if f is reduced significantly compared to what the model m_k predicted.

Informally speaking, a line-search method first computes the direction to travel along, then decides how far away to step based on the reduction in f. On the other hand, a trust-region method decides how far away from the iterate it is willing to travel, and only then decides the direction—which can be recalculated with a smaller trust-region radius if f is not reduced sufficiently. Because the subproblem is a nonconvex constrained optimization problem, it is significantly harder to solve than the corresponding line-search method subproblem to obtain the next iterate. A thorough discussion on trust-region methods can be found in Conn, Gould and Toint [CGT00]

Trust-region methods are introduced here only to point out that recent work has been done by Wang and Yuan [WY06] that bring a reduced-Hessian framework to the quasi-Newton trust-region subproblem (2.16). They do this, in part, by extending the theory to allow for a non-positive-definite H_k. Because they

are able to use a reduced-Hessian method, they are able to implement lingering and reinitialization, as well.

Chapter 3

Active-Set Methods for Box-Constrained Optimization

While unconstrained optimization routines are an important and widely-used class of algorithms, many optimization problems require some sort of restriction on the variables. The simplest type is box-constrained optimization. Problems of this type have the form

$$\min_x f(x) \text{ such that } l \leq x \leq u, \tag{3.1}$$

where x, l, $u \in \mathbb{R}^n$, $l \leq x \leq u$ is defined componentwise, and components of l and u may be negative or positive infinity, respectively, to signify no lower or upper bound on that component of x. All algorithms discussed in this chapter assume that f is continuously differentiable. None of the algorithms use second-derivative information about f to minimize it, although some authors require conditions on $\nabla^2 f$ when discussing convergence results. For this reason, the methods outlined in this chapter are most effective when computing $\nabla^2 f$ is not practical or possible. If $\nabla^2 f$ is readily accessible, a method more like Newton's method for the box-constrained case is likely more appropriate than the methods described here.

3.1 Definitions

In order to discuss optimization algorithms for box-constrained optimization, several definitions are needed. A point x is *feasible* if it satisfies $l \leq x \leq u$. The set of all feasible points, called the *feasible set*, is denoted by $\mathbb{F}$, where

$$\mathbb{F} = \{x \in \mathbb{R}^n : l \leq x \leq u\}. \tag{3.2}$$

The *active set at* x is defined to be

$$\mathcal{A}(x) = \{i : x_i = l_i \text{ or } x_i = u_i\}.$$

Given a feasible point x and its gradient $g = \nabla f(x)$, the *lower working set at* x is defined to be

$$\mathcal{L}(x) = \{i : x_i = l_i \text{ and } g_i > 0\},$$

and the *upper working set at* x is

$$\mathcal{U}(x) = \{i : x_i = u_i \text{ and } g_i < 0\}.$$

Given $\mathcal{L}(x)$ and $\mathcal{U}(x)$, the *working set at* x is defined to be

$$\mathcal{W}(x) = \mathcal{L}(x) \cup \mathcal{U}(x).$$

Given an indexing set $\mathcal{I}$, the *complement of* $\mathcal{I}$ is defined to be

$$\mathcal{I}^c = \{1, 2, \ldots, n\} \setminus \mathcal{I}.$$

The set $\mathcal{W}^c(x)$ will sometimes be referred to as the *free set*, denoted by $\mathcal{F}(x)$. For brevity, we define

$$\mathcal{I}_k = \mathcal{I}(x_k), \qquad \mathcal{I}^* = \mathcal{I}(x^*), \qquad \mathcal{I}_* = \mathcal{I}(x_*),$$

where $\mathcal{I}(x)$ is usually $\mathcal{A}(x)$ or $\mathcal{W}(x)$, and x^* and x_* are defined later.

Let $P(x) = P(x, l, u)$ be the closest point in $\mathbb{F}$ to x, so that $P(x)$ is the unique solution to

$$\operatorname*{argmin}_{y \in \mathbb{F}} \|x - y\|_2.$$

Then $P(x)$ is called the *projected point of x with respect to $\mathbb{F}$*. Given the definition of $\mathbb{F}$, $P(x)$ can be defined componentwise as

$$[P(x)]_i = \begin{cases} l_i & \text{if } x_i < l_i, \\ u_i & \text{if } x_i > u_i, \\ x_i & \text{otherwise.} \end{cases}$$

Given l and u, the *projected direction of p at x* is defined to be $P_x(p)$, where $P_x(p)$ is defined componentwise as

$$[P_x(p)]_i = \begin{cases} 0 & \text{if } x_i = l_i \text{ and } p_i < 0, \\ 0 & \text{if } x_i = u_i \text{ and } p_i > 0, \\ p_i & \text{otherwise.} \end{cases}$$

Given x and p, p is said to be a *feasible direction at x* if $p = P_x(p)$. The set of all feasible directions at x is denoted by

$$\mathbb{P}_x = \{p \in \mathbb{R}^n : p = P_x(p)\}.$$

In a like manner, we define $P_{\mathcal{I}}(p)$ to be the *projected direction of p with respect to $\mathcal{I}$*, where $P_{\mathcal{I}}(p)$ is defined componentwise as

$$[P_{\mathcal{I}}(p)]_i = \begin{cases} 0 & \text{if } i \in \mathcal{I} \\ p_i & \text{if } i \notin \mathcal{I}. \end{cases}$$

Direction p is said to be a *feasible direction with respect to $\mathcal{I}$* if $p = P_{\mathcal{I}}(p)$. The set of all feasible directions with respect to $\mathcal{I}$ is denoted by

$$\mathbb{P}_{\mathcal{I}} = \{p \in \mathbb{R}^n : p = P_{\mathcal{I}}(p)\}.$$

For example, given any feasible point x and direction $p \in \mathbb{P}_{\mathcal{A}(x)}$, $x + p$ is on the same "face" as x; i.e., $\mathcal{A}(x) \subseteq \mathcal{A}(x + p)$.

Given an index set $\mathcal{I} \subseteq \{1, 2, \ldots, n\}$ and a symmetric $n \times n$ matrix M with elements $M_{i,j}$, matrix M is said to be *diagonal with respect to $\mathcal{I}$* if

$$M_{ij} = 0 \text{ for all } i \in \mathcal{I}, \ \ j \in \{1, 2, \ldots, n\}, \text{ and } j \neq i.$$

All methods discussed in this chapter make use of projected paths. To that end, the *projected path at x in direction p* is defined to be

$$x(\alpha) = P(x + \alpha p). \tag{3.3}$$

Note that $x(\alpha)$ is a piecewise linear path. Due to the iterative nature of the algorithms discussed in this chapter, we define

$$x_k(\alpha) = P(x_k + \alpha p_k).$$

Recall definition (2.5) on page 12, which defines $\phi(\alpha) = f(x + \alpha p)$. In a like manner, we define

$$\psi(\alpha) = f(x(\alpha)), \tag{3.4}$$

and

$$\psi_k(\alpha) = f(x_k(\alpha)). \tag{3.5}$$

As f is a continuous and differentiable function, and $x(\alpha)$ is a piecewise linear path, $\psi(\alpha)$ and $\psi_k(\alpha)$ are continuous, piecewise-differentiable functions.

Given an algorithm that generates a sequence of iterates $\{x_0, x_1, x_2, \ldots\}$ (referred to as $\{x_k\}$ hereafter) from an initial feasible point x_0, let x_* denote a limit point of $\{x_k\}$. The first-order necessary conditions for optimality at x can be written as

$$g^T p \geq 0 \ \text{ for all } \ p \in \mathbb{P}_x$$

or componentwise as

$$g_i = 0 \ \text{ if } \ i \notin \mathcal{A}^*, \quad g_i \leq 0 \ \text{ if } \ x_i = l_i, \quad g_i \geq 0 \ \text{ if } \ x_i = u_i,$$

for all $i \in \{1, \ldots, n\}$, where $g = g(x)$.

A point that satisfies the above conditions is called a *stationary point* and is denoted by the symbol x^*. Note that, since no knowledge of $\nabla^2 f$ is assumed, a stationary point is considered a solution. If x^* is a stationary point, it is said to satisfy the *strict complementarity property* if, for each component $i \in \{1, 2, \ldots, n\}$, exactly one of the conditions

$$g_i^* = 0 \ \text{ or } \ i \in \mathcal{A}^*$$

holds, where $g^* = g(x^*)$. If, for some i, $g_i^* = 0$ and $i \in \mathcal{A}^*$, the point x^* is called *degenerate*.

3.2 Gradient-projection methods

In its simplest form, a gradient-projection method is an extension of the steepest-descent method to box-constrained optimization. Most early published results and algorithms are for the specific case where $l = 0$ and u is a vector whose components are all positive infinity. Extending convergence results to the general box-constrained case for such algorithms is not done here but is straightforward. Several papers discuss algorithms where the feasible region is a general convex set. Given the scope of this chapter, such regions are simplified and represented as $\mathbb{F}$ in this chapter.

Proposed independently by Goldstein [Gol64] and Levitin and Polyak [LP66], the general formula used to obtain a sequence of converging iterates is

$$x_{k+1} = P(x_k + \alpha_k p_k) = x_k(\alpha_k), \quad \text{where} \quad x_0 \in \mathbb{F}. \tag{3.6}$$

The algorithms proposed in both papers [Gol64, LP66] define step $p_k = -g_k$, require $g(x)$ to be Lipschitz continuous with Lipschitz constant λ, and define α_k such that

$$\epsilon \leq \alpha_k \leq \frac{2}{\lambda}(1 - \epsilon)$$

holds, where $0 < \epsilon < 1$. Given these conditions, Goldstein, Levitin and Polyak show that any limit point x_* of $\{x_k\}$ is a stationary point.

To avoid the problematic condition of needing a Lipschitz constant, McCormick [McC69] defines $p_k = -g_k$ but takes α_k to be a global minimizer of

$$\min_{\alpha > 0} \psi_k(\alpha),$$

where $\psi_k(\alpha)$ is defined by (3.5). Given a continuously differentiable f, McCormick proves that any limit point of $\{x_k\}$ is a stationary point. Although this algorithm does not require a Lipschitz constant, performing an exact minimization at every iteration to obtain α_k renders the algorithm impractical.

The first practical projected-gradient method, proposed by Bertsekas [Ber76], selects α_k by using a backtracking Armijo-like line search and sets $p_k = -g_k$. Given $0 < c_A < \frac{1}{2}$, α_k must satisfy

$$\psi_k(\alpha) < \psi_k(0) + c_A g_k^T (x(\alpha) - x_k). \tag{3.7}$$

When there are no constraints, condition (3.7) is identical to the Armijo condition,

$$\phi_k(\alpha) < \phi_k(0) + c_A \phi_k'(0)\alpha$$

where

$$\phi_k(\alpha) = f(x_k + \alpha p_k). \tag{3.8}$$

Bertsekas shows that if f is continuously differentiable, $\{x_k\}$ is a sequence of points generated by (3.6) where α_k satisfies (3.7), and $p_k = -g_k$, then any limit point x_* of $\{x_k\}$ is a stationary point. Bertsekas also proves the following result.

Theorem 3.2.1. *Let x^* be an isolated nondegenerate local minimizer of f. Assume f is twice continuously differentiable on the feasible portion of some neighborhood near x^*, and*

$$\mu_1 p^T p \leq p^T \nabla^2 f(x^*) p \leq \mu_2 p^T p$$

for some μ_1, $\mu_2 > 0$ and for all $p \in \mathbb{P}_{\mathcal{A}^*}$. Let $\{x_k\}$ be a sequence generated by (3.6) where $p_k = -g_k$ and α_k satisfies (3.7). Then there exists $\delta > 0$ such that, if for some j, $\|x^* - x_j\|_2 < \delta$, then $\{x_k\}$ converges to x^* and $\mathcal{A}_i = \mathcal{A}^*$ for all $i > j$. □

Because the active set at x^* is identified after a finite number of iterations given suitable conditions, Bertsekas observes that all subsequent iterations can be viewed as an unconstrained minimization on a subset of the variables, i.e.,

$$\min_x f(x) \text{ such that components } x_i \text{ remain fixed, where } i \in \mathcal{A}^*.$$

To take advantage of this finite identification property, Bertsekas suggests switching to a superlinear convergent unconstrained algorithm such as a Newton or quasi-Newton method when the active set stabilizes. Bertsekas [Ber82] also proves that some algorithms using search directions other than $p_k = -g_k$ identify the active set at x^* in finite number of iterations (see Section 3.3).

Calamai and Moré [CM87] generalize the sufficient conditions for an algorithm to possess the finite identification property using projected gradients. Given a feasible point x, we define the projected gradient at x to be

$$g_F(x) = -P_x(-g(x))$$

and

$$g_{F,k} = g_F(x_k) = -P_{x_k}(-g_k).$$

(The definition presented here for $g_F(x)$ is the negative of the projected gradient given in Calamai and Moré.) They observe that x^* is a stationary point if and only if $g_F(x^*) = 0$. Further, they prove the following result.

Theorem 3.2.2. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be continuously differentiable on $\mathbb{F}$ and let $\{x_k\}$ be an arbitrary sequence in $\mathbb{F}$ that converges to x^*. If $\{\|g_{F,k}\|\}$ converges to zero and x^* is nondegenerate then $\mathcal{A}_k = \mathcal{A}^*$ for all k sufficiently large.* □

The significance of this theorem is that *any* algorithm that drives $\|g_{F,k}\|$ to zero identifies $\mathcal{A}^*$ in a finite number of iterations, provided x^* is nondegenerate. Calamai and Moré also prove that the working set $\mathcal{W}^*$ is identified in a finite number of iterations as well. This has ramifications for projected-search methods, which are described in the next section.

In addition to their work on identifying $\mathcal{A}^*$ and $\mathcal{W}^*$, Calamai and Moré propose a variant of the projected-gradient method: the negative projected gradient is only used as a search direction during iterations k, with

$$k \in K = \{k_0, k_1, k_2, \ldots\} \subseteq \{0, 1, 2, \ldots\},$$

for some K. When the projected-gradient method is not used to calculate x_{k+1}, x_{k+1} is chosen so that $x_{k+1} \in \mathbb{F}$ and $f(x_{k+1}) \leq f(x_k)$. One possible method for computing x_{k+1} is to perform an unconstrained minimization on f where the components x_i are fixed, with $i \in \mathcal{A}_k$ and project the result into $\mathbb{F}$.

Calamai and Moré prove that, if $|K| = \infty$, then

$$\lim_{i \to \infty} \|g_{F,k_i}\| = 0.$$

Additionally, if $\mathcal{A}_k \subseteq \mathcal{A}_{k+1}$ for all $k \in K^c$ and $\{x_k\}$ is bounded, then $\mathcal{A}^*$ is identified in a finite number of iterations.

Moré and Toraldo [MT91] also present a two-phase algorithm for box-constrained quadratic minimization that exploits the finite identification property. During the first phase, x_{k+1} is obtained from (3.6) with $p_k = -g_k$ and α_k taken

from an Armijo-like line search. If $\mathcal{A}_{k+1} = \mathcal{A}_k$ or the step fails to make reasonable progress in reducing f, the algorithm switches to phase two. Phase two iteratively solves for x_{k+1} using (3.6) with p_k defined as a conjugate-gradient direction (taken from the unconstrained optimization problem where the components of x corresponding to elements of $\mathcal{A}^c(x_k)$ are fixed) and α_k satisfies the Armijo-like condition. Phase two continues until a solution is found or until $\mathcal{W}_k \neq \mathcal{A}_k$. If $\mathcal{W}_k \neq \mathcal{A}_k$, the algorithm switches back to the first phase.

Moré and Toraldo prove that if f is a strictly convex, quadratic function, then the iterates $\{x_k\}$ generated by the algorithm converge to a minimizer x^*. If x^* is nondegenerate, then the algorithm terminates at x^* in a finite number of steps.

3.2.1 Algorithm L-BFGS-B

The most well-known and often-used algorithm to date that solves equation (3.2) when $\nabla^2 f(x)$ does not exist or is not practical to compute is L-BFGS-B [BLNZ95]. Algorithm L-BFGS-B is a limited-memory variation of a projected-gradient method. The algorithm is best explained by running through a typical iteration, which is detailed below.

Given $x_k \in \mathbb{F}$, perform an exact line search on $q_k(P(x_k - \alpha g_k))$, where $q_k(x)$ is the quadratic (2.2) (page 10) defined with the quasi-Newton approximate Hessian (2.4). Define x^c, called the *generalized Cauchy point*, to be the first local minimizer along the path $P(x_k - \alpha g_k)$. Although $q_k(P(x_k - \alpha g_k))$ is only piecewise differentiable with respect to α, it is piecewise quadratic, which allows L-BFGS-B to find x^c in $O(n)$ operations.

Next, compute the minimizer of $q_k(x)$ while keeping components of x fixed that correspond to elements of $\mathcal{A}(x^c)$. Byrd et al. outline three different ways to solve this system. One uses a conjugate-gradient method and the other two—a primal method and a dual method—store and update the inverse approximate limited-memory Hessian using the Sherman-Morrison-Woodbury formula. The default option is the primal method, which appears to be the most competitive of the three based on numerical results in Byrd et al. The numerical results in Chapter 6 use the default primal method.

After obtaining the minimizer in the previous step, L-BFGS-B backtracks toward x^c (if necessary) until it obtains a feasible point, which is labeled $\bar{x}_{k+1}$. Next, it forms the search direction $p_k = \bar{x}_{k+1} - x_k$ and performs a Wolfe line search along p_k to compute x_{k+1}. In order to ensure that x_{k+1} is a feasible point, $\alpha_{\max} = 1$ when performing the line search.

Although Algorithm L-BFGS-B is a very competitive algorithm, Byrd et al. do not give any convergence results.

3.3 Projected-search methods

For the sake of categorization, any method that repeatedly uses the negative gradient as a search direction during some portion of the method is called a projected-gradient method. On the other hand, a projected-search method calculates x_{k+1} from (3.6) but, in general, does not use $p_k = -g_k$.

The first practical projected-search method was proposed by Bertsekas [Ber76, Ber82]. All search directions calculated using a projected-search method can be written in the form

$$p_k = -D_k g_k, \tag{3.9}$$

where D_k is a positive-definite symmetric matrix that is diagonal with respect to a set of indices. (Recall that a symmetric matrix M of dimension n is said to be diagonal with respect to set $\mathcal{I}$ if $M_{ij} = 0$ for all $i \in \mathcal{I}$, $j \in \{1, 2, \ldots, n\}$, and $j \neq i$.) Berksekas establishes a number of important convergence results for projected-search methods; however, all original results are proved for the feasible region $\{x : x \geq 0\}$. To better preserve the flow of the chapter, Bertsekas' propositions are described for the more general case, $\mathbb{F} = \{x : l \leq x \leq u\}$.

To establish that projected-search methods are well-defined, Bertsekas proves the following proposition [Ber82].

Proposition 3.3.1. *Let $x \in \mathbb{F}$, and let D be a positive-definite symmetric matrix that is diagonal with respect to $\mathcal{W}(x)$. Define*

$$x(\alpha) = P(x + \alpha p) = P(x - \alpha D g(x)).$$

1. *The vector x is a stationary point if and only if*

$$x = x(\alpha) \ \textit{for all} \ \alpha \geq 0.$$

2. *If x is not a stationary point, there exists a scalar $\overline{\alpha} > 0$ such that*

$$f(x(\alpha)) < f(x) \ \textit{for all} \ \alpha \in (0, \overline{\alpha}]. \qquad \square$$

As $x = x(\alpha)$ for all $\alpha \geq 0$ if and only if $P_x(p) = 0$, the proposition guarantees that at every iteration, any projected-search algorithm will either terminate successfully or be able to find a path along which f decreases. It should be noted that the second condition is equivalent to guaranteeing that $P_x(p)$ is a descent direction if $P_x(p) \neq 0$. Bertsekas also proves that, under suitable conditions, all limit points x_* of $\{x_k\}$ generated by (3.6) and (3.9) are stationary points.

In order to discuss Bertsekas' other results, we define the *expanded working set at x*, which we denote as $\mathcal{E}(x)$ or $\mathcal{E}_k = \mathcal{E}(x_k)$, to be

$$\mathcal{E}(x) = \{i : l_i \leq x_i \leq l_i + \epsilon(x), g_i > 0\} \cup \{i : u_i - \epsilon(x) \leq x_i \leq u_i, g_i < 0\}, \quad (3.10)$$

where $\epsilon(x) > 0$ for all x. Bertsekas defines $\epsilon(x_k)$ to be

$$\epsilon(x_k) = \epsilon_k = \min\{\epsilon, \|P(x_k - Mg_k) - x_k\|_2\},$$

where $\epsilon > 0$ is a scalar constant and M is a constant positive-definite diagonal matrix (such as the identity matrix). In practice, all proposed algorithms implemented in Chapter 5 use the standard working set, $\mathcal{W}(x)$. We introduce expanded working sets solely because Bertsekas makes use of them in his convergence results.

Complementing the expanded working set $\mathcal{E}(x)$, Bertsekas also makes use of another Armijo-like line search that, given a current point x and direction p, seeks a scalar α such that

$$\psi(\alpha) \leq \psi(0) + c_A g(x)^T (P_{\mathcal{E}^c(x)}(p) + P_{\mathcal{E}(x)}(x(\alpha) - x))\alpha. \quad (3.11)$$

If $\mathcal{E}(x)$ is replaced by $\mathcal{W}(x)$, this Armijo-like condition simplifies to

$$\psi(\alpha) < \psi(0) + c_A \psi'(0)\alpha, \quad (3.12)$$

where $\psi'(0) = g(x)^T P_x(p)$. This simplified Armijo-like condition differs from (3.7) and (3.11) in that the right-hand side of (3.12) is linear with respect to α.

The following result, originally proved by Bertsekas [Ber82], is stated with stronger assumptions than is necessary for the sake of brevity.

Proposition 3.3.2. *Let x^* be a nondegenerate local minimizer of problem* (3.1) *such that, for some $\delta > 0$, f is twice continuously differentiable in the open neighborhood $S = \{x : \|x - x^*\| < \delta\}$ and the eigenvalues of $\nabla^2 f(x)$ are uniformly bounded above and away from zero for all $x \in S$. Let the eigenvalues of D_k be uniformly bounded above and away from zero for all $k \in \{0, 1, \ldots\}$. Additionally, assume that*

$$[D_k]_{ii} \geq \overline{\lambda} \text{ for all } k \in \{0, 1, \ldots\} \text{ and } i \in \mathcal{E}_k.$$

for some scalar $\overline{\lambda}$.

Then there exists a scalar $\overline{\delta} > 0$ such that if $\{x_k\}$ is a sequence generated by (3.6) *and* (3.9) *and for some index N,*

$$\|x_N - x^*\| \leq \overline{\delta},$$

then $\{x_k\}$ converges to x^* and

$$\mathcal{E}_k = \mathcal{A}_k = \mathcal{A}^*$$

for all $k > N$. □

The significance of this proposition is that, like many gradient-projection methods, given suitable conditions, a projected-search method identifies the active set at the solution after a finite number of iterations. As before, this means that the convergence rate of a projected-search method is the same as the corresponding unconstrained optimization method—in this case, determined by the choice of D_k.

If f is strictly convex, one such choice for D_k, Bertsekas [Ber82] observes, is to choose D_k so that D_k^{-1} is given elementwise as

$$[D_k^{-1}]_{i,j} = \begin{cases} 0 & \text{if } i \neq j \text{ and either } i \in \mathcal{E}_k \text{ or } j \in \mathcal{E}_k, \\ [\nabla^2 f(x)]_{i,j} & \text{otherwise.} \end{cases}$$

In this case, after the optimal active set is found, the algorithm is effectively the same as using Newton's method on the components of x that do not correspond to $\mathcal{A}^*$ while the other components are fixed. Given conditions similar to the first part of the previous proposition, it can be shown that $\{x_k\}$ converges to x^* and that the rate of convergence is superlinear—or quadratic if $g(x)$ is Lipschitz continuous in a neighborhood of x^*. Additionally, if f is quadratic, then problem (3.1) is solved in a finite number of iterations. Bertsekas also suggests a similar choice of D_k based on a quasi-Newton method.

Ni and Yuan [NY97] propose a quasi-Newton projected-search method that is similar in structure to Bertsekas' method [Ber82] but that uses a different definition of the expanded working set by choosing a different function $\epsilon(x)$. Additionally, the algorithm calculates p_k by using a reduced inverse approximate Hessian in place of solving (3.9)—see below for how this is done. The algorithm also differs from Bertsekas' in that it uses the Armijo-like condition (3.12), even though it does not use a standard working set. Ni and Yuan also prove that, given suitable conditions, every limit point of $\{x_k\}$ is a stationary point.

3.3.1 Solving the quasi-Newton equation

Using the working set

As mentioned above, it is possible to choose D_k so that, after a finite number of iterations, a projected-search method is essentially using an unconstrained quasi-Newton method on the components of x that are not in the expanded working set. To illustrate how this works in practice, for simplicity, we use the regular working set $\mathcal{W}(x)$ (and its complement, the free set, $\mathcal{F}(x)$) instead of $\mathcal{E}(x)$. To simplify notation, all iteration subscripts are suppressed for the remainder of this chapter. Given a feasible point $x \in \mathbb{R}^n$, assume without loss of generality that, if $|\mathcal{F}(x)| = |\mathcal{W}^c(x)| = w$, then the first w indices are in $\mathcal{F}(x)$. That is, let $\mathcal{W}(x) = \{n_w, n_w + 1, \ldots, n-1, n\}$, where $n_w = n - w + 1$.

Define $M = D^{-1}$ so that (3.9) is equivalent to solving

$$Mp = -g \tag{3.13}$$

for p. Matrices M and D can be expressed in block-diagonal form as

$$M = \begin{pmatrix} M_{\mathcal{F}} & \\ & M_{\mathcal{W}} \end{pmatrix} = \begin{pmatrix} M_{\mathcal{F}} & & & \\ & \mu_{n_w} & & \\ & & \ddots & \\ & & & \mu_n \end{pmatrix},$$

and

$$D = \begin{pmatrix} D_{\mathcal{F}} & \\ & D_{\mathcal{W}} \end{pmatrix} = \begin{pmatrix} D_{\mathcal{F}} & & & \\ & \delta_{n_w} & & \\ & & \ddots & \\ & & & \delta_n \end{pmatrix},$$

where $M_{\mathcal{F}}$ and $D_{\mathcal{F}}$ are $w \times w$ positive-definite matrices, μ_i, $\delta_i \in \mathbb{R}$ and $\mu_i > 0$, $\delta_i > 0$, $i \in \{n_w, \ldots, n\}$. Note that $M_{\mathcal{F}} = D_{\mathcal{F}}^{-1}$ and $\mu_i \delta_i = 1$ for $i \in \{n_w, \ldots, n\}$. The vectors p and g are partitioned in the same fashion so that

$$p = \begin{pmatrix} p_{\mathcal{F}} \\ p_{\mathcal{W}} \end{pmatrix} \quad \text{and} \quad g = \begin{pmatrix} g_{\mathcal{F}} \\ g_{\mathcal{W}} \end{pmatrix}.$$

Given the block-diagonal structure of M, it holds that

$$\begin{aligned} Mp = -g \iff & \begin{pmatrix} M_{\mathcal{F}} & \\ & M_{\mathcal{W}} \end{pmatrix} \begin{pmatrix} p_{\mathcal{F}} \\ p_{\mathcal{W}} \end{pmatrix} = \begin{pmatrix} -g_{\mathcal{F}} \\ -g_{\mathcal{W}} \end{pmatrix} \\ \iff & M_{\mathcal{F}} p_{\mathcal{F}} = -g_{\mathcal{F}} \quad \text{and} \quad M_{\mathcal{W}} p_{\mathcal{W}} = -g_{\mathcal{W}}. \end{aligned}$$

Since $M_{\mathcal{W}}$ is diagonal, p can be calculated by solving the smaller system

$$M_{\mathcal{F}} p_{\mathcal{F}} = -g_{\mathcal{F}}, \tag{3.14}$$

for $p_{\mathcal{F}}$ and setting

$$p_{\mathcal{W}} = -M_{\mathcal{W}}^{-1} g_{\mathcal{W}} = \begin{pmatrix} -g_{n_w}/\mu_{n_w} \\ \vdots \\ -g_n/\mu_n \end{pmatrix},$$

where g_i is the i^{th} component of g.

Since $\mu_i > 0$ for $i \in \{n_w, \ldots, n\}$, the signs of the components of $p_{\mathcal{W}}$ are the same as the signs of the components of $g_{\mathcal{W}}$. The definition of $\mathcal{W}$ implies that

the last $n - w$ components of $P(x + \alpha p)$ are the last $n - w$ components of x for all $\alpha > 0$. Put another way, the last $n - w$ components of $P_x(p)$ are zero. Since $\alpha_1 \geq \alpha_2 \geq 0$ implies $\mathcal{A}(x(\alpha_1)) \supseteq \mathcal{A}(x(\alpha_2))$, choosing p to be of the form

$$p = \begin{pmatrix} p_{\mathcal{F}} \\ 0 \end{pmatrix}, \tag{3.15}$$

yields the same result as choosing

$$p = \begin{pmatrix} p_{\mathcal{F}} \\ p_{\mathcal{W}} \end{pmatrix}.$$

Solving

$$Mp = -g_F$$

for p (recall $g_F = -P_x(-g)$) yields a search direction of the form (3.15).

In general, when $\mathcal{W}^c(x) \neq \{1, \ldots, w\}$, p is still calculated by solving a reduced system. Let Π be an $n \times w$ matrix whose columns are taken from the set $\{e_i : i \in \mathcal{F}(x)\}$ and where e_i is the i^{th} column of an identity matrix of order n. Then

$$M_{\mathcal{F}} = \Pi^T M \Pi, \quad p_{\mathcal{F}} = \Pi^T p, \quad g_{\mathcal{F}} = \Pi^T g = \Pi^T g_F, \tag{3.16}$$

where $p \in \text{col}(\Pi)$ is obtained from (3.14), so that

$$p = \Pi \Pi^T p = \Pi p_{\mathcal{F}}.$$

The right-hand sides of (3.16) are strictly formal: for example, in practice, the reduced matrix $\Pi^T M \Pi$ is obtained by "deleting" all the rows and columns of M that correspond to elements of $\mathcal{W}$. Similarly, p is easily formed by "scattering" the components of $p_{\mathcal{F}}$ into the components of a n-length zero vector that correspond to elements of $\mathcal{F}(x)$. For example, if $n = 5$, $\mathcal{F}(x) = \{1, 3, 4\}$, and $p_{\mathcal{F}} = \begin{pmatrix} 1 & 9 & 2.3 \end{pmatrix}$, then $p = \begin{pmatrix} 1 & 0 & 9 & 2.3 & 0 \end{pmatrix}$.

Using an expanded working set

To describe the case where an expanded working set is used, we use the same matrix and vector partitions as above, except that all occurrences of $\mathcal{W}$ are

now labeled $\mathcal{E}$. The free set $\mathcal{F}(x)$ refers to the complement of $\mathcal{E}(x)$ in the remainder of this subsection, and without loss of generality, for the sake of partitioning, it is assumed that $\mathcal{F}(x) = \{1, 2, \ldots, w\}$.

When using an expanded working set, $p_{\mathcal{E}}$ can be set to be something other than the zero vector, in which case,

$$p = \Pi p_{\mathcal{F}} + \Gamma p_{\mathcal{E}},$$

where Γ is an $n \times (n - w)$ matrix whose columns are taken from the set $\{e_i : i \in \mathcal{E}(x)\}$. For example, if $\mathcal{E}(x) = \{2, 5\}$, $p_{\mathcal{E}} = \begin{pmatrix} -.02 & .03 \end{pmatrix}$, and n, $\mathcal{F}(x)$, and $p_{\mathcal{F}}$ are as above, then $p = \begin{pmatrix} 1 & -.02 & 9 & 2.3 & .03 \end{pmatrix}$.

The diagonal elements of $M_{\mathcal{E}}$ can be chosen to be arbitrary positive real numbers. For example, matrix $M_{\mathcal{E}}$ can be chosen so that $x_i + p_i$ steps to either l_i or u_i, where $i \in \mathcal{E}$, or $M_{\mathcal{E}}$ can be chosen as the identity matrix of order $n - w$. In effect, p is still obtained by solving the smaller system involving $p_{\mathcal{F}}$ described above, but components of $p_{\mathcal{E}}$ are not necessarily zero. Importantly, if index $i \in \mathcal{E}(x)$, then it must be the case that $i \in \mathcal{E}(P(x + \alpha p))$ for all $\alpha > 0$, since $\text{sign}(p_i) = \text{sign}(-g_i)$.

Implications

Up to this point, the matrices M and $D = M^{-1}$ have been left unspecified. To create a quasi-Newton method for box-constrained optimization, we choose M so that

$$M_{\mathcal{F}} = \Pi^T M \Pi = \Pi^T H \Pi,$$

where H is an approximate Hessian matrix. Since the working set (or expanded working set) may change frequently, algorithms generally store H (or a limited-memory equivalent) and not the reduced approximate Hessian $\Pi^T H \Pi$.

Since p is chosen so that $\Pi \Pi^T p = P_x(p) = p$ and since

$$Hp = -g_F \implies \Pi^T H \Pi \Pi^T p = -\Pi^T g,$$

calculating a quasi-Newton step p using a projected-search method has the same effect as setting

$$p = \Pi q \text{ where } q \text{ solves } (\Pi^T H \Pi) q = -\Pi^T g. \tag{3.17}$$

Parentheses are not necessary here, but are used to suggest the key components of the equation. Geometrically, the search direction p obtained from (3.17) can be viewed as the solution to

$$\operatorname*{argmin}_{p \in \mathbb{P}_{\mathcal{E}}} q(x+p).$$

or

$$\operatorname*{argmin}_{p \in \mathbb{P}_{\mathcal{W}}} q(x+p).$$

where $q(x)$ is the quadratic model of f defined at x with Hessian H. In other words, $x+p$ minimizes the quadratic model $q(x)$ on the "face" defined by $\mathcal{E}$ (or $\mathcal{W}$).

3.3.2 Line searches for projected-search methods

One advantage that Algorithm L-BFGS-B has over current projected-search methods is its use of a Wolfe line search. If the Wolfe conditions are satisfied, then the current approximate Hessian can be updated with the BFGS update and remain positive definite. In contrast, if only the Armijo condition is satisfied, it may or may not be possible to update the approximate Hessian while staying positive definite, which can lead to poor convergence rates.

Because a projected-search method performs a line search along a piecewise linear path, the univariate function $\psi(\alpha) = f(x(\alpha))$ is only piecewise differentiable. Unfortunately, since a Wolfe line search requires a differentiable function, all projected-search methods currently use an Armijo-like line search.

Chapter 4

Line Searches on Piecewise-Differentiable Functions

Recall definition (2.6) (Chapter 2, page 13) that a step α is an *Armijo step* if it satisfies

$$\phi(\alpha) \leq \phi(0) + c_A \phi'(0)\alpha,$$

where $\phi(\alpha) = f(x + \alpha p)$ (2.5). Similarly, a *strong Wolfe step* (2.8) α is an Armijo step that satisfies

$$|\phi'(\alpha)| \leq c_W |\phi'(0)|.$$

Since projected-search methods perform a line search on the piecewise-differentiable function $\psi(\alpha) = f(x(\alpha)) = f(P(x + \alpha p))$, it is not possible for such methods to use a Wolfe line search. Instead, all projected-search methods to date use an Armijo or Armijo-like line search. Although these methods can add and drop indices in the working set rapidly without needing to reevaluate f, they can suffer from poor convergence rates when the search direction is obtained by using a quasi-Newton method. This is due to the fact that using an Armijo or Armijo-like step does not guarantee that the approximate Hessian can be updated while staying positive definite.

On the other hand, while L-BFGS-B employs a Wolfe line search, it is hindered by an artificial cap on the maximum step it can take in order to enforce feasibility. This means that if a Wolfe line search cannot find a Wolfe step in the

first interval, it cannot test successively larger intervals to find an appropriate step.

In this chapter, we define a new step type, called a *quasi-Wolfe step*, and a corresponding line search that is meant to keep advantages from both approaches listed above while minimizing their downsides. In Section 4.1, we provide a more detailed account of the theory behind a Wolfe line search. In Section 4.2, we discuss the theory involved with a quasi-Wolfe line search. Section 4.3 focuses on implementation issues and Section 4.4 presents an algorithm that uses the new line search.

4.1 Differentiable functions: the Wolfe step

The key principle that drives a Wolfe line search is that it is possible, when certain conditions are met, to know when an interval contains a Wolfe step. In addition to the propositions laid out below, which borrow from Morè and Thuente [MT94], more information can be found in Wolfe [Wol72] and Nocedal and Wright [NW99].

Recall, from Chapter 2, the proposition that drives the first stage of a Wolfe line search, which we now prove.

Proposition 2.3.1. *Let $\{\alpha_i\}$ be a strictly monotonically increasing sequence with $\alpha_0 = 0$, and let $\phi : \mathbb{R} \to \mathbb{R}$ be a continuously differentiable function. If it exists, let j be the smallest index where at least one of the following conditions (collectively called stage-one conditions) is true:*

1. *α_j is a Wolfe step, or*
2. *α_j is not an Armijo step, or*
3. *$\phi(\alpha_j) \geq \phi(\alpha_{j-1})$, or*
4. *$\phi'(\alpha_j) \geq 0$.*

If such a j exists, then there exists a Wolfe step $\alpha^* \in [\alpha_{j-1}, \alpha_j]$.

Proof. For convenience, given c_A, define the linear function used in the Armijo test to be

$$\theta(\alpha) = \phi(0) + c_A\phi'(0)\alpha,$$

and define

$$\omega(\alpha) = \phi(\alpha) - \theta(\alpha).$$

Then

$$\omega'(\alpha) = \phi'(\alpha) - c_A\phi'(0).$$

Observe that if $j > 1$, α_{j-1} satisfies none of the conditions (1)–(4), otherwise stage one would have terminated already. This implies that

$$\phi'(\alpha_{j-1}) < 0$$

by (4), and hence

$$\phi'(\alpha_{j-1}) < c_W\phi'(0)$$

by (1). If $j = 1$, $\phi'(\alpha_{j-1}) = \phi'(0) < c_W\phi'(0)$, thus

$$\phi'(\alpha_{j-1}) \leq c_W\phi'(0) \tag{4.1}$$

for all $j \geq 1$. Note that $\omega(\alpha_{j-1}) \leq 0$.

These introductory results are used in the following proofs for each of the four cases.

Case 1: If (1) is true, the proposition is true trivially.

Case 2: If (2) is true, then $\omega(\alpha_j) > 0$. Define

$$\alpha_m = \sup\{\alpha \in [\alpha_{j-1}, \alpha_j] : \omega(\alpha) \leq 0 \text{ for all } \beta \in [\alpha_{j-1}, \alpha]\}.$$

By inequality (4.1), $\omega'(\alpha_{j-1}) < 0$, thus $\alpha_{j-1} < \alpha_m < \alpha_j$ and $\omega(\alpha_m) = 0$. Since f is differentiable, $\omega(\alpha_{j-1}) \leq \omega(\alpha_m)$ and $\omega'(\alpha_{j-1}) < 0$, then, by the mean-value theorem, there exists $\alpha^* \in [\alpha_{j-1}, \alpha_m]$ such that $\omega'(\alpha^*) = 0$. Then

$$c_W\phi'(0) < c_A\phi'(0) = \phi'(\alpha^*) < 0,$$

and α^* is a Wolfe step.

Case 3: If (3) is true and (2) is false, then without loss of generality, assume $\phi(\alpha) \leq \theta(\alpha)$ for all $\alpha \in [\alpha_{j-1}, \alpha_j]$. (If, for some $\alpha \in [\alpha_{j-1}, \alpha_j]$, α is not an Armijo step, use the preceding argument with α in place of α_j.) Since $\phi(\alpha_{j-1}) \leq \phi(\alpha_j)$ and $\phi'(\alpha_{j-1}) < 0$, there exists a step $\alpha^* \in [\alpha_{j-1}, \alpha_j]$ such that $\phi'(\alpha^*) = 0$. Since $\omega(\alpha^*) \leq 0$, α^* is a Wolfe step.

Case 4: If (4) is true and (2) is false, assume without loss of generality that $\phi(\alpha) \leq \theta(\alpha)$ for all $\alpha \in [\alpha_{j-1}, \alpha_j]$, as in the previous case. Since $\phi'(\alpha_{j-1}) < 0$ and $\phi'(\alpha_j) \geq 0$, the continuity of ϕ' implies that there exists $\alpha^* \in [\alpha_{j-1}, \alpha_j]$ such that $\phi'(\alpha^*) = 0$. Since $\omega(\alpha^*) \leq 0$, α^* is a Wolfe step. □

The proof for Proposition 2.3.2 (page 18) uses exactly the same arguments. □

4.2 Piecewise-differentiable functions: the quasi-Wolfe step

Performing a line search on the univariate function

$$\psi(\alpha) = f(x(\alpha)) = f(P(x + \alpha p)),$$

instead of

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

is a substantially more difficult task, since ψ is only piecewise differentiable, with a finite number of jump discontinuities in the derivative. Since the proposition established in the previous section uses the mean-value theorem and requires the underlying function to be differentiable, it is not possible to guarantee a Wolfe step using said conditions. To compensate for the loss of differentiability, we introduce a new step type, defined below.

We define the right derivative of function ψ at α to be

$$\psi'_+(\alpha) = \lim_{\beta \to \alpha_+} \psi'(\beta),$$

and the left derivative of function ψ at α to be

$$\psi'_-(\alpha) = \lim_{\beta \to \alpha_-} \psi'(\beta).$$

The following lemmas, used in proposition below, are stated here without proof.

Lemma 4.2.1. *Let a, $b \in \mathbb{R}$ such that $0 \leq a < b$, and assume that f is a univariate, continuous, piecewise-differentiable function with a finite number of jump discontinuities in the derivative.*

$$f'_+(a) \leq 0 \ \text{ and } \ f(a) \leq f(b),$$

then there exists a point $x \in [a, b]$ such that

$$f'_-(x) \leq 0 \leq f'_+(x),$$

with equality throughout if $f'(x)$ exists. □

Lemma 4.2.2. *Let a, $b \in \mathbb{R}$ such that $0 \leq a < b$ and assume that f is a univariate, continuous, piecewise-differentiable function with a finite number of jump discontinuities in the derivative. If $f'_+(a) \leq 0$ and $f'_-(b) \geq 0$ then there exists a point $x \in [a, b]$ such that*

$$f'_-(x) \leq 0 \leq f'_+(x),$$

with equality throughout if $f'(x)$ exists. □

A step α is called a *quasi-Wolfe step* if it is an Armijo step and satisfies at least one of the following conditions:

1. $|\psi'_-(\alpha)| \leq c_W |\psi'_+(0)|$;
2. $|\psi'_+(\alpha)| \leq c_W |\psi'_+(0)|$;
3. $\psi'(\alpha)$ does not exist and $\psi'_-(\alpha) \leq 0 \leq \psi'_+(\alpha)$.

A quasi-weak-Wolfe step can be defined in a similar manner by modifying the first two conditions. We construct a new line search by using the framework from the differentiable case.

Proposition 4.2.1. *Let $\{\alpha_i\}$ be a strictly monotonically increasing sequence with $\alpha_0 = 0$, and let $\psi : \mathbb{R} \to \mathbb{R}$ be a continuous piecewise-differentiable function whose derivative has a finite number of jump discontinuities. If it exists, let j be the smallest index where at least one of the following "stage-one" conditions is true:*

1. *α_j is a quasi-Wolfe step, or*
2. *α_j is not an Armijo step, or*
3. *$\psi(\alpha_j) \geq \psi(\alpha_{j-1})$, or*
4. *$\psi'_-(\alpha_j) \geq 0$.*

If such a j exists, then there exists a quasi-Wolfe step $\alpha^* \in [\alpha_{j-1}, \alpha_j]$.

Proof. Given c_A, define

$$\theta(\alpha) = \psi(0) + c_A \psi'_+(0)\alpha,$$

and

$$\omega(\alpha) = \psi(\alpha) - \theta(\alpha).$$

Then

$$\omega'(\alpha) = \psi'(\alpha) - c_A \psi'(0).$$

Observe that if $j > 1$, α_{j-1} satisfies none of the conditions (1)–(4), otherwise stage one would have terminated already. This implies that

$$\psi'_-(\alpha_{j-1}) < 0$$

by (4). If $\psi'(\alpha_{j-1})$ exists, then $\psi'_+(\alpha_{j-1}) = \psi'_-(\alpha_{j-1}) < 0$ and

$$\psi'_+(\alpha_{j-1}) < c_W \psi'_+(0)$$

by (1). If $\psi'(\alpha_{j-1})$ does not exist, then (1) implies $\psi'_+(\alpha_{j-1}) < 0$, which, in turn, implies

$$\psi'_+(\alpha_{j-1}) < c_W \psi'_+(0),$$

which also follows from (1). If $j = 1$, $\psi'_+(\alpha_{j-1}) = \psi'_+(0) < c_W \psi'_+(0)$, thus

$$\psi'_+(\alpha_{j-1}) \leq c_W \psi'_+(0), \tag{4.2}$$

for all $j \geq 1$. As in the differentiable case, $\omega(\alpha_{j-1}) \leq 0$.

These introductory results are used in the following proofs for each of the four cases.

Case 1: The proposition is true trivially if (1) is true.

Case 2: If (2) is true, then $\omega(\alpha_j) > 0$. Define

$$\alpha_m = \sup\{\alpha \in [\alpha_{j-1}, \alpha_j] : \omega(\alpha) \leq 0 \text{ for all } \beta \in [\alpha_{j-1}, \alpha]\}.$$

By inequality (4.2), $\omega'_+(\alpha_{j-1}) < 0$, thus $\alpha_{j-1} < \alpha_m < \alpha_j$ and $\omega(\alpha_m) = 0$. Since $\omega(\alpha_{j-1}) \leq \omega(\alpha_m)$ and $\omega'_+(\alpha_{j-1}) < 0$, by Lemma 4.2.1, there exists $\alpha^* \in [\alpha_{j-1}, \alpha_m]$ such that

$$\omega'_-(\alpha^*) \leq 0 \leq \omega'_+(\alpha^*).$$

This implies that

$$\psi'_-(\alpha^*) \leq c_A \psi'_+(0) \leq \psi'_+(\alpha^*).$$

By the definition of α_m, α^* is an Armijo step. Observe that $\psi'_-(\alpha^*) < 0$. Therefore, if $\psi'_+(\alpha^*) \geq 0$, α^* must be a quasi-Wolfe step. On the other hand, if $\psi'_+(\alpha^*) < 0$, then

$$c_W \psi'_+(0) < c_A \psi'_+(0) \leq \psi'_+(\alpha^*) < 0,$$

and α^* is a quasi-Wolfe step.

Case 3: If (3) is true and (2) is false, then without loss of generality, assume $\psi(\alpha) \leq \theta(\alpha)$ for all $\alpha \in [\alpha_{j-1}, \alpha_j]$. Since $\psi(\alpha_{j-1}) \leq \psi(\alpha_j)$ and $\psi'_+(\alpha_{j-1}) \leq 0$, there exists an Armijo step $\alpha^* \in [\alpha_{j-1}, \alpha_j]$ such that

$$\psi'_-(\alpha^*) \leq 0 \leq \psi'_+(\alpha^*),$$

by Lemma 4.2.1. If $\psi'_-(\alpha^*) = 0$ or $\psi'_+(\alpha^*) = 0$, α^* is a quasi-Wolfe step. Otherwise, $\psi'_-(\alpha^*) < 0 < \psi'_+(\alpha^*)$, which also implies that α^* is a quasi-Wolfe step.

Case 4: Finally, consider the case where (4) is true and (2) is false. Assume without loss of generality that $\psi(\alpha) \leq \theta(\alpha)$ for all $\alpha \in [\alpha_{j-1}, \alpha_j]$. By Lemma 4.2.2, there exists an Armijo step $\alpha^* \in [\alpha_{j-1}, \alpha_j]$ such that

$$\psi'_-(\alpha^*) \leq 0 \leq \psi'_+(\alpha^*).$$

Thus, by the exact same argument in the preceding paragraph, α^* is a quasi-Wolfe step. □

Proposition 4.2.2. *Let $\mathcal{I}$ be an interval whose endpoints are α_l and α_u and let $\psi : \mathbb{R} \to \mathbb{R}$ be a continuous, piecewise-differentiable function whose derivative has a finite number of jump discontinuities. Assume α_l and α_u ($\alpha_l \neq \alpha_u$) satisfy the following conditions:*

1. *α_l is an Armijo step, and*
2. *$\psi(\alpha_l) \leq \psi(\alpha_u)$ if α_u is an Armijo step, and*
3. *$\psi'_+(\alpha_l) < 0$ if $\alpha_l < \alpha_u$ or $\psi'_-(\alpha_l) > 0$ if $\alpha_l > \alpha_u$.*

Then there exists a quasi-Wolfe step $\alpha^* \in \mathcal{I}$.

Proof. The proof is similar to that of Proposition 4.2.1. □

A simplified example of a quasi-Wolfe line search is almost identical to the one presented in Chapter 2.

Algorithm 4.1 Simplified quasi-Wolfe line search

choose $c_A \in (0,1)$, $c_W \in (c_A,1)$, $\alpha_{\max}$
$\alpha \leftarrow 1$, $\alpha_{\text{old}} \leftarrow 0$
while α is not a quasi-Wolfe step
    if α is not an Armijo step or $\psi(\alpha) \geq \psi(\alpha_{\text{old}})$
        $\alpha \leftarrow$ StageTwo(α_{old},α); break
    if $\psi'_+(\alpha) \geq 0$
        $\alpha \leftarrow$ StageTwo(α,α_{old}); break
    $\alpha_{\text{old}} \leftarrow \alpha$
    increase α towards $\alpha_{\max}$
end
return α

function StageTwo(α_l,α_u)
    choose α_{new} in interval defined by α_l and α_u using interpolation
    if α_{new} is a quasi-Wolfe step
        return α_{new}
    if α_{new} is not an Armijo step or $\psi(\alpha_{\text{new}}) \geq \psi(\alpha_l)$
        return StageTwo(α_l,α_{new})
    if $\psi'_+(\alpha_{\text{new}})(\alpha_u - \alpha_l) < 0$
        return StageTwo(α_{new},α_u)
    otherwise
        return StageTwo(α_{new},α_l)
end function

4.2.1 Convergence results

Speaking in terms of exact arithmetic, if a quasi-Wolfe line search terminates prematurely during the first stage by reaching $\alpha_{\max}$ without locating an interval that contains a quasi-Wolfe step, all of the steps computed so far are Armijo steps. Also, if all the components of u and l are finite, then there exists a step, α_J, such that $\psi'(\alpha) = 0$, for all $\alpha \geq \alpha_J$.

A quasi-Wolfe line search seems to perform better in practice than a Armijo-like line search when used with a quasi-Newton method (see Chapter 6). However, in theory, it has a downside when compared to a Wolfe line search. If the next iterate is given by

$$\bar{x} = P(x + \bar{\alpha}p),$$

where $\bar{\alpha}$ is a quasi-Wolfe step, then the approximate curvature

$$(g(\bar{x}) - g(x))^T(\bar{x} - x)$$

need not be greater than zero. It is worth pointing out that this downside is only possible if the path $P(x + \alpha p)$ changes direction for some $\alpha \in (0, \bar{\alpha})$. If it does change direction, $\psi'_+(0)$ and $\psi'_-(\bar{\alpha})$ can be directional derivatives of f in a direction other than $\bar{x} - x$. For example, using Figure 4.1, which has lower bounds $x_1 = 0$ and $x_2 = 0$, $\psi'_+(0)$ is a directional derivative of f in direction p_1 and $\psi'_-(\bar{\alpha})$ is a directional derivative of f in direction p_2.

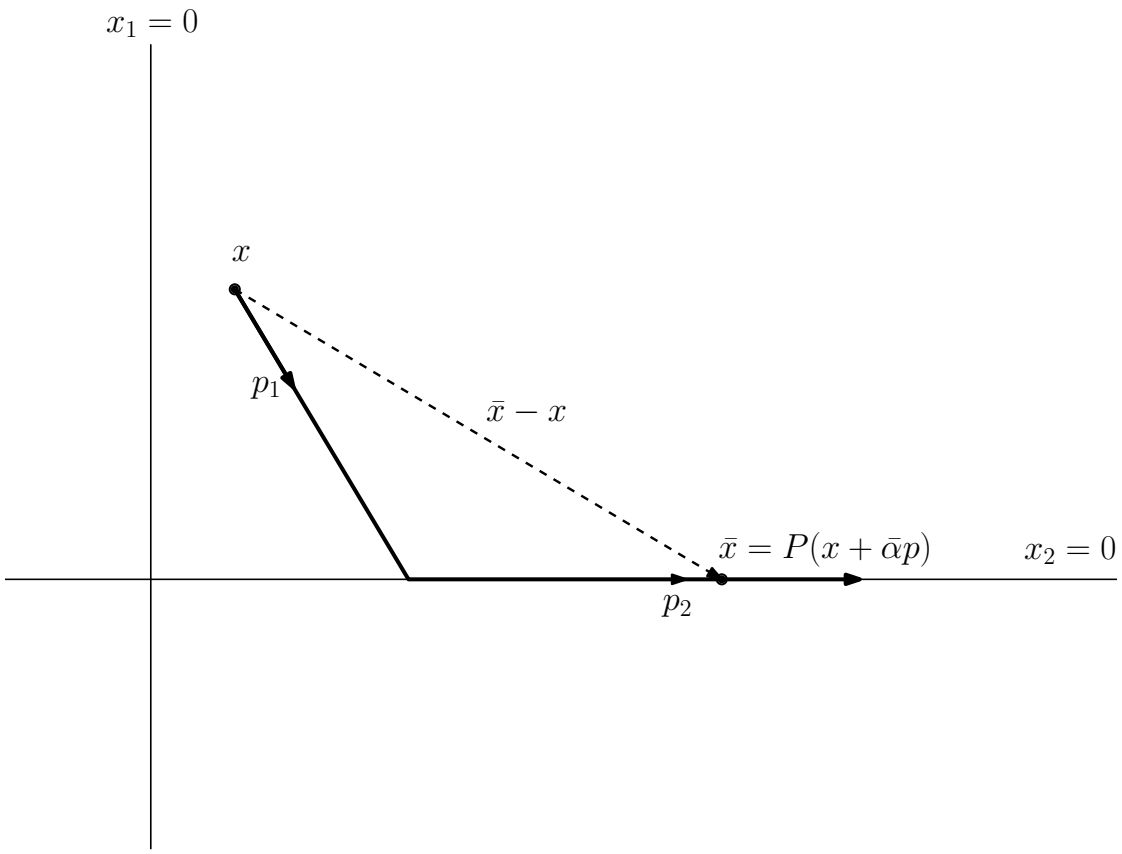

Figure 4.1: No guarantee to update approximate Hessian

As a result, if the path changes direction for $\alpha \in (0, \bar{\alpha})$, then it suffers from the same theoretical downside that an Armijo line search suffers from: there is no guarantee that the approximate Hessian can be updated. It is worth pointing out that this downside only occurs if the

In practice, it seems to be uncommon for an algorithm using a quasi-Wolfe line search to skip the approximate Hessian update. Using an implicit MATLAB implementation of Algorithm LRHB (5.2) (see Chapter 5), which utilizes a quasi-Wolfe line search, on a set of 111 problems from the CUTEr test set (see Chapter 6), the update to the approximate Hessian was skipped 183 times out of a total of 32,337 iterations. Out of those 183 times, the line search successfully calculated a quasi-Wolfe step 152 times (85 occurrences came from one problem and 31 came from another, both problems for which LRHB did not converge). On 19 problems, the line search successfully found a quasi-Wolfe interval but failed to find a step

within a user-defined number of iterations. On 9 others, a quasi-Wolfe step was not found due to $\alpha_{\max}$ being too large. On the remaining 3, no useful step was found.

If it can be shown that an algorithm using a quasi-Wolfe line search correctly identifies the active set at the solution in a finite number of iterations, then, after the active set stabilizes, a quasi-Wolfe line search behaves exactly like a Wolfe line search in that updates to the approximate Hessian are guaranteed. Under such circumstances, and given suitable conditions, the convergence rate of a quasi-Newton method using a quasi-Wolfe line search is Q-superlinear.

4.3 Practical considerations

There are two main issues that separate a quasi-Wolfe line search from a Wolfe line search. Apart from these two, which are discussed below, the code used in a Wolfe line search is almost identical to a quasi-Wolfe line search.

The first difference between a Wolfe and quasi-Wolfe line search concerns how each calculate $\psi'(\alpha)$ (or $\phi'(\alpha)$). Recall that, for the differentiable case,

$$\phi'(\alpha) = (f(x + \alpha p))' = g(x + \alpha p)^T p.$$

In other words, the slope of ϕ is related to the directional derivative of f at $x + \alpha p$ in direction p. On the other hand, because $\psi(\alpha) = f(P(x + \alpha p))$,

$$\psi'_+(\alpha) = f'(P(x + \alpha p))^T P_{x+\alpha p}(p) = g(x(\alpha))^T P_{x(\alpha)}(p).$$

Thus, the slope of ψ going forward from α is related to the directional derivative of f at $x(\alpha)$ in the projected direction $P_{x(\alpha)}(p)$.

A step α is called a *cusp step* if, for some component i,

$$(x_i + \alpha p_i = l_i \ \text{ and } \ p_i < 0) \ \text{ or } \ (x_i + \alpha p_i = u_i \ \text{ and } \ p_i > 0).$$

Note that, if $\psi'(\alpha)$ does not exist, then α is a cusp step. A step α is called a *cusp step with respect to* i if component i satisfies the condition above.

If α is a cusp step, it is almost always the case that $\psi'_-(\alpha) \neq \psi'_+(\alpha)$. In order to compute $\psi'_-(\alpha)$, the algorithm must compute the projected direction that

the piecewise linear path $x(\alpha)$ follows when approaching α from below. If this direction is denoted by $P^-_{x(\alpha)}(p)$, then

$$P^-_{x(\alpha)}(p) = \lim_{\beta \to \alpha^-} P_{x(\beta)}(p).$$

The projected vector $P^-_{x(\alpha)}(p)$ can be computed componentwise as

$$[P^-_{x(\alpha)}(p)]_i = \begin{cases} p_i & \text{if } \alpha \text{ is a cusp step with respect to } i, \\ [P_{x(\alpha)}(p)]_i & \text{otherwise.} \end{cases}$$

Therefore,

$$\psi'_-(\alpha) = g(x(\alpha))^T P^-_{x(\alpha)}(p).$$

Given α and β, where $0 \leq \alpha \leq \beta$, it is straightforward to calculate the number of cusp steps that occur between them. An upper bound on the number of cusp steps is

$$|\mathcal{A}(x(\beta))| - |\mathcal{A}(x(\alpha))|,$$

with equality if no step is a cusp step with respect to more than one index.

If $c \in \mathbb{R}^n$ is a vector defined componentwise as,

$$c_i = \begin{cases} \dfrac{u_i - x_i}{p_i} & \text{if } p_i > 0, \\ \dfrac{l_i - x_i}{p_i} & \text{if } p_i < 0, \\ \infty & \text{if } p_i = 0, \end{cases}$$

then α is a cusp step with respect to i if $\alpha = c_i$ and the number of cusp steps in the interval (α, β) is equal to the cardinality of the set

$$\{\alpha \in (\alpha, \beta) : \alpha = c_i \text{ for some } i = \{1, \ldots, n\}\}.$$

The second major difference between a Wolfe and quasi-Wolfe line search concerns the issue of interpolating to find new steps in the second stage. One approach, which makes no changes to the interpolation code in moving from a Wolfe line search to a quasi-Wolfe one, effectively treats the function at a cusp step as if the function was differentiable but "merely" highly nonlinear.

Since the function is only piecewise differentiable, methods used to interpolate points on a differentiable function may not be ideal. Another approach is to use knowledge of the number of cusp steps between two points. As a simple example, if there is exactly one cusp step between the two points, pick α_{new} to be that cusp step. As the number of cusp steps in an interval increases, it becomes more difficult to strike a balance between making effective use of the knowledge they exist and efficiency; for example, if an interval contains 10^6 cusp steps, it is not practical to jump to the middle one and repeat on each subinterval. Another strategy for dealing with piecewise differentiable functions is to interpolate without using first-derivative information.

4.3.1 Current implementation

The quasi-Wolfe line search algorithms as they are implemented in MATLAB and f90 are all simplistic insofar as they ignore cusps, i.e., no changes are made to the interpolation code. As a consequence, for the sake of computational efficiency and due to the complexity of the underlying line-search code, $\psi'_-(\alpha)$ is substituted with $\psi'_+(\alpha)$ in all relevant locations. The quasi-Wolfe line search used in all new methods proposed in Chapter 5 uses the line-search code written by Gill, Murray and Saunders [GMS97] as a foundation.

It is theoretically possible for the quasi-Wolfe line search, as implemented, to fail to identify a quasi-Wolfe step or an interval that contains one. This is only possible if one of the steps computed during the first or second stage functions is a cusp step. For reference, using the same numerical tests found in Section 4.2.1, out of the 37,373 times the quasi-Wolfe line search evaluates the objective function, only two of steps computed are cusp steps.

If an alternative implementation made changes to the interpolation code to actively seek cusp steps, it is likely that such an implementation would also benefit from computing and using $\psi'_-(\alpha)$, even at the cost of a small amount of additional overhead.

4.4 A variant of Algorithm L-BFGS-B

Although a quasi-Wolfe line search is used in all the algorithms proposed in Chapter 5, testing the resulting algorithms against Algorithm L-BFGS-B does not adequately demonstrate the strengths or weaknesses of a quasi-Wolfe line search algorithm compared to the line search strategy used in L-BFGS-B. This is because the underlying line-search codes are from two different sources. In order to make a fair comparison, we modify Algorithm L-BFGS-B to use a quasi-Wolfe line search based on existing line-search code in L-BFGS-B.

Recall that $q_k(x)$ is the quadratic model used by a quasi-Newton method during iteration k to model f. A typical iteration of the new algorithm, which we call Algorithm LBFGSB-M (for modified), takes the form:

1. Set x_c to be the first minimizer of $q_k(P(x_k - \alpha g_k))$.
2. Set x'_{k+1} to be the minimizer of $q_k(x)$ such that $[x'_{k+1}]_i = [x_c]_i$ for all $i \in \mathcal{A}(x_c)$.
3. Backtrack from x'_{k+1} toward x_c to obtain a feasible point, $\bar{x}_{k+1}$.
4. Form search direction $p_k = \bar{x}_{k+1} - x_k$.
5. Perform a quasi-Wolfe line search along p_k starting at x_k ($\alpha_{\max} > 1$).

The only difference between Algorithm L-BFGS-B and LBFGSB-M is in step five. For a numerical comparison between L-BFGS-B and LBFGSB-M, see Chapter 6.

It is possible to modify L-BFGS-B further and turn it into a pure projected-search method by using the working set instead of the active set and eliminating all steps involving x_c. For the purposes of this dissertation, this is a relatively low-priority task, since Algorithm LBFGSB-M already serves the purpose of adequately testing the effectiveness of using a quasi-Wolfe line search.

Chapter 5

Reduced-Hessian Methods for Box-Constrained Optimization

Unless otherwise noted, iteration subscripts are suppressed for the entirety of this chapter. Bars above symbols are used to denote fully updated quantities with respect to the current iteration and subscripts are used to refer to quantities that are only partially updated with respect to the current iteration. Many definitions found in Chapter 3 (see page 36 and following) are used extensively in this chapter.

Given x, recall from Chapter 3 that the formal way to solve a projected-search quasi-Newton equation is to solve

$$\Pi^T H \Pi q = -\Pi^T g \tag{5.1}$$

for q, and to set $p = \Pi q$. In this case, Π is a matrix whose orthonormal columns span the set of projected directions with respect to $\mathcal{W}(x)$. Since the feasible region is defined by $\mathbb{F} = \{x : l \leq x \leq u\}$, the columns of Π can be taken as the columns of the identity matrix of order n that correspond to elements of $\mathcal{W}^c(x)$.

Throughout this chapter, it is assumed that the working set is used (instead of an expanded working set) unless otherwise noted. Thus, $\mathcal{F}(x)$ is defined to be the set complement of $\mathcal{W}(x)$. Any of the algorithms in this chapter can be modified to use an expanded working set, with only minimal changes needed.

Recalling that $\mathbb{P}_{\mathcal{W}(x)}$ is defined to be the set of all feasible directions with

respect to $\mathcal{W}(x)$, the search direction p obtained above is the unique solution to

$$\underset{p'}{\text{argmin}} \; g^T p' + \frac{1}{2} {p'}^T H p' \;\; \text{such that} \;\; p' \in \mathbb{P}_{\mathcal{W}(x)}.$$

In other words, $x + p$ is the unique minimizer of the quadratic model subject to the constraint $p = P_{\mathcal{W}(x)}(p)$. It is important to note that $x+p$ may not be feasible. If $\mathcal{A}(x) = \mathcal{W}(x)$, the requirement on p is equivalent to saying that p must be a feasible direction.

Further, we recall from Chapter 2 that a reduced-Hessian search direction p is obtained by solving

$$Z'^T H Z' q = -Z'^T g \tag{5.2}$$

for q, and setting $p = Z'q$. In this case, Z' is a matrix whose orthonormal columns span the gradient subspace $\mathcal{G}$ (or, in the case of the limited-memory variants, some subspace of $\mathcal{G}$). To simplify notation, all references to Z, B, and T in Chapter 2 are labeled Z', B', and T', respectively, in Chapter 5. The matrix labels Z, B, and T are reserved for the equivalent projected variants, which are defined below.

Given the identical structure in (5.1) and (5.2), it seems natural to merge the two ideas together, i.e., to create a reduced-Hessian method that is also a projected-search method. This is achieved by solving

$$Z^T H Z q = -Z^T g \tag{5.3}$$

for q, and setting $p = Zq$, where Z can be formally defined as the orthonormal component of the QR decomposition of $\Pi\Pi^T Z'$. Thus, the column space of Z is the *projected* gradient space (or subspace) with respect to $\mathcal{W}(x)$, i.e.,

$$\text{col}(Z) = \{\Pi\Pi^T p : p \in \mathcal{G}\}.$$

As was briefly mentioned in Chapter 3, in practice there is never a need to form matrix products involving Π or Π^T. Given $x \in \mathbb{R}^n$, a matrix or vector M with n rows, and Π (defined by $\mathcal{W}(x)$), then the reduced matrix $\Pi^T M$ is formed from the rows of M that correspond to elements of $\mathcal{F}(x)$. Similarly, if N is a matrix or vector with $w = |\mathcal{F}(x)|$ rows, the matrix ΠN is a matrix whose k nonzero rows

are taken from N and correspond to elements of $\mathcal{F}(x)$. Finally, $\Pi\Pi^T M$ is a matrix whose rows are such that, for $i \in \{1, 2, \ldots, n\}$,

$$e_i^T \Pi\Pi^T M = \begin{cases} e_i^T M & \text{if } i \in \mathcal{F}(x) \\ 0 & \text{if } i \notin \mathcal{F}(x). \end{cases}$$

In other words, $\Pi\Pi^T M$ is formed by taking M and "zeroing out" the rows that correspond to elements of $\mathcal{W}(x)$.

The formal definition of Z suggests that there are two triangular factors to maintain: one from the QR decomposition of Z' and the other from Z. In practice, only one factor is needed, which is updated when Z' or Π changes. Given B', a basis matrix whose columns are taken to be search directions or gradients, we define the projected basis to be $B = \Pi\Pi^T B'$ and define Z and T with the QR decomposition $B = ZT$.

There are several difficulties that must be addressed in creating a practical algorithm that implements Z as discussed above. In particular, we must:

- consider how to modify B (or Z if using an explicit method), T, and any quantity that is defined in terms of B or Z when the working set changes; and
- address the potential loss of rank when "zeroing out" rows in B and Z.

There are also considerations to be made when modifying a reduced-Hessian algorithm to be a projected-search method. We need to:

- project all points into the feasible region;
- use projected gradients instead of gradients in most places;
- use a line search that is capable of handling problems with box constraints; and
- address how projected-search directions impact the BFGS update when applied to the Cholesky factor of the reduced Hessian $\bar{Z}^T H \bar{Z}$.

When the working set changes and the matrix $\bar{\Pi}$ is different from Π, it is desirable to update Z (either implicitly by updating B and T or explicitly) instead of re-orthogonalizing it from scratch after zeroing or "un-zeroing" rows. The procedure describing how to "un-zero" a row is described fully in Section 5.3. Additionally, to avoid computing Z from scratch, T must be stored, even if an explicit method is used. Due to the need to store T, all RH-B methods (except methods discussed in the next section) are built on a limited-memory foundation.

Since reinitialization is extremely effective on problems where n is large, all algorithms implement it by default when $n > \min(6, m)$, where m is the maximum number of columns stored in B. For a full-memory implementation, reinitialization is enabled when $n > 6$. In our algorithm naming convention, the postfix "R" (for reinitialization) is used if reinitialization is enabled for all n, the postfix "r" is used if reinitialization is never used. The absence of both letters signifies the default behavior described above.

To describe the projected-search variants of algorithms from Chapter 2, the suffix "-B" or just "B" (for box constraints) is appended to the end of the algorithm name. For example, the projected-search equivalent of Algorithm L-RHR is Algorithm L-RHR-B (or LRHRB). Similarly, the projected-search equivalent of Algorithm RHL is Algorithm RHrL-B.

Section 5.1 describes some of the obstacles that must be overcome to transform any reduced-Hessian method into a projected-search method. Section 5.2 outlines a simple but naïve full-memory implementation that stores only Z. From there, Section 5.3 details the linear algebra necessary to update relevant matrices when the working set changes. Section 5.4 presents several variations on the basic reduced-Hessian projected-search method. Section 5.5 discusses convergence results and Section 5.6 briefly discusses future work to be done.

5.1 Converting to a projected-search method

All projected-search reduced-Hessian methods share common functionality and structure. Therefore, many of the modifications that are required to allow an

RH algorithm to handle box constraints are common to all new projected-search reduced-Hessian methods. From the previous section, we recall that these universal modifications include:

- projecting all points into the feasible region;
- using the projected gradient instead of the gradient in most places;
- using a line search capable of handling problems with box constraints; and
- accounting for the inability to install curvature for any direction outside of $\text{col}(\bar{Z})$.

Regarding the first point, the initial iterate x_0 is assumed to be feasible. If it is not, $P(x_0)$ is used in its place, provided $l \leq u$. If $l_i > u_i$ for some component i, the algorithm is terminated with an appropriate error indication.

Since the columns of Z should span the projected gradient space (or subspace) instead of the full gradient space, the projected gradient $g_F = \Pi\Pi^T g$ is used in place of the full gradient almost everywhere. The full gradient is used to compute the projected gradient and is added to B' to be potentially used in "un-zeroing" rows during a later iteration if the projected gradient is not rejected.

Because of its role in projected-search methods, any line search used must traverse the path $x(\alpha) = P(x + \alpha p)$. Since $\psi(\alpha) = f(x(\alpha))$ is only piecewise differentiable, a regular Wolfe line search cannot be used to find the next iterate. Instead of using Armijo-like line search, a quasi-Wolfe line search is employed. A detailed description of a quasi-Wolfe line search is given in Chapter 4.

One difficulty in creating a projected-search RH method is discussed in Section 2.4.1, which observes that the BFGS update to R_1, the Cholesky factor of $\bar{Z}^T H \bar{Z}$, is well-defined only if $\delta \in \text{col}(\bar{Z})$, where $\delta = \bar{x} - x$. Unfortunately, it can often be the case that $\delta \notin \text{col}(\bar{Z})$ when using a projected-search method, since it is not necessary for δ to be a scalar multiple of p, nor is it necessary that $p \in \text{col}(\bar{Z})$ if $\mathcal{W}(x) \neq \mathcal{W}(\bar{x})$. One solution to this problem, and the one used here, is to install curvature information to H in a direction that is in the column space of $\bar{Z}$.

To discuss this solution, several terms must be defined. As in Section 2.4.1, we define

$$\delta = \bar{x} - x, \qquad \gamma = \bar{g} - g, \qquad s = \bar{Z}^T\delta, \qquad y = \bar{Z}^T\gamma, \quad \text{and} \quad R_1^T R_1 = \bar{Z}^T H \bar{Z}.$$

Note that $\bar{Z}$ refers to the matrix whose columns span the projected gradient subspace, which is not the same matrix used in Chapter 2. Instead of installing curvature in direction δ, we install curvature in direction δ_1, where

$$\delta_1 = x_1 - x \;\text{ and }\; x_1 = x + \bar{Z}\bar{Z}^T(\alpha p).$$

For computational efficiency, $g(x_1)$ is not computed. Instead, g_1 is defined to be the gradient of the quadratic model

$$q_{\bar{x}}(z) = f(\bar{x}) + \bar{g}^T(z - \bar{x}) + \frac{1}{2}(z - \bar{x})^T H (z - \bar{x}),$$

at x_1. Thus,

$$g_1 = \nabla q_{\bar{x}}(x_1) = \bar{g} + H(\bar{Z}\bar{Z}^T\alpha p - \delta).$$

Given g_1, the following quantities are also defined.

$$\begin{aligned}
\gamma_1 &= g_1 - g = \bar{g} - g + H(\bar{Z}\bar{Z}^T\alpha p - \delta) = \gamma + H\delta_1 - H\delta, \\
s_1 &= \bar{Z}^T\delta_1 = \bar{Z}^T\bar{Z}\bar{Z}^T(\alpha p) = \bar{Z}^T(\alpha p), \\
y_1 &= \bar{Z}^T\gamma_1 = \bar{Z}^T\gamma + \bar{Z}^T H(\bar{Z}\bar{Z}^T\alpha p - \delta) = y + R_1^T R_1(s_1 - s).
\end{aligned}$$

The last equality in the last definition follows from the fact that

$$\bar{Z}^T H\delta = \bar{Z}^T\bar{Z}R_1^T R_1\bar{Z}^T\delta + \sigma\bar{Z}^T\delta - \sigma\bar{Z}^T\bar{Z}\bar{Z}^T\delta = R_1^T R_1 s,$$

by implicit definition of H (2.11) (page 23). If $x_1 = \bar{x}$, then $g_1 = \bar{g}$, $s_1 = s$, and $y_1 = y$, as required.

Given y_1 and s_1, the BFGS update applied to R_1 is

$$R_2 = R_1 + w_1 w_2^T,$$

where

$$w_1 = \frac{1}{\|R_1 s_1\|} R_1 s_1 \;\text{ and }\; w_2 = \frac{1}{\sqrt{y_1^T s_1}} y_1 - R_1^T w_1.$$

Since R_2 is not upper triangular, R_3 is defined to be the triangular matrix associated with the QR factorization of R_2. If reinitialization is implemented (see Section 2.4.2), $\bar{R}$ is a $r \times r$ matrix such that,

$$\bar{R}_{ij} = \begin{cases} [R_2]_{ij} & \text{if } i \neq r \text{ or } j \neq r \\ \sqrt{\bar{\sigma}} & \text{if } i = j = r, \end{cases}$$

for some $\bar{\sigma}$. If reinitialization is not implemented, $\bar{R} = R_2$.

In order to highlight the similarity of a projected-search RH method and a standard RH method in the pseudo-code, s_1 and y_1 are referred to as s and y. In the initial assignment, the quantity αq is exactly s_1 and the quantity $u - v$ is exactly y, not y_1.

5.2 Simple implementation of a projected-search RH algorithm

One simple though inefficient way to avoid the difficulties inherent in updating the various matrices when the working set changes is to "restart" the RH algorithm every time the working set changes. Restarting a RH algorithm is equivalent to resetting all the relevant matrices and vectors (i.e., Z, R, w, etc) to whatever values they would be if we were to call the RH algorithm anew starting with input x. In practice it is not necessarily to call the algorithm again.

The principle for this algorithm rests on two observations. First, the initial search direction obtained after a restart will always be a positive scalar multiple of the direction obtained from the steepest descent method (with equality if $\sigma = 1$). Second, we recall from Chapter 3 that a gradient-projection method identifies the active set at a solution x^* in a finite number of iterations under suitable conditions.

The practical consequence is that such an algorithm will behave at worst like a gradient-projection method while constraints are being added and dropped, but will retain a Q-superlinear convergence rate after a finite number of iterations. Geometrically, such an algorithm can be thought of as accumulating curvature information on a subspace of the projected gradient space until the working set

changes, at which point it throws away any information obtained so far and begins the process again starting from the new iterate.

In the pseudo-code and the actual implementations, matrices can have either zero columns or rows. Matrix multiplication and addition is well-defined provided that the dimensions of the operands match appropriately. It is important to note that a gradient is always accepted if the working set has changed.

Algorithm 5.1 Algorithm RHSB (RH-Simple-B)

Choose $\sigma > 0$ and x

$x \leftarrow P(x)$

$g \leftarrow \nabla f(x)$

$g_F \leftarrow P_x(g)$

$Z \leftarrow g_F/\|g_F\|$, $R \leftarrow \sqrt{\sigma}$, $v \leftarrow \|g_F\|$

while not converged

    Solve $R^T d = -v$ for d

    Solve $Rq = d$ for q

    $p \leftarrow Zq$

    Compute quasi-Wolfe step α and projected direction p'

    $x_{\text{old}} \leftarrow x$, $x \leftarrow x + \alpha p'$, $g \leftarrow \nabla f(x)$

    $g_F \leftarrow P_x(g)$

    if $\mathcal{W}(x) \neq \mathcal{W}(x_{\text{old}})$

        $Z \leftarrow n \times 0$ dimension empty matrix

        $R \leftarrow 0 \times 0$ dimension empty matrix

        v, $q \leftarrow 0$ dimension empty vectors

    end

    $w \leftarrow Z^T g_F$

    Update Z, R, w, v, q based on whether gradient is accepted

    $s \leftarrow \alpha q$, $y \leftarrow u - v$

    if $x - x_{\text{old}} \notin \text{col}(Z)$

        $y \leftarrow y + R^T R(Z^T(x_{\text{old}} - x) + s)$

    end

    Apply BFGS update to R if $y^T s > 0$

    if $n > 6$

        Compute new σ for reinitialization

        Replace last diagonal element of R with $\sqrt{\sigma}$

    end

    $v \leftarrow w$

end

In Chapter 6, Algorithm RHSL-B is implemented, that is, the lingering version of the above algorithm. Algorithm RHSB is presented above instead of RHSLB for the sake of simplicity. A brief summary of lingering can be found in Section 2.4.3. A more detailed description of lingering for the unconstrained case can be found in Gill and Leonard [GL01].

5.3 Updating the working set

The constraints in the working set can change from one iteration to the next; and one of the strengths of projected-search methods is the ability to add and drop many constraints between function evaluations. Changes to the working set alter the number of nonzero rows in B. As the quantities Z, T, R, q, and v all depend on B they must also be updated. Other matrices depend on B but are computed after B is updated, and hence do not need to be updated. Although T is not used in full-memory versions of RH methods for unconstrained minimization, it must be used in the constrained case in order to properly update all other values. For this reason, with the exception of the simple variant in the previous section, all RH-B methods are derived from the limited-memory family of RH algorithms, such as Algorithm L-RHR.

As with Algorithm L-RHR, all of the algorithms in this chapter can be implemented using either an explicit or implicit method. An implicit method, which stores B but not Z, is most often useful when $m \lesssim 6$. It is generally the faster of the two methods, as it avoids computing expensive updates to Z. However, it suffers when m is larger due to its inability to reorthogonalize gradients that are implicitly added to Z. Conversely, an explicit method stores Z but not B. As Z is updated much more frequently in the constrained case, an explicit method often exhibits much slower performance than an implicit method. The exception to this is if function calls are expensive relative to the cost of the linear algebra, and a larger value of m reduces the number of function calls.

Throughout this section, procedures for updating Z and B are described, but any given implementation uses only one of them at a time. Since $B = ZT$, it is

assumed that elements of Z can be computed in an implicit method and elements of B can be computed in an explicit method, as T is stored in both cases.

It is sufficient to show how to modify the relevant matrices for two cases—when one constraint is added and when one constraint is removed from the working set—since all other changes to the working set are repetitions of these two actions. In many cases, subscripts refer to the various stages of update with respect to adding or dropping one constraint. Fully updated values will be represented with bars above the variable name. In this context, it is important to note that the phrase "fully updated" only refers to the specific action of adding or dropping one constraint. Subscripts will continue to be used to refer to specific elements in a matrix or vector, as well.

In practice, all algorithms described in the remainder of this chapter store and use B' and either B or Z. This is not a necessary implementation but is done—at the expense of storing an additional mn floating point numbers—in order to take advantage of already-existing optimized BLAS routines[1]. If memory storage is a large concern, the algorithms may be modified to store only B' and perform matrix-matrix, matrix-vector, or vector-vector operations on the appropriate rows. If speed is a concern, it should be possible to implement BLAS-like routines that operate on subsets of rows and that are at least as fast as their full-row counterparts.

5.3.1 Removing a constraint from the working set

Updating B, Z, and T

When computing a search direction, the i^{th} component of p, p_i, will be zero if $i \in \mathcal{W}(x)$ since the i^{th} row of B, $e_i^T B$, is also zero. If, after the next update to the working set, $i \notin \bar{\mathcal{W}}$, then the i^{th} component of subsequent search directions are nonzero. This is accomplished by "restoring" the i^{th} row of B so that $e_i^T \bar{B} = e_i^T B'$. Formally, $\bar{B}$ can be defined as a rank-1 update to B:

$$\bar{B} = B + e_i b^T \quad \text{where} \quad b^T = e_i^T B'. \tag{5.4}$$

[1]Basic Linear Algebra Subprograms (BLAS) is an interface to call architecture-specific optimized routines that perform various linear algebra tasks.

The QR factors of B, Z and T, can be updated using procedures described by Daniel, Gragg, Kaufman, and Stewart [DGKS76], which are summarized below.

From equation (5.4), $\bar{B}$ can be expressed as:

$$\bar{B} = B + e_i b^T = \begin{pmatrix} Z & e_i \end{pmatrix} \begin{pmatrix} T \\ b^T \end{pmatrix} = \begin{pmatrix} Z & e_i \end{pmatrix} G_0^T G_0 \begin{pmatrix} T \\ b^T \end{pmatrix}, \tag{5.5}$$

where G_0 is a product of Givens matrices such that

$$G_0 \begin{pmatrix} T \\ b^T \end{pmatrix} = \begin{pmatrix} \bar{T} \\ 0 \end{pmatrix},$$

and $\bar{T}$ is a nonsingular upper-triangular matrix. Given $r = \dim(T)$,

$$G_0 = G_{0,r} G_{0,r-1} \cdots G_{0,2} G_{0,1},$$

where $G_{0,i}$ is a Givens matrix (plane rotation) that operates on rows i and $r+1$ of T and zeros out the i^{th} element of row $r+1$.

To illustrate this process, consider the case for $r = 4$. Elements denoted by "x" describe an original value, or, after the first step, an element that is unmodified from the last step. Elements that have been modified from the previous step are denoted by "m" (or "0" if the present value is zero). Blank elements denote unchanged zeros.

$$\begin{pmatrix} T \\ b^T \end{pmatrix} = \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & & \mathtt{x} & \mathtt{x} \\ & & & \mathtt{x} \\ \mathtt{x} & \mathtt{x} & \mathtt{x} & \mathtt{x} \end{pmatrix} \xrightarrow{G_{0,1}} \begin{pmatrix} \mathtt{m} & \mathtt{m} & \mathtt{m} & \mathtt{m} \\ & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & & \mathtt{x} & \mathtt{x} \\ & & & \mathtt{x} \\ \mathtt{0} & \mathtt{m} & \mathtt{m} & \mathtt{m} \end{pmatrix} \xrightarrow{G_{0,2}} \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & \mathtt{m} & \mathtt{m} & \mathtt{m} \\ & & \mathtt{x} & \mathtt{x} \\ & & & \mathtt{x} \\ & \mathtt{0} & \mathtt{m} & \mathtt{m} \end{pmatrix}$$

$$\xrightarrow{G_{0,3}} \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & & \mathtt{m} & \mathtt{m} \\ & & & \mathtt{x} \\ & & \mathtt{0} & \mathtt{m} \end{pmatrix} \xrightarrow{G_{0,4}} \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & & \mathtt{x} & \mathtt{x} \\ & & & \mathtt{m} \\ & & & \mathtt{0} \end{pmatrix} = \begin{pmatrix} \bar{T} \\ 0 \end{pmatrix}.$$

The upper-triangular matrix $\bar{T}$ is nonsingular since, due to the nature of Givens rotations, $|\bar{T}_{i,i}| \geq |T_{i,i}|$ for $i = 1, \ldots, r$ and because T is nonsingular.

Once Z is postmultiplied by G_0^T and T is premultiplied by G_0, $\bar{Z}$ and $\bar{T}$ are obtained from (5.5):

$$\bar{B} = \begin{pmatrix} Z & e_i \end{pmatrix} G_0^T G_0 \begin{pmatrix} T \\ b^T \end{pmatrix} = \begin{pmatrix} \bar{Z} & z \end{pmatrix} \begin{pmatrix} \bar{T} \\ 0 \end{pmatrix} = \bar{Z}\bar{T}.$$

Because $Z^T e_i = 0$, the columns of $\begin{pmatrix} Z & e_i \end{pmatrix}$ are orthonormal. Since Givens matrices and products of Givens matrices are orthonormal, the columns of $\bar{Z}$ are orthonormal as well.

In order to better describe the other updates and because all modifications to Z and T are done *in situ*, it is useful to identify explicitly each of the partial modifications to Z and T. In particular, starting with $Z_0 = Z$, the first update adds a column to Z_0:

$$Z_1 = \begin{pmatrix} Z_0 & e_i \end{pmatrix}. \tag{5.6}$$

Next, Z_1 is postmultiplied by G_0^T so that

$$Z_2 = Z_1 G_0^T. \tag{5.7}$$

Finally, the last column of Z_2, z, is deleted so that $\bar{Z}$ $(:= Z_3)$ satisfies,

$$Z_2 = \begin{pmatrix} \bar{Z} & z \end{pmatrix}. \tag{5.8}$$

The matrices T_0, T_1, T_2, and $\bar{T} = T_3$ are defined in a similar way.

Updating q and v

The reduced column vectors $q = Z^T p$ and $v = Z^T g$ can be updated in one of two ways. The simplest and usually least efficient way to compute $\bar{q}$ and $\bar{v}$ is to compute them from scratch after Z is updated. The second method involves updating q and v using corresponding updates to Z. As $q = Z^T p$, (5.6) gives

$$q_1 = Z_1^T p = \begin{pmatrix} Z^T \\ e_i^T \end{pmatrix} p = \begin{pmatrix} Z^T p \\ p_i \end{pmatrix}.$$

Next, (5.7) yields

$$q_2 = Z_2^T p = G_0 Z_1^T p = G_0 q_1 = G_{0,r} G_{0,r-1} \cdots G_{0,2} G_{0,1} q_1.$$

Finally, from (5.8), set $\bar{q} = q_3$ to be the first r components of q_2 since

$$q_2 = Z_2^T p = \begin{pmatrix} Z_3^T \\ z^T \end{pmatrix} p = \begin{pmatrix} Z_3^T p \\ z^T p \end{pmatrix} = \begin{pmatrix} q_3 \\ z^T p \end{pmatrix}.$$

The update for $v = Z^T g$ is done in exactly the same way. The cost of updating q and v versus the cost of computing from them scratch will be addressed later.

Updating the Cholesky factor of $Z^T H Z$

As $R^T R = Z^T H Z$, one way to update R is to follow the same steps to update Z. Because $Z^T e_i = B^T e_i = 0$, the definition of H (2.11) implies that

$$H e_i = Z R^T R Z^T e_i + \sigma (I_n - Z Z^T) e_i = \sigma e_i. \tag{5.9}$$

Thus, the first partial update to Z of (5.6) gives the first partial update to R as

$$R_1 = \begin{pmatrix} R & 0 \\ 0 & \sqrt{\sigma} \end{pmatrix},$$

since R_1 satisfies

$$R_1^T R_1 = \begin{pmatrix} R^T R & 0 \\ 0 & \sigma \end{pmatrix} = \begin{pmatrix} Z^T H Z & 0 \\ 0 & \sigma \end{pmatrix} = \begin{pmatrix} Z^T H Z & Z^T H e_i \\ e_i^T H Z & e_i^T H e_i \end{pmatrix} = Z_1^T H Z_1.$$

Equation (5.7) suggests that the next updated quantity should be $R_1 G_0^T$ since

$$(R_1 G_0^T)^T (R_1 G_0^T) = G_0 R_1^T R_1 G_0^T = G_0 Z_1^T H Z_1 G_0^T = Z_2^T H Z_2.$$

This is an unsuitable update, however, because $R_1 G_0^T$ is not upper-triangular. Using the same notation as above, the loss of triangularity can be seen with an

example, for instance, when $r = 3$:

$$R_1 = \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \\ & \mathtt{x} & \mathtt{x} & \\ & & \mathtt{x} & \\ & & & \mathtt{x} \end{pmatrix} \xrightarrow{G_{0,1}^T} \begin{pmatrix} \mathtt{m} & \mathtt{x} & \mathtt{x} & \mathtt{m} \\ & \mathtt{x} & \mathtt{x} & \\ & & \mathtt{x} & \\ \mathtt{m} & & & \mathtt{m} \end{pmatrix} \xrightarrow{G_{0,2}^T} \begin{pmatrix} \mathtt{x} & \mathtt{m} & \mathtt{x} & \mathtt{m} \\ & \mathtt{m} & \mathtt{x} & \mathtt{m} \\ & & \mathtt{x} & \\ \mathtt{x} & \mathtt{m} & & \mathtt{m} \end{pmatrix}$$

$$\xrightarrow{G_{0,3}^T} \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{m} & \mathtt{m} \\ & \mathtt{x} & \mathtt{m} & \mathtt{m} \\ & & \mathtt{m} & \mathtt{m} \\ \mathtt{x} & \mathtt{x} & \mathtt{m} & \mathtt{m} \end{pmatrix} = R_1 G_0^T.$$

In general, the matrix $R_1 G_0^T$ is upper-triangular with a "row spike" in row $r+1$. To obtain the upper-triangular update R_2, the intermediate quantity $R_1 G_0^T$ is multiplied on the left by another set of Givens matrices:

$$R_2 = G_1 R_1 G_0^T = (G_{1,r} G_{1,r-1} \cdots G_{1,2} G_{1,1}) R_1 G_0^T,$$

where $G_{1,i}$ is a Givens matrix that acts on rows i and $r+1$ of the product

$$G_{1,i-1} \cdots G_{1,1} R_1 G_0^T$$

to zero out the i^{th} element in row $r+1$. Premultiplying $R_1 G_0^T$ by G_1 preserves the necessary identities since

$$\begin{aligned} R_2^T R_2 &= (G_1 R_1 G_0^T)^T (G_1 R_1 G_0^T) \\ &= G_0 R_1^T G_1^T G_1 R_1 G_0^T \\ &= G_0 R_1^T R_1 G_0^T \\ &= G_0 Z_1^T H Z_1 G_0^T \\ &= Z_2^T H Z_2. \end{aligned} \tag{5.10}$$

Finally, $\bar{R} = R_3$ is defined as the $r \times r$ leading principal submatrix of R_2:

$$R_2 = \begin{pmatrix} R_3 & a_1 \\ 0 & \gamma \end{pmatrix},$$

where a_1 is a vector of length r and γ is a scalar. Equation (5.10) gives

$$\begin{pmatrix} R_3^T R_3 & R_3^T a_1 \\ a_1^T R_3 & a_1^T a_1 + \gamma^2 \end{pmatrix} = R_2^T R_2 = Z_2^T H Z_2 = \begin{pmatrix} Z_3^T H Z_3 & Z_3^T H z \\ z^T H Z_3 & z^T H z \end{pmatrix},$$

and $R_3^T R_3 = Z_3^T H Z_3$ as required.

Updating the Cholesky factor of $B^T H B$

Instead of updating R directly, it is also possible to update R indirectly by updating a Cholesky factor of $B^T H B$. The first step is to define $S = RT$, so that

$$S^T S = T^T R^T R^T T = T^T Z^T H Z T = B^T H B.$$

Assuming $\bar{S}$ can be computed such that $\bar{S}^T \bar{S} = \bar{B}^T H \bar{B}$, $\bar{R}$ can be obtained by using a triangular solver to solve the equation $\bar{S} = \bar{R}\bar{T}$ for $\bar{R}$. If R is updated using this method, R is transformed into S only once, before any constraints are dropped. The Cholesky factor S is updated for each dropped constraint. Finally, $\bar{R}$ is computed only once at the very end after all relevant constraints have been removed (and added, if the same Cholesky factor is used when adding constraints). Computing S from R and computing $\bar{R}$ from $\bar{S}$ each take $O(m^3)$ operations.

Because the process to obtain the updated $\bar{S}$ is different from the process used to obtain $\bar{R}$ directly, subscripts will still be used to describe partial updates but will no longer correspond to similar stages in updating Z.

Given an initial factor S, the updated matrix $\bar{S}$ must be upper-triangular and satisfy

$$\bar{S}^T \bar{S} = (B + e_i b^T)^T H (B + e_i b^T) = S^T S + B^T H e_i b^T + b e_i^T H B + b e_i^T H e_i b^T. \quad (5.11)$$

The structure of (5.11) suggests a rank-1 update to S. We define S_1 to be of the form

$$S_1 = S + a_1 a_2^T,$$

where a_1 and a_2 are column vectors of length r. Substituting into (5.11) and attempting to match terms yields values $a_1 = S^{-T} B^T H e_i$ and $a_2 = b$. This is only

a partial update, however, as (5.11) gives

$$\begin{aligned} S_1^T S_1 &= S^T S + S^T S^{-T} B^T H e_i b^T + b e_i^T H B S^{-1} S + b a_1^T a_1 b^T \\ &= S^T S + B^T H e_i b^T + b e_i^T H B + (a_1^T a_1) b b^T \\ &= \bar{S}^T \bar{S} + (a_1^T a_1 - e_i^T H e_i) b b^T. \end{aligned}$$

Given that $He_i = \sigma e_i$ (5.9) and $B^T e_i = 0$, the vector a_1 is exactly zero:

$$a_1 = S^{-T} B^T H e_i = \sigma S^{-T} B^T e_i = 0.$$

Since $a_1 = 0$, $S_1 = S$ and $\bar{S}$ must satisfy

$$\bar{S}^T \bar{S} = S_1^T S_1^T + \sigma b b^T = S^T S + \sigma b b^T.$$

It is worth noting that He_i and a_1 simplify in such a manner only when removing a constraint. In general, the vector a_1 can be nonzero when adding a constraint.

Next, we define

$$S_2 = \begin{pmatrix} S_1 \\ (\sqrt{\sigma}) b^T \end{pmatrix},$$

so that

$$S_2^T S_2 = \begin{pmatrix} S_1^T & (\sqrt{\sigma}) b \end{pmatrix} \begin{pmatrix} S_1 \\ (\sqrt{\sigma}) b^T \end{pmatrix} = S_1^T S_1 + \sigma b b^T.$$

Finally, we define $\bar{S}$ to be the $r \times r$ matrix such that

$$\begin{pmatrix} \bar{S} \\ 0 \end{pmatrix} = G_2 S_2 = (G_{2,r} G_{2,r-1} \cdots G_{2,2} G_{2,1}) \begin{pmatrix} S_1 \\ (\sqrt{\sigma}) b^T \end{pmatrix},$$

so that $G_{2,i}$ is a Givens matrix that operates on rows i and $r+1$ of matrix S_2 to zero out the i^{th} element of row $r+1$. Observe that the nonsingular and upper-triangular matrix $\bar{S}$ satisfies

$$\bar{S}^T \bar{S} = \begin{pmatrix} \bar{S}^T & 0 \end{pmatrix} \begin{pmatrix} \bar{S} \\ 0 \end{pmatrix} = S_2^T G_2^T G_2 S_2 = S_2^T S_2 = S_1^T S_1 + \sigma b b^T,$$

as required.

Computational costs

In order to discuss the computational efficiency of a particular algorithm, we define a *flop*—for floating point operation—to be an addition, subtraction, multiplication or division. For example, computing the expression $a * (b + c/d)$ takes 3 flops, if a, b, c, $d \in \mathbb{R}$.

If the number of constraints removed from the working set is denoted by n_d, then updating R, the Cholesky factor of $Z^T H Z$, after n_d constraints have been dropped costs approximately $(6r^2 + 19r)n_d$ flops. On the other hand, it costs $(3r^2 + 5r)n_d$ flops to update S, the Cholesky factor of $B^T H B$.

This does not include the conversion costs to convert from R to S and back again. In theory, these costs can be as low as $\frac{1}{3}r^3 + \frac{1}{2}r^2 + \frac{1}{6}r$ flops per conversion. In practice, since there are no BLAS routines for forward- and backward-solves for triangular matrices, a BLAS routine is used that performs forward- and backward-solves on general matrices, which costs r^3 flops per conversion.

Thus, the total cost to update S, including the conversion from R to S and back again, is $(3r^2 + 5r)n_d + 2r^3$ flops. One factor not considered here is that the cost of converting from R to S and back again can be shared if S is updated (instead of R) when constraints are *added* as well. This consideration turns out to be a moot point, however, as R is almost always cheaper to update than S when constraints are added. More information about this point can be found at the end of Section 5.3.2.

Given the number of flops required for each method, it is straightforward to show that the number of flops required to update R is less than the number required to update S if and only if

$$n_d > \frac{2r^2}{3r + 14}.$$

As it is implemented, all RH-B algorithms update either R or S, based on whether the above inequality is satisfied.

The same analysis can be applied to choosing the method to compute $\bar{q}$ and $\bar{v}$. In particular, it costs $12r^2 n_d$ flops to update q *and* v when dropping n_d constraints. The cost to recompute q and v from scratch depends on whether an

implicit or explicit method is used. Using an implicit method, it costs $4nr+2r^2-2r$ flops to update both quantities, regardless of the number of constraints dropped. Similarly, it costs $4nr-2r$ flops using an explicit method. Because the decision to update versus compute from scratch also depends on constraints that are added, final comparisons are only presented at the end of Section 5.3.2.

5.3.2 Adding a constraint to the working set

Updating B, Z, and T

Adding a constraint with index i to the working set is equivalent to restricting any further movement in the i^{th} component. Since any search direction p is in the column space of Z, restricting movement in the i^{th} component is accomplished by "zeroing out" the i^{th} row of Z or B. As with the previous case, the updated matrix $\bar{B}$ is a rank-1 update to B; i.e.,

$$\bar{B} = B + e_i b^T \text{ where } b^T = -e_i^T B.$$

The row vector b^T, which is the negative of the i^{th} row of B, will be used throughout this subsection and is not the same quantity used in the previous subsection. Just like before, every other required matrix or vector modification is a direct consequence of modifying B.

Once again T and Z are updated using a method described in Daniel, et al [DGKS76], which is summarized below.

First, if $e_i \notin \text{col}(B)$, we define z to be the normalized component of e_i that is orthogonal to Z, i.e., z and ρ are defined so that

$$\rho z = (I_n - ZZ^T)e_i = e_i - ZZ^T e_i = e_i - Zz_i \text{ with } z_i = Z^T e_i, \tag{5.12}$$

where $\|z\| = 1$ and $\rho > 0$. Note that z_i, a column vector representing the i^{th} row of Z, can be obtained by solving $T^T z_i = -b$ if using an implicit method. If $e_i \in \text{col}(B)$ then $\rho = 0$ and z can be any unit vector. In this case, z is not a normalized component of e_i orthogonal to Z.

If $e_i \in \text{col}(B)$, then the matrix $\begin{pmatrix} B & e_i \end{pmatrix}$ is rank deficient and the process for "zeroing out" row i in B will fail to the extent that the resulting updated

matrix will also be rank deficient. To remedy this and to make the algorithm for "zeroing out" a row well-defined, a column is dropped from B before e_i is added. Keeping with the philosophy used in a limited-memory algorithm, the column that is dropped is the oldest column of B that is not orthogonal to e_i. If B_0 is the matrix that results from taking B and dropping such a column, then the matrix $\begin{pmatrix} B_0 & e_i \end{pmatrix}$ has full column rank and "zeroing out" the i^{th} row returns a matrix that also has full column rank. The details on how to drop a column from B and how to update the relevant matrices that depend on B can be found in Section 5.3.3.

Assuming $e_i \notin \text{col}(B)$ and that an explicit method is used, it is desirable to use reorthogonalization, when applicable, on z. More information can be found in Daniel, et al. [DGKS76].

For the remainder of this subsection, without loss of generality, it is assumed that $e_i \notin \text{col}(B)$ and $\rho > 0$. If $e_i \in \text{col}(B)$, a column of B is dropped, all matrices that depend on B are adjusted accordingly and the updates provided below are applied to the resulting matrices. Given z from equation (5.12), matrices Z_1 and T_1 are defined so that

$$B = ZT = \begin{pmatrix} Z & z \end{pmatrix} \begin{pmatrix} T \\ 0 \end{pmatrix} = Z_1 T_1.$$

If the i^{th} component of z is denoted by the scalar ζ, then the i^{th} row of Z_1 is the row vector $\begin{pmatrix} z_i^T & \zeta \end{pmatrix}$. Daniel et al [DGKS76] show that $\zeta = \rho$. Thus,

$$\zeta^2 = \rho^2 = (\rho z)^T(\rho z) = (e_i - Zz_i)^T(e_i - Zz_i) = 1 - z_i^T z_i. \tag{5.13}$$

The significance of this equation is that, for implicit methods, it allows access to the only element of z that is needed without requiring the full vector z, which is expensive to compute. This will be seen in the next several paragraphs.

Next, we define a product of Givens matrices

$$G_0 = G_{0,1} G_{0,2} \cdots G_{0,r-1} G_{0,r}$$

such that

$$e_i^T Z_1 G_0^T = \begin{pmatrix} z_i^T & \zeta \end{pmatrix} G_{0,r}^T G_{0,r-1}^T \cdots G_{0,2}^T G_{0,1}^T = \begin{pmatrix} 0 & \tau \end{pmatrix}$$

for a scalar τ. Note that $G_{0,i}^T$ acts on columns i and $r+1$ of the product of

$$\begin{pmatrix} z_i^T & \zeta \end{pmatrix} G_{0,r} G_{0,r-1} \cdots G_{0,i+1}$$

to zero out the entry in column i. As Givens rotations preserve length, it follows that $\tau = \pm 1$.

Given G_0, we define $Z_2 = Z_1 G_0^T$ and $T_2 = G_0 T_1$ so that

$$Z_2 T_2 = Z_1 G_0^T G_0 T_1 = Z_1 T_1 = B. \tag{5.14}$$

Finally, we set $\bar{Z}$ ($:= Z_3$) to be the first r columns of Z_2 and $\bar{T}$ ($:= T_3$) to be the first r rows of T_2. Daniel et al [DGKS76] show that the last column of Z_2 is τe_i and the last row of T_2 is $-\tau b^T$, which, together with (5.14), gives

$$B = Z_2 T_2 = \begin{pmatrix} \bar{Z} & \tau e_i \end{pmatrix} \begin{pmatrix} \bar{T} \\ -\tau b^T \end{pmatrix} = \bar{Z}\bar{T} - \tau^2 e_i b^T = \bar{Z}\bar{T} - e_i b^T.$$

Thus

$$\bar{Z}\bar{T} = B + e_i b^T = \bar{B},$$

and $e_i^T \bar{Z} = 0$ as required. Given $\rho > 0$, the columns of Z_2 are orthonormal since G_0 is orthonormal, and hence the columns of $\bar{Z}$ are orthonormal as required. The matrix $\bar{T}$ is upper-triangular, which is evident by considering a small example, say, when $r = 4$. Using the same notation as before, the transformation from T_1 to T_2 is as follows:

$$T_1 = \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & & \mathtt{x} & \mathtt{x} \\ & & & \mathtt{x} \\ & & & \end{pmatrix} \xrightarrow{G_{0,4}} \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & & \mathtt{x} & \mathtt{x} \\ & & & \mathtt{m} \\ & & & \mathtt{m} \end{pmatrix} \xrightarrow{G_{0,3}} \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & & \mathtt{m} & \mathtt{m} \\ & & & \mathtt{x} \\ & & \mathtt{m} & \mathtt{m} \end{pmatrix}$$

$$\xrightarrow{G_{0,2}} \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & \mathtt{m} & \mathtt{m} & \mathtt{m} \\ & & \mathtt{x} & \mathtt{x} \\ & & & \mathtt{x} \\ & \mathtt{m} & \mathtt{m} & \mathtt{m} \end{pmatrix} \xrightarrow{G_{0,1}} \begin{pmatrix} \mathtt{m} & \mathtt{m} & \mathtt{m} & \mathtt{m} \\ & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & & \mathtt{x} & \mathtt{x} \\ & & & \mathtt{x} \\ \mathtt{m} & \mathtt{m} & \mathtt{m} & \mathtt{m} \end{pmatrix} = T_2 = \begin{pmatrix} \bar{T} \\ -\tau b^T \end{pmatrix}.$$

Updating q and v

As with the previous subsection, the vectors $v = Z^T g$ and $q = Z^T p$ can be computed from scratch once $\bar{Z}$ is obtained or they can be updated based on the partial updates to Z.

Using the definition of q and equation (5.12), we define

$$q_1 = Z_1^T p = \begin{pmatrix} Z^T \\ z^T \end{pmatrix} p = \begin{pmatrix} Z^T p \\ \rho^{-1}(e_i^T - e_i^T Z Z^T)p \end{pmatrix} = \begin{pmatrix} Z^T p \\ \rho^{-1}(p_i - z_i^T q) \end{pmatrix},$$

where p_i is the i^{th} component of p. Next we define

$$q_2 = Z_2^T p = G_0 Z_1^T p = G_{0,1} G_{0,2} \cdots G_{0,r-1} G_{0,r} q_1,$$

and set $\bar{q} = q_3$ to be the first r elements of vector q_2 since

$$q_2 = Z_2^T p = \begin{pmatrix} \bar{Z}^T \\ \tau e_i^T \end{pmatrix} p = \begin{pmatrix} \bar{Z}^T p \\ \tau p_i \end{pmatrix}.$$

The vector $v = Z^T g$ is updated in a similar fashion.

Updating the Cholesky factor of $Z^T H Z$

As in the case when removing constraints, it is possible to update R directly following the same sequence used to update Z. Since $Z^T z = 0$, the definition of H (2.11) implies

$$Hz = \sigma z.$$

Therefore,

$$R_1 = \begin{pmatrix} R & 0 \\ 0 & \sqrt{\sigma} \end{pmatrix},$$

since

$$R_1^T R_1 = \begin{pmatrix} R^T R & 0 \\ 0 & \sigma \end{pmatrix} = \begin{pmatrix} Z^T H Z & Z^T H z \\ z^T H Z & z^T H z \end{pmatrix} = Z_1^T H Z_1.$$

As before, the quantity $R_1 G_0^T$ is not upper-triangular; in fact, it will likely have no zero elements at all. Consider an example where $r = 3$:

$$R_1 = \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \\ & \mathtt{x} & \mathtt{x} & \\ & & \mathtt{x} & \\ & & & \mathtt{x} \end{pmatrix} \xrightarrow{G_{0,3}^T} \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{m} & \mathtt{m} \\ & \mathtt{x} & \mathtt{m} & \mathtt{m} \\ & & \mathtt{m} & \mathtt{m} \\ & & \mathtt{m} & \mathtt{m} \end{pmatrix} \xrightarrow{G_{0,2}^T} \begin{pmatrix} \mathtt{x} & \mathtt{m} & \mathtt{x} & \mathtt{m} \\ & \mathtt{m} & \mathtt{x} & \mathtt{m} \\ & \mathtt{m} & \mathtt{x} & \mathtt{m} \\ & \mathtt{m} & \mathtt{x} & \mathtt{m} \end{pmatrix}$$
$$\xrightarrow{G_{0,1}^T} \begin{pmatrix} \mathtt{m} & \mathtt{x} & \mathtt{x} & \mathtt{m} \\ \mathtt{m} & \mathtt{x} & \mathtt{x} & \mathtt{m} \\ \mathtt{m} & \mathtt{x} & \mathtt{x} & \mathtt{m} \\ \mathtt{m} & \mathtt{x} & \mathtt{x} & \mathtt{m} \end{pmatrix} = R_1 G_0^T$$

To prevent full loss of triangularity, $R_1 G_0^T$ is premultiplied by another product of Givens matrices and the two sets of Givens rotations are applied in an interlacing fashion. We define $G_1 = G_{1,r} G_{1,r-1} \cdots G_{1,3} G_{1,2}$, where $G_{1,i}$ operates on rows i and $r+1$ of the current product of matrices to zero out element $r+1$ in row i. This product of matrices is still not upper-triangular, so it is labeled as intermediate update R_{1a}:

$$R_{1a} = [G_{1,2} \cdots ([G_{1,r-1}([G_{1,r}(R_1 G_{0,r}^T)] G_{0,r-1})] \cdots G_{0,2}^T)] G_{0,1}^T.$$

To see an example of how the Givens matrices are applied, consider an example where $r = 3$:

$$R_1 = \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \\ & \mathtt{x} & \mathtt{x} & \\ & & \mathtt{x} & \\ & & & \mathtt{x} \end{pmatrix} \xrightarrow{G_{0,3}^T} \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{m} & \mathtt{m} \\ & \mathtt{x} & \mathtt{m} & \mathtt{m} \\ & & \mathtt{m} & \mathtt{m} \\ & & \mathtt{m} & \mathtt{m} \end{pmatrix} \xrightarrow{G_{1,3}} \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & & \mathtt{m} & \mathtt{0} \\ & & \mathtt{m} & \mathtt{m} \end{pmatrix}$$
$$\xrightarrow{G_{0,2}^T} \begin{pmatrix} \mathtt{x} & \mathtt{m} & \mathtt{x} & \mathtt{m} \\ & \mathtt{m} & \mathtt{x} & \mathtt{m} \\ & & \mathtt{x} & \\ & \mathtt{m} & \mathtt{x} & \mathtt{m} \end{pmatrix} \xrightarrow{G_{1,2}} \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \mathtt{x} \\ & \mathtt{m} & \mathtt{m} & \mathtt{0} \\ & & \mathtt{x} & \\ & \mathtt{m} & \mathtt{m} & \mathtt{m} \end{pmatrix} \xrightarrow{G_{0,1}^T} \begin{pmatrix} \mathtt{m} & \mathtt{m} & \mathtt{m} & \mathtt{m} \\ & \mathtt{x} & \mathtt{x} & \\ & & \mathtt{x} & \\ \mathtt{m} & \mathtt{m} & \mathtt{m} & \mathtt{m} \end{pmatrix} = R_{1a}.$$

While it is possible to zero out entry $(1, r+1)$ by premultiplying by a final Givens matrix $G_{1,1}$, it is not necessary given the next update to R. Although R_{1a}

satisfies

$$R_{1a}^T R_{1a} = G_0 R_1^T G_1^T G_1 R_1 G_0^T = G_0 R_1^T R_1 G_0^T = Z_2^T H Z_2,$$

it contains a row spike in row $r+1$. Therefore, we define R_2 to be the product

$$R_2 = G_2 R_{1a},$$

where

$$G_2 = G_{2,r} G_{2,r-1} \cdots G_{2,2} G_{2,1}.$$

The Givens matrix $G_{2,i}$ operates on rows i and $r+1$ of the product

$$G_{2,i-1} \cdots G_{2,2} G_{2,1} R_{1a}$$

to zero out element i in row $r+1$. Note that R_2 is upper-triangular and that

$$R_2^T R_2 = R_{1a}^T G_2^T G_2 R_{1a} = R_{1a}^T R_{1a} = Z_2^T H Z_2$$

as required.

Finally, as in the previous case, $\bar{R} = R_3$ is taken to be the $r \times r$ leading principal submatrix of R_2, since the first r elements of the last row of R_3 are zero.

Updating the Cholesky factor of $B^T H B$

As with the previous case, it is possible to update R indirectly by updating a Cholesky factor S, where $S^T S = B^T H B$. The details for converting from R to S and from $\bar{S}$ to $\bar{R}$ can be found in the previous subsection on removing a constraint. Numerical subscripts will refer to partial updates to S, but do not correspond with partial updates to Z, given the direct dependence of S on B.

Using equation (5.11), we define the first partial update to be

$$S_1 = S + a_1 a_2^T,$$

where $a_1 = S^{-T} B^T H e_i$ and $a_2 = b$. Recall that $b = -e_i^T B$ when adding a constraint. From equation (5.11), $\bar{S}$ must satisfy

$$S_1^T S_1 = \bar{S}^T \bar{S} + (a_1^T a_1 - e_i^T H e_i) b b^T.$$

The similarity to the previous case ends here. The definition of H (2.11) implies

$$He_i = ZR^T RZ^T e_i + \sigma(I_n - ZZ^T)e_i = ZR^T RZ^T e_i + \sigma(e_i - ZZ^T e_i). \qquad (5.15)$$

Since $B = ZT$ and $R = ST$, this equation can be used to simplify the expression for a_1:

$$\begin{aligned} a_1 &= S^{-T}B^T He_i \\ &= (SS^{-1})S^{-T}(ZT)^T(ZR^T RZ^T e_i + \sigma e_i - \sigma ZZ^T e_i) \\ &= S(S^T S)^{-1}T^T(R^T RZ^T e_i + \sigma Z^T e_i - \sigma Z^T e_i) \\ &= S(B^T HB)^{-1}T^T(Z^T HZ)(Z^T e_i) \\ &= S(T^T Z^T HZT)^{-1}T^T(Z^T HZ)z_i \\ &= ST^{-1}(Z^T HZ)^{-1}T^{-T}T^T(Z^T HZ)z_i \\ &= ST^{-1}z_i. \end{aligned}$$

Although S_1 is defined formally, it is never computed directly, due to its total loss of upper-triangularity. To prevent a full loss of triangularity, the two terms composing S_1 are premultiplied by a product of Givens matrices. We define

$$G_2 = G_{2,1}G_{2,2}\cdots G_{2,r-2}G_{2,r-1}$$

where $G_{2,i}$ is a Givens matrix that acts on elements i and $i+1$ of $G_{2,i+1}\cdots G_{2,r-1}a_1$ to zero out element $i+1$ so that

$$G_2 a_1 = \gamma e_1$$

where $\gamma = \pm\|a_1\|$. Next, we define

$$S_2 = G_2 S_1 = G_2 S + G_2 a_1 a_2^T = G_2 S + \gamma e_1 a_2^T.$$

The matrix S_2 is upper-Hessenberg since $G_2 S$ is upper-Hessenberg and $\gamma e_1 a_2^T$ is a matrix whose only nonzero elements are in the first row.

Next, we define

$$S_3 = G_3 S_2,$$

where

$$G_3 = G_{3,1}G_{3,2}\cdots G_{3,r-1},$$

and $G_{3,i}$ is a Givens matrix that acts on rows i and $i+1$ of the product

$$G_{3,i+1} \cdots G_{3,r-1} S_2$$

to zero out element $(i, i+1)$. Note that S_3 is upper-triangular and satisfies

$$\begin{aligned} S_3^T S_3 &= S_1^T G_2^T G_3^T G_3 G_2 S_1 \\ &= S_1^T S_1 \\ &= \bar{S}^T \bar{S} - (e_i^T H e_i - a_1^T a_1) b b^T, \end{aligned}$$

where

$$\begin{aligned} e_i^T H e_i - a_1^T a_1 &= e_i^T (Z Z^T H Z Z^T e_i + \sigma e_i - \sigma Z Z^T e_i) - e_i^T Z T^{-T} R_B^T R_B T^{-1} Z^T e_i \\ &= e_i^T Z Z^T H Z Z^T e_i + \sigma - \sigma e_i^T Z Z^T e_i - e_i^T Z Z^T H Z Z^T e_i \\ &= \sigma (1 - e_i^T Z Z^T e_i) \\ &= \sigma (1 - z_i^T z_i) \\ &= \sigma \zeta^2 \\ &> 0, \end{aligned}$$

by equations (5.15) and (5.13).

Since $\sigma \zeta^2 > 0$, we define

$$S_4 = \begin{pmatrix} S_3 \\ (\sqrt{\sigma}) \zeta b^T \end{pmatrix}$$

so that S_4 satisfies

$$S_4^T S_4 = S_3^T S_3 + (e_i^T H e_i - a_1^T a_1) b b^T = \bar{S}^T \bar{S}.$$

Finally, $\bar{S}$ is defined as the $r \times r$ nonsingular upper-triangular matrix such that

$$\begin{pmatrix} \bar{S} \\ 0 \end{pmatrix} = G_4 S_4 = (G_{4,r} G_{4,r-1} \cdots G_{4,2} G_{4,1}) \begin{pmatrix} S_3 \\ (\sqrt{\sigma}) \zeta b^T \end{pmatrix},$$

where $G_{4,i}$ is a Givens matrix that operates on rows i and $r+1$ of the product $G_{4,i-1} \cdots G_{4,1}$ to zero out the i^{th} element of row $r+1$. Observe that

$$\bar{S}^T \bar{S} = \begin{pmatrix} \bar{S}^T & 0 \end{pmatrix} \begin{pmatrix} \bar{S} \\ 0 \end{pmatrix} = S_4^T G_4^T G_4 S_4 = S_3^T S_3 + (\sigma \zeta^2) b b^T,$$

as required.

Computational costs

Recall from the end of Section 5.3.1, that a *flop* is defined to be an addition, subtraction, multiplication or division.

If the number of constraints added to the working set is denoted n_a, then updating R after n_a constraints are added requires $(9r^2 + 23r)n_a$ flops. On the other hand, updating S costs $(11r^2 + 21r - 19)n_a$ flops, not including the cost of converting R to S and back again. With conversion costs, the number of flops used becomes $(11r^2 + 21r - 19)n_a + 2r^3$. Even without the conversion costs factored in, it is always cheaper to update R if $r \geq 4$. Further, the differences between costs when $r < 4$ is almost negligible. For this reason, the update to R is always used in all RH-B methods implemented to date.

The cost to update q and v after n_a constraints are added is $(12r^2+4r+2)n_a$ flops. Recall that the cost to recompute q and v from scratch is $4nr + 2r^2 - 2r$ if an implicit method is used and $4nr - 2r$ if an explicit method is used. Thus, if n_d represents the number of constraints that are dropped, the number of flops required to compute $\bar{q}$ and $\bar{v}$ is

$$12(n_a + n_d)r^2 + 4n_a r + 2n_a.$$

All RH-B algorithms implemented in MATLAB can compute $\bar{q}$ and $\bar{v}$ using either option, depending on cost. All RH-B algorithms currently implemented in Fortran 90 recompute q and v from scratch after the working set changes.

5.3.3 Dealing with rank reduction

There are two procedures in an RH-B algorithm that remove a column from B and update the other relevant matrices and vectors. The first procedure removes a column from B if, after accepting a new gradient, the resulting basis matrix has $m + 1$ columns, where m is the user-defined limit on the number of columns. The second procedure drops a column from B if, when adding constraint i to the working set, $e_i \in \text{col}(B)$ (see Section 5.3.2 for more information).

To simplify notation and preserve readability, the subscripts representing partial updates to B, Z, T, R, w, q and v within a larger RH-B algorithm at the

time the column-dropping procedure occurs are suppressed. Instead, subscripts and bars are used, respectively, to denote partial and final updates *within the column-dropping procedure only*. Both procedures can be described by the same matrix updates. The only difference between the two procedures is which matrices are updated as B is changed. Both procedures track updates to B, Z, T, and R. The first procedure also updates w, while the second procedure updates q and v. The vectors w, q, and v all share the same structure; namely, they are all reduced vectors of the form $Z^T u'$, for some u'. Therefore, we need only describe how to update a generic reduced vector, denoted $u = Z^T u'$.

Although the first procedure always removes the first column, it is possible that the second procedure can remove any column. Therefore, the general procedure for removing a column is described in terms of removing column k, where $k \in \{1, 2, \ldots, r\}$ with $r = \text{rank}(B)$. The algorithm for removing a column from B and modifying Z and T comes from Daniel, et al [DGKS76].

Let B be partitioned as

$$B = \begin{pmatrix} B' & b & B'' \end{pmatrix},$$

so that the k^{th} column of B is b. Matrices B' and B'' may be empty matrices with zero columns. The partition for B defines a similar partition for T, namely,

$$T = \begin{pmatrix} T' & t & T'' \end{pmatrix}.$$

Thus,

$$\begin{pmatrix} B' & b & B'' \end{pmatrix} = Z \begin{pmatrix} T' & t & T'' \end{pmatrix},$$

and

$$\begin{pmatrix} B' & B'' \end{pmatrix} = Z \begin{pmatrix} T' & T'' \end{pmatrix} \quad \text{and} \quad b = Zt.$$

We define $\bar{B} = \begin{pmatrix} B' & B'' \end{pmatrix}$ and $T_1 = \begin{pmatrix} T' & T'' \end{pmatrix}$, so that T_1 is upper-Hessenberg. A composition of Givens matrices, denoted G_1, is applied on the left of T_1 to restore it to upper-triangularity, with

$$G_1 = G_{1,k+1} G_{1,k+2} \cdots G_{1,r},$$

so that $G_{1,i}$ is an $r \times r$ Givens matrix that works on rows i and $i-1$ of the product of $G_{1,i+1} \cdots G_{1,r} T_1$ to zero out element $i-1$ in row i. Let $T_2 = G_1 T_1$, so that T_2 is an $r \times (r-1)$ upper-triangular matrix. Finally, $\bar{T}$ is defined to be the leading $(r-1) \times (r-1)$ principal submatrix of T_2, so that

$$T_2 = \begin{pmatrix} \bar{T} \\ 0 \end{pmatrix}.$$

Since $\bar{B} = ZT_1$, it holds that

$$\bar{B} = ZG_1^T G_1 T_1 = ZG_1^T T_2.$$

Thus, Z_1 is defined to be $ZG_1^T = ZG_{1,r}^T G_{1,r-1}^T \cdots G_{1,k+2}^T G_{1,k+1}^T$. If z is the last column of Z_1, then $\bar{Z}$ can be defined so that Z_1 is partitioned as

$$Z_1 = \begin{pmatrix} \bar{Z} & z \end{pmatrix}.$$

We observe that

$$\bar{B} = Z_1 T_2 = \begin{pmatrix} \bar{Z} & z \end{pmatrix} \begin{pmatrix} \bar{T} \\ 0 \end{pmatrix} = \bar{Z}\bar{T},$$

as required.

The sequence of updates to R is motivated by the updates to Z. Since

$$Z_1^T H Z_1 = G_1 Z^T H Z G_1^T = G_1 R^T R G_1^T = (RG_1^T)^T (RG_1^T),$$

we define $R_1 = RG_1^T = RG_{1,r}^T G_{1,r-1}^T \cdots G_{1,k+2}^T$. Unfortunately, R_1 is not upper-triangular—in fact, it is possible that it has no zero elements. To remedy this, another product of Givens matrices is applied on the left,

$$G_2 = G_{2,r} G_{2,r-1} \cdots G_{2,k+1}.$$

The Givens matrices from G_1 and G_2 are applied in an interlaced fashion, with $G_{1,r}^T$ being applied first so that $G_{2,i}$ acts on rows i and $i-1$ of the current product to zero out element $i-1$ in row i. We define $R_2 = G_2 R G_1^T$. Finally, $\bar{R}$ is chosen to be the leading $(r-1) \times (r-1)$ principal submatrix of R_2 so that

$$\bar{R}^T \bar{R} = \bar{Z}^T H \bar{Z}.$$

The sequence of updates to any of the reduced vectors, here denoted by $u = Z^T u'$, is also motivated by the updates to Z. The vector u_1 is defined to be

$$u_1 = Z_1^T u' = G_1 Z^T u' = G_1 u.$$

Finally, if υ is the last element of vector u_1, then $\bar{u}$ is defined so that u_1 is partitioned

$$u_1 = \begin{pmatrix} \bar{u} \\ \upsilon \end{pmatrix}$$

This gives $\bar{u} = \bar{Z}^T u'$, as required.

5.4 RH-B algorithms

We have implemented a family of RH-B algorithms in MATLAB and Fortran 90. This includes quasi-Wolfe line searches in both languages as well. Both versions are able to use the reinitialization technique and neither implement lingering. As was discussed in the beginning of this chapter, by default, reinitialization is enabled if $n > \min(6, m)$ and disabled otherwise. The Fortran 90 version only implements an implicit method, though the MATLAB version implements an implicit and explicit method. In addition to the standard RH-B algorithms, Algorithm RHSL-B (see Section 5.2) is implemented in MATLAB.

All RH-B methods are based on a limited-memory framework. A full-memory version can be done simply by setting $m = n$. This is not recommended, however, unless f and g are expensive to compute relative to the cost of the linear algebra.

A limited-memory implicit implementation of an RH-B algorithm with default reinitialization behavior is shown below.

Algorithm 5.2 Algorithm LRHB (implicit method)

Choose m, $\sigma > 0$ and x

$x \leftarrow P(x)$

$g \leftarrow \nabla f(x)$

$g_F \leftarrow P_x(g)$

$B' \leftarrow g$, $B \leftarrow g_F$, $T \leftarrow \|g_F\|$, $R \leftarrow \sqrt{\sigma}$, $v \leftarrow \|g_F\|$

while not converged

$d \leftarrow -R^{-T}v$

$q \leftarrow R^{-1}d$

$p \leftarrow BT^{-1}q$

if last projected gradient was accepted,

Swap last column of B' and B with p and update T

Compute quasi-Wolfe step α

$x_{\text{old}} \leftarrow x$

$x \leftarrow P(x + \alpha p)$, $g \leftarrow \nabla f(x)$, $g_F \leftarrow P_x(g)$

Update B, T, R, v, q if working set changed

$w \leftarrow T^{-T}B^T gF$

Update B', B, T, R, w, v, q if projected gradient is accepted

$s \leftarrow \alpha q$, $y \leftarrow w - v$

if $x - x_{\text{old}} \notin \text{col}(B)$

$y \leftarrow y + R^T R((T^{-T}B^T(x_{\text{old}} - x)) + s)$

Apply BFGS update to R if $y^T s > 0$

if $n > \min(6, m)$, compute new σ and reinitialize R

if $\text{rank}(B) > m$,

Drop oldest basis vector in B and update B', T, R, and w

$v \leftarrow w$

end

Adaptive-memory variants

Most of the work done in any of the RH-B algorithms usually takes place when the working set changes, especially if a large number of constraints are added

or dropped. Although this is not implemented yet, it would likely be advantageous to allow m to change according to stability of the working set: when $\mathcal{W}(x)$ changes dramatically, m should be kept small to avoid excessive work and when $\mathcal{W}(x)$ is more stable, m should be allowed to grow in order to speed convergence. The parameter m could change gradually or suddenly—if m is currently large and the working set changes dramatically, the procedure to drop columns could be called repeatedly if $r = m$ and we wish to reduce m by more than one. Alternatively, the algorithm could be restarted in such a case. Although a restart may be helpful if the algorithm wishes to drop many columns from B at once or if search directions are poorly chosen due to an inaccurate R, restarting a RH-B method destroys all curvature information built up within R and, hence, should be used sparingly.

5.5 Convergence results

Section 3.3.1 implies that an RH-B method can be viewed as a method that calculates a search direction p_k so that $p_k = -D_k g_k$, where D_k is a matrix that is diagonal with respect to $\mathcal{W}(x_k)$.

Let $\{x\}$ be a sequence generated by an RH-B method that uses an expanded working set, $\mathcal{E}(x)$, instead of $\mathcal{W}(x)$, where the Armijo condition associated with the step α is generalized to equation (3.11) (see page 44). If the conditions in Proposition 3.3.2 are satisfied, except that the sequence $\{x_k\}$ is generated as above, and $\{x_k\}$ converges to x^*, then, for some N,

$$\mathcal{E}(x_k) = \mathcal{A}(x_k) = \mathcal{A}(x^*),$$

for all $k > N$.

Once the active set stabilizes, given suitable conditions, the rate of convergence is Q-superlinear.

5.6 Future work

In addition to implementing an adaptive-memory variant for RH-B methods, we would like to extend the work done with RH-B methods to a class of

nonlinear-constrained optimization. If $\mathbb{F}$ is a convex feasible region where projection functions are easy to compute (i.e., an n-sphere, Cartesian products of n-spheres and box-constraints, etc), all of the material from Chapter 5 should generalize in a fairly straightforward way. From an implementation point-of-view, the two most-difficult aspect would likely be handling constraints that are not orthogonal to each other and handling constraints that are not orthogonal to columns of an identity matrix of order n.

Chapter 6

Numerical Results

In this chapter we describe the methods used to test a variety of optimization algorithms and the results obtained from such tests. Although the algorithms proposed in Chapters 4 and 5 and implemented in MATLAB and Fortran 90 are competitive, there are several variants that, once implemented, should lead to even better results. These variants include implementing an adaptive-memory method and an intelligent way to handle near singularity in the Cholesky factor R, or the upper-triangular factor T, e.g., restarting the algorithm when near-singularity is detected.

There are three implementations of a quasi-Wolfe line search. The first two are built on Fortran 90 and MATLAB implementations of a Wolfe line search used in SNOPT [GMS97], a general nonlinearly-constrained optimization method. The third is a direct modification of the Wolfe line search used in L-BFGS-B [BLNZ95] for use in LBFGSB-M.

There are three implementations of an RH-B method, as well. The first, Algorithm RHSL-B (see the end of Section 5.2), is implemented in MATLAB only and is provided for the sake of completeness, and as a way to compare the number of function calls that various RH-B methods make. The default behavior is to enable reinitialization if $n > 6$ and disable it otherwise. It is not intended to be a competitive algorithm, as many of the subroutines are not designed in a computationally efficient way.

The second RH-B method is implemented in Fortran 90. This method is a

limited-memory variant that is not based on any prior code, though it does use the Fortran 90 quasi-Wolfe line search described above, which is based on prior code. The implementation does not include lingering. The default behavior is to enable reinitialization if $n > \min(6, m)$ and disable it otherwise. Due to time constraints, only the implicit method is implemented. Further, the method recomputes q and v from scratch and is not able to update them as the working set changes (see Section 5.3). This is the most competitive implementation of an RH-B algorithm, given that it is written in Fortran 90. It is compiled using *g95* using optimization flag `-O3`. As all testing is done in MATLAB, this implementation is called using the mex interface.

The third RH-B method is implemented in MATLAB. This method is also a limited-memory variant that is not based on any prior code, barring the quasi-Wolfe line search implemented in MATLAB. Lingering is not implemented and reinitialization behavior is identical to that of the Fortran 90 implementation. This implementation includes the implicit and explicit method. When the working set changes, it is able to update q and v or recompute them from scratch. It uses the MATLAB function "rcond" to estimate the reciprocal condition number of R and T and restarts the algorithm if it is less than some tolerance. The number of function calls is competitive but the time needed to solve a problem is not, because of the inherent slowness of MATLAB. Both diagnostics described in Chapter 4 (counting the number of cusp steps and counting the times the approximate Hessian update fails) are done using this implementation.

In addition to the RH-B method implementations, we also test Fortran 77 implementations of L-BFGS-B and LBFGSB-M (see Section 4.4). Algorithm L-BFGS-B is written in Fortran 77 [ZBLN97], which is called from MATLAB using C++ wrappers written by Carbonetto [Car07]. Algorithm LBFGSB-M is based directly on L-BFGS-B and uses the same wrappers by Carbonetto. Mex interfaces for both algorithms are obtained with the command

```
mex -output algname *.cpp solver.f.
```

For convenience, Table 6.1 below summarizes the RH-B algorithms that are discussed in Chapter 6.

Table 6.1: RH-B algorithms

RHSB	Full-memory RH-B method (Section 5.2)
RHSLB	Algorithm RHSB, but with lingering
LRHrB	Limited-memory RH-B method; no reinitialization
LRHRB	Limited-memory RH-B method; with reinitialization
LRHB	Limited-memory RH-B method; conditional reinitialization

All tests were performed on an iMac with a 2.8 GHz Intel Core 2 Duo processor and 4 GB of 800 MHz DDR2 RAM running Mac OS X, version 10.5.8 (32 bit). All testing was done using MATLAB, version R2007b. An algorithm was considered to have successfully solved a problem if $\|P_{x_k}(g_k)\|_\infty < 10^{-5}$ (see page 37 for the definition of $P_{x_k}(g_k)$). If an algorithm reached iteration 1000 without meeting this condition, it was terminated and was considered to have failed to converge.

6.1 Test problem selection

All testing was done on problems taken from the CUTEr test set (see Gould, Orban, and Toint [GOT03]). The CUTEr test set is a collection of problems that range in constraint types, function types and number of variables. Many problems have user-defined parameters that determine the dimension of the problem. Some problems have a low number of variables that can be artificially increased through the parameters and others form families that are variations on a single problem. For this reason, the CUTEr test set does not necessarily represent a balanced collection of problems. It is also worth noting that, with the exception of problem *bleachng*, none of the problems that we tested had computationally expensive functions. For this reason, the numerical results below include information about the number of function calls needed, in addition to the time needed to solve the problem.

All testing was done on a set of problems that was formed from the preliminary list obtained by running the command

```
grep 'classification .BR.-..-*-*' *.SIF
```

within a directory containing all CUTEr problems. This provided a list of all 112 box-constrained problems that are twice-continuously differentiable. It is worth noting that, due to the classification system, unconstrained problems were not included in this list. The only problem that was removed from the final test set was problem *odnamur*, which caused MATLAB to crash when it loaded into memory.

On the remaining 111 problems, if given a choice, parameters were chosen to give the largest number of variables. The only parameters that were not chosen to maximize the number of variables were for the problems: *chardis0*, *hadamals*, *harkerp2*, *mccormck*, and *powellbc*. On these five problems, the parameter was chosen to give the largest number of variables such that MATLAB does not crash.

Tables 6.2–6.4 list the full test set. The second column represents the dimension of the problem. The third column represents the number of indices in the active set at the solution, as measured by L-BFGS-B and LRHB, both using $m = 5$, with the Fortran 90 LRHB implementation. If the active sets differed, the average was taken. If only one converged, only the number taken from the converging algorithm was used. If neither converged, the symbol "–" is displayed. The final column describes the parameters used on the problem, if any.

Table 6.2: Full test set (problems 1–37)

Problem	Dimension	$\mathcal{A}(x^*)$	Parameters
3pk	30	–	
allinit	4	1	
antwerp	27	–	$n = 5000$
bdexp	5000	0	$n = 5000$
biggsb1	5000	–	
bleachng	17	13	
bqp1var	1	1	
bqpgabim	50	14	
bqpgasim	50	10	
bqpgauss	2003	–	
camel6	2	0	$np1 = 200$
chardis0	400	0	$n = 100$
chebyqad	100	0	$ndegen = 500$
chenhark	5000	–	$n = 10000$
cvxbqp1	10000	10000	
deconvb	61	16	
eg1	3	1	$n = 1200$, $m = 100$
explin	1200	1149	$n = 1200$, $m = 100$
explin2	1200	1181	$n = 1200$, $m = 100$
expquad	1200	–	$n = 32$
hadamals	1024	63	$n = 100$
harkerp2	100	99	
hart6	6	0	
hatflda	4	0	
hatfldb	4	1	
hatfldc	25	0	
himmelp1	2	1	
hs1	2	0	$n = 200$
hs110	200	200	
hs2	2	1	
hs25	3	1	
hs3	2	1	
hs38	4	0	
hs3mod	2	1	
hs4	2	2	
hs45	5	5	
hs5	2	0	$pt = 125$, $py = 125$

Table 6.3: Full test set (problems 38–74)

Problem	Dimension	$\mathcal{A}(x^*)$	Parameters
jnlbrng1	12500	4332	$pt = 125$, $py = 125$
jnlbrng2	12500	5218	$pt = 125$, $py = 125$
jnlbrnga	12500	4556	$pt = 125$, $py = 125$
jnlbrngb	12500	–	
koebhelb	3	0	$n = 1000$
linverse	1999	594	
logros	2	0	
maxlika	8	1	$n = 10000$
mccormck	10000	1	
mdhole	2	1	
minsurfo	5306	518	$n = 10000$
ncvxbqp1	10000	10000	$n = 10000$
ncvxbqp2	10000	9935	$n = 10000$
ncvxbqp3	10000	9853	$q = 61$
nobndtor	14884	2813	$n = 10000$
nonscomp	10000	1	$px = 125$, $py = 125$
obstclae	12500	6308	$px = 125$, $py = 125$
obstclal	12500	6310	$px = 125$, $py = 125$
obstclbl	12500	3372	$px = 125$, $py = 125$
obstclbm	12500	3371	$px = 125$, $py = 125$
obstclbu	12500	3372	
oslbqp	8	7	
palmer1	4	0	
palmer1a	6	–	
palmer1b	4	0	
palmer1e	8	–	
palmer2	4	0	
palmer2a	6	0	
palmer2b	4	0	
palmer2e	8	–	
palmer3	4	0	
palmer3a	6	0	
palmer3b	4	0	
palmer3e	8	–	
palmer4	4	1	
palmer4a	6	0	
palmer4b	4	0	

Table 6.4: Full test set (problems 75–111)

Problem	Dimension	$\mathcal{A}(x^*)$	Parameters
palmer4e	8	–	
palmer5a	8	–	
palmer5b	9	–	
palmer5d	8	0	
palmer5e	8	–	
palmer6a	6	0	
palmer6e	8	–	
palmer7a	6	–	
palmer7e	8	–	
palmer8a	6	0	
palmer8e	8	–	$n = 5000$
pentdi	5000	4998	$p = 100$
powellbc	200	63	$n = 500$
probpenl	500	0	
pspdoc	4	1	$n = 5000$, $m = 1100$
qrtquad	5000	–	$n = 5000$, $m = 2500$
qudlin	5000	5000	$n = 100$
s368	100	39	$n = 5000$, $ln = 4500$
scond1ls	5002	–	
sim2bqp	2	2	
simbqp	2	1	$n = 1000$
sineali	1000	0	$k = 3$
specan	9	0	$q = 61$
torsion1	14884	4916	$q = 61$
torsion2	14884	4886	$q = 61$
torsion3	14884	9676	$q = 61$
torsion4	14884	9672	$q = 61$
torsion5	14884	12316	$q = 61$
torsion6	14884	12316	$q = 61$
torsiona	14884	4772	$q = 61$
torsionb	14884	4772	$q = 61$
torsionc	14884	9612	$q = 61$
torsiond	14884	9608	$q = 61$
torsione	14884	12284	$q = 61$
torsionf	14884	12284	
weeds	3	0	
yfit	3	0	

6.2 Explanation of Results

In all the full-page tables in the remainder of this chapter, the top row is used to describe the algorithms being tested and the values of any relevant optimization parameters used. To distinguish between implementations, we append the suffix "(f)" to an algorithm name to denote the Fortran 90 RH-B (implicit) method. We use the suffix "(m,i)" to signify an RH-B implicit method in MATLAB and "(m,e)" to signify an RH-B explicit method in MATLAB.

Each algorithm has two columns associated with it. The first, represented by the header entry "nfg," denotes the number of calls an algorithm made to a function to minimize it. If an algorithm failed to converge, the symbol "–" is displayed. The second, represented by "time," lists the number of seconds that elapsed for the algorithm to attempt to minimize a function, whether it converged or not. Time was measured using the MATLAB commands "tic" and "toc." The Fortran 90 LRHB implementation stalled on three problems. On these problems, the symbol "–" is displayed in the "time" column, instead.

Because the amount of time taken can vary from one run to another, all algorithms were run six times per problem unless otherwise noted. The first run was thrown away and the time listed was taken to be the average of the next five runs. If an algorithm failed to converge during the first run (or the second run in the case of LRHB(f), see Section 6.3.3), the time was taken to be the time for the first or second run and no further runs were done.

If a problem name (located in the left-most column) is marked with an asterisk, it signifies that at least two algorithms represented in that table converged to two different solutions. If the solutions are x_1 and x_2, we classify them as different if at least one of the following is true:

(a) $|f_1 - f_2| > 10^{-1}$; or

(b) $\|x_1 - x_2\| > 10^{-2}$.

If (a) is true, it suggests that x_1 and x_2 converged to two different local solutions. On the other hand, if (a) is false and (b) is true, the function evaluations were

close but the solutions were far apart. This suggests that the problem is very "flat" near x_1 and x_2.

Algorithms that failed to converge are not marked as converging to a different solution. Problems in which at least two algorithms converged to different solutions are not shown in the performance profiles (see below) unless otherwise noted.

Finally, to ensure an accurate testing environment, all algorithms that appear in the same table were tested at the same time. One side effect is that an algorithm may be listed in more than one table with different times displayed for each.

6.2.1 Performance Profiling

We use a method called *performance profiling*, developed by Dolan and Moré [DM02], to measure the relative effectiveness of various optimization routines. The method is designed to normalize the weight of each problem in a test set relative to the others, and to use information from problems even where one or more routines failed to converge, unlike using a simple sum over all (converging) problems.

Let $\mathcal{S}$ be a set of solvers (routines) that are used to minimize problems from the set $\mathcal{P}$. We define $\mathcal{S}_p = \{s \in \mathcal{S} : s \text{ successfully solves } p\}$. Let n_s be the number of solvers and n_p be the number of problems. Given a problem p and solver s, Dolan and Moré define

$$t_{p,s} = \text{computing time required to solve problem } p \text{ by solver } s.$$

We define a *performance ratio* to be

$$r_{p,s} = \begin{cases} \dfrac{t_{p,s}}{\min\{t_{p,s} : s \in \mathcal{S}_p\}} & \text{if } s \in S_p \\ r_M & \text{if } s \notin S_p, \end{cases}$$

where the parameter r_M is chosen so that,

$$(i) \quad r_M \geq r_{p,s} \text{ and } (ii) \quad r_M = r_{p,s} \iff s \notin S_p.$$

Note that a solver s solves problem p in the shortest amount of time relative to other solvers in $\mathcal{S}$ if and only if $r_{p,s} = 1$.

Dolan and Moré define a *performance profile for solver s* to be

$$\rho_s(\tau) = \frac{1}{n_p}|\{p \in P : r_{p,s} \leq \tau\}|,$$

so that $\rho_s(\tau)$ is the probability that a performance ratio $r_{p,s}$ is within a factor $\tau \in \mathbb{R}$ of the best possible ratio.

A performance profile $\rho_s(\tau)$ satisfies a number of properties. It is a monotonically-increasing function from $\mathbb{R}$ to the interval $[0, 1]$ that is piecewise constant and right continuous. The quantity $\rho_s(1)$ represents the probability that s solves p the fastest out of all solvers in $\mathcal{S}_p$, if p is chosen randomly from $\mathcal{P}$. The quantity $\rho_s(r_M)$ is one, and the quantity

$$\lim_{\tau \to r_M^-} \rho_s(\tau)$$

represents the probability that $s \in \mathcal{S}_p$, if p is chosen randomly from the set $\mathcal{P}$.

The description above is used to create a performance profile with respect to run time (computing time). The same approach can be used to create a performance profile with respect to the number of function calls needed by solver s to solve problem p. This is done by defining a new performance ratio $r_{p,s}$.

In order to capture behavior near $\tau = 1$ and behavior near $\tau = r_M$, we use a logarithmic scale along the horizontal axis. That is, we define

$$\pi_s(\tau) = \frac{1}{n_p}|\{p \in P : \log_2(r_{p,s}) \leq \tau\}|.$$

The quantity $\pi_s(0)$ represents the probability that s solves p the fastest out of all solvers in $\mathcal{S}_p$, if p is chosen randomly from $\mathcal{P}$. Speaking broadly, given s, the farther the graph $y = \pi_s(\tau)$ is to the left and up, the better the solver s is.

6.3 Numerical Results

6.3.1 Reinitialization

In this first set of tests, we verify that reinitialization tends to have a positive influence on the effectiveness of an RH-B algorithm when n is large and a negative

effect when n is small. To illustrate this, we tested the MATLAB-based RH-B methods in four cases:

- An implicit method, with $m = 5$, without reinitialization (LRHrB),
- An implicit method, with $m = 5$, with reinitialization (LRHRB),
- An explicit method, with $m = 20$, without reinitialization (LRHrB),
- An explicit method, with $m = 20$, with reinitialization (LRHRB).

None of the four methods tested include the default application of reinitialization, i.e., reinitialize if and only if $n > \min(6, m)$. Tables 6.5–6.7 list the number of function calls needed and time taken for each method. Problems with 4 or fewer variables are displayed in bold. No performance profiles are included with this test.

Table 6.5: Effects of reinitialization (Problems 1–37)

Problem	LRHrB(m,i) $m = 5$		LRHRB(m,i) $m = 5$		LRHrB(m,e) $m = 20$		LRHRB(m,e) $m = 20$	
	nfg	time	nfg	time	nfg	time	nfg	time
3pk	–	1.4117	–	0.9785	–	1.7742	–	0.6955
allinit	21	0.0200	23	0.0109	21	0.0090	23	0.0110
antwerp	–	1.5538	–	1.0186	–	1.5916	–	0.8044
*bdexp**	15	0.0605	15	0.0560	15	0.0496	15	0.0561
biggsb1	–	4.5322	–	3.7705	–	12.1946	–	11.7265
bleachng	11	14.9053	6	8.4171	11	14.9075	6	8.4043
bqp1var	2	0.0015	2	0.0011	2	0.0011	2	0.0011
bqpgabim	55	0.0423	30	0.0248	50	0.0294	23	0.0164
bqpgasim	49	0.0402	31	0.0262	72	0.0529	27	0.0210
bqpgauss	–	3.6514	–	2.6105	–	8.2290	–	7.3454
camel6	14	0.0151	15	0.0078	14	0.0064	15	0.0078
chardis0	4	0.0239	4	0.0127	4	0.0127	4	0.0126
chebyqad	–	45.8167	–	5.1107	–	45.6794	–	5.9952
chenhark	–	4.8145	–	3.7887	–	13.4231	–	12.7791
cvxbqp1	9	0.8653	11	0.8420	9	9.4132	9	9.1485
*deconvb**	324	0.2281	113	0.1206	274	0.3152	153	0.3478
eg1	10	0.0138	10	0.0076	10	0.0049	10	0.0053
*explin**	401	0.4698	271	0.4404	390	0.8744	208	0.7798
explin2	150	0.2471	87	0.2183	264	0.6124	105	0.4639
expquad	613	0.5782	524	0.7249	819	1.1172	603	1.2051
hadamals	12	0.0311	16	0.0305	10	0.0218	16	0.0303
harkerp2	17	0.0341	42	0.0515	16	0.0308	18	0.0333
hart6	40	0.0290	23	0.0156	26	0.0105	22	0.0112
hatflda	127	0.0766	98	0.0510	127	0.0644	98	0.0515
hatfldb	91	0.0499	70	0.0381	86	0.0388	76	0.0421
hatfldc	41	0.0301	24	0.0209	47	0.0249	24	0.0176
himmelp1	14	0.0134	14	0.0048	14	0.0048	14	0.0049
hs1	24	0.0202	989	0.4174	24	0.0117	1200	0.5119
hs110	2	0.0242	2	0.0141	2	0.0211	2	0.0211
hs2	27	0.0180	241	0.1023	27	0.0093	385	0.1628
hs25	1	0.0018	1	0.0003	1	0.0003	1	0.0003
hs3	9	0.0117	9	0.0031	9	0.0031	9	0.0031
hs38	47	0.0350	1151	0.5055	47	0.0243	1385	0.6126
hs3mod	10	0.0126	13	0.0046	10	0.0039	12	0.0044
hs4	2	0.0112	2	0.0011	2	0.0011	2	0.0012
hs45	4	0.0116	4	0.0016	4	0.0016	4	0.0017
hs5	8	0.0126	10	0.0055	8	0.0039	10	0.0055

Table 6.6: Effects of reinitialization (Problems 38–74)

Problem	LRHrB(m,i) $m = 5$		LRHRB(m,i) $m = 5$		LRHrB(m,e) $m = 20$		LRHRB(m,e) $m = 20$	
	nfg	time	nfg	time	nfg	time	nfg	time
jnlbrng1	645	6.7603	401	6.4021	807	84.4593	361	112.1789
jnlbrng2	1185	10.0195	660	7.3251	1178	24.4039	589	18.1493
jnlbrnga	877	6.9602	274	2.9187	637	16.7722	275	9.6021
jnlbrngb	–	15.5435	–	10.6159	–	35.4883	–	30.8003
koebhelb	225	0.1143	–	0.7186	225	0.1079	–	0.7503
*linverse**	1261	7.7335	399	4.3490	1824	254.7442	399	129.8082
logros	104	0.0570	–	0.6956	104	0.0490	–	0.7040
maxlika	753	0.6425	574	0.5624	95	0.0677	228	0.1571
mccormck	47	0.3019	23	0.1752	52	0.5224	26	0.2377
mdhole	82	0.0441	278	0.1124	81	0.0355	254	0.1070
minsurfo	547	2.3634	364	2.1333	539	4.4525	306	4.2052
ncvxbqp1	7	1.5655	7	1.5560	7	15.1032	7	15.1386
*ncvxbqp2**	161	2.2148	72	2.0229	282	18.9116	73	17.1342
*ncvxbqp3**	253	2.2869	150	2.1247	292	16.8766	86	15.1633
nobndtor	343	3.3928	196	2.7864	295	9.3001	177	17.6827
*nonscomp**	88	7.8912	29	0.8617	138	277.9882	30	202.0317
obstclae	322	4.8501	160	3.8402	263	118.0170	156	102.0850
obstclal	197	1.9479	102	1.3558	163	9.9384	102	9.5931
obstclbl	172	3.0010	115	2.6344	184	55.4091	100	51.2115
obstclbm	174	2.2900	103	2.0090	163	31.0176	96	34.6957
obstclbu	197	2.9832	110	2.0112	179	43.1151	98	41.0516
oslbqp	2	0.0109	2	0.0009	2	0.0008	2	0.0009
palmer1*	41	0.0245	37	0.0145	41	0.0160	37	0.0145
palmer1a	–	1.2820	–	0.9607	74	0.0346	–	0.6903
palmer1b	42	0.0287	808	0.3658	42	0.0204	–	0.6792
palmer1e	–	1.2637	–	0.9823	62	0.0344	–	0.7016
palmer2*	76	0.0376	113	0.0460	87	0.0364	110	0.0455
palmer2a	–	1.2713	480	0.3914	127	0.0614	–	0.6862
palmer2b	37	0.0254	–	0.6736	37	0.0170	–	0.6762
palmer2e	–	1.2515	–	0.9601	138	0.0747	–	0.6797
palmer3*	46	0.0294	49	0.0205	46	0.0211	49	0.0207
palmer3a	497	0.3438	674	0.5559	168	0.0796	–	0.6769
palmer3b	122	0.0636	315	0.1410	126	0.0572	436	0.1976
palmer3e	–	1.2511	–	0.9597	128	0.0674	–	0.6868
palmer4*	180	0.0878	48	0.0208	143	0.0633	48	0.0212
palmer4a	–	3.3635	598	0.4840	–	3.1611	–	0.6902
palmer4b	78	0.0427	182	0.0887	78	0.0345	329	0.1539

Table 6.7: Effects of reinitialization (Problems 75–111)

Problem	LRHrB(m,i) $m = 5$		LRHRB(m,i) $m = 5$		LRHrB(m,e) $m = 20$		LRHRB(m,e) $m = 20$	
	nfg	time	nfg	time	nfg	time	nfg	time
palmer4e	–	1.2459	–	0.9567	92	0.0497	–	0.6779
palmer5a	–	1.2649	–	0.9745	–	0.7263	–	0.6594
palmer5b	–	1.2759	–	0.9522	893	0.4411	–	0.6926
palmer5d	14	0.0163	1254	0.5666	14	0.0077	427	0.1963
palmer5e	–	1.2760	–	0.9594	–	0.7250	–	0.7237
palmer6a	775	0.5385	936	0.7522	219	0.1076	–	0.6641
palmer6e	–	1.2414	–	0.9594	97	0.0527	–	0.6865
palmer7a	17	0.0212	–	0.9174	–	0.7373	–	0.6763
palmer7e	–	1.2647	–	0.9828	697	0.3438	–	0.6842
palmer8a	1245	0.8558	383	0.2872	192	0.0896	–	0.6894
*palmer8e**	619	0.4428	–	0.9170	81	0.0453	–	0.6775
pentdi	4	0.0164	4	0.0076	4	0.0091	4	0.0091
*powellbc**	1969	1.9130	635	0.8021	2188	2.9328	631	1.4176
probpenl	4	0.0130	4	0.0022	4	0.0022	4	0.0022
pspdoc	11	0.0147	16	0.0090	11	0.0062	16	0.0091
qrtquad	–	5.5129	–	4.3805	–	13.6879	–	5.0876
*qudlin**	12	0.5244	11	0.5049	12	3.4085	11	3.3559
*s368**	37	0.1047	26	0.0704	48	0.1222	30	0.0877
scond1ls	–	9.2143	–	5.7165	–	17.0836	–	13.9444
sim2bqp	2	0.0112	2	0.0011	2	0.0010	2	0.0011
simbqp	5	0.0133	5	0.0023	5	0.0023	5	0.0023
*sineali**	137	0.1442	60	0.0812	222	0.3333	57	0.1164
specan	270	0.9062	190	0.6706	60	0.1943	82	0.2672
torsion1	253	3.1162	177	2.7509	233	38.6543	103	33.2246
torsion2	397	7.6901	211	5.4099	242	261.5487	170	243.3204
torsion3	165	2.0039	95	1.3994	132	14.1548	79	9.3448
torsion4	214	5.6693	128	4.7701	183	155.5217	91	165.7370
torsion5	88	0.8695	46	0.6307	66	1.1357	37	2.4965
torsion6	111	4.0930	65	3.5046	92	98.9366	56	85.7698
torsiona	322	4.1900	149	2.4769	176	22.3967	118	21.9066
torsionb	407	7.8278	229	6.5101	277	178.7796	222	141.8691
torsionc	154	1.9164	98	1.5249	129	9.1458	71	15.5911
torsiond	177	6.0905	127	5.2407	180	148.8514	86	158.4350
torsione	71	0.7316	41	0.5451	67	2.2188	39	2.7432
torsionf	107	4.1840	56	3.3263	87	98.2336	44	53.1282
weeds	54	0.0325	–	0.6741	54	0.0242	–	0.6630
yfit	84	0.0482	–	0.7012	84	0.0400	–	0.7051

6.3.2 RH-B methods in MATLAB

To give an idea of the strengths and weaknesses of the MATLAB-based RH-B methods, we tested four of them together:

- Algorithm LRHB, implicit method, $m = 5$,
- Algorithm LRHB, explicit method, $m = 5$,
- Algorithm LRHB, explicit method, $m = 50$,
- Algorithm RHSLB.

Summing over all problems where all solvers converged to the same solution, we obtain the following totals using Tables 6.9–6.11 (see following pages).

Table 6.8: RH-B methods in MATLAB (Sum Total)

Algorithm	nfg	Time (sec)	Failed
LRHB(m,i), $m = 5$	9104	82.91	21
LRHB(m,e), $m = 5$	9078	472.47	21
LRHB(m,e), $m = 50$	6316	2423.18	21
RHSLB	12802	151.98	16

We can make several observations from Tables 6.8–6.11, and Figures 6.1–6.2. In particular, Algorithm RHSLB and LRHB(m,i) are generally the fastest algorithms—RHSLB looks to be faster in Figure 6.1, but most of this is due to the higher percentage of solved cases and the use of fewer nested "for" loops that RHSLB achieves. It is also clear that RHSLB is least efficient with respect to function calls while LRHB (with $m = 50$) is the most efficient. We can also see that using an explicit method provides only minimal gain with respect to the number of function calls at the expense of a longer optimization time when m is small.

Table 6.9: RH-B methods in MATLAB (Problems 1–37)

Problem	LRHB(m,i) $m = 5$		LRHB(m,e) $m = 5$		LRHB(m,e) $m = 50$		RHSLB	
	nfg	time	nfg	time	nfg	time	nfg	time
3pk	–	1.2510	–	1.0836	–	0.7041	–	0.8071
allinit	21	0.0201	21	0.0091	21	0.0091	21	0.0147
antwerp	–	1.3215	–	1.0199	–	0.8019	–	0.1611
*bdexp**	15	0.0632	15	0.0700	15	0.0530	17	0.0600
biggsb1	–	3.8623	–	4.8817	–	26.7136	–	78.8636
bleachng	6	8.4251	6	8.4125	6	8.4029	9	12.3311
bqp1var	2	0.0016	2	0.0011	2	0.0011	2	0.0033
bqpgabim	30	0.0350	30	0.0270	23	0.0166	29	0.0216
bqpgasim	31	0.0388	31	0.0285	27	0.0192	33	0.0254
bqpgauss	–	2.6474	–	3.2819	–	14.8369	–	2.0896
camel6	14	0.0152	14	0.0064	14	0.0064	14	0.0122
chardis0	4	0.0242	4	0.0127	4	0.0126	4	0.0189
chebyqad	–	5.3573	1004	4.9482	–	7.8629	776	4.2806
chenhark	–	3.7726	–	5.9520	–	29.8271	–	50.0598
cvxbqp1	11	0.8540	9	8.7395	9	8.5684	5	0.0260
*deconvb**	113	0.1294	113	0.1352	151	0.4680	462	0.3156
eg1	10	0.0136	10	0.0049	10	0.0049	9	0.0119
*explin**	271	0.4257	239	0.5525	217	1.3044	320	0.3160
explin2	87	0.2190	75	0.3282	129	0.4512	147	0.1169
expquad	524	0.7466	662	0.9946	481	0.6225	671	0.5840
hadamals	16	0.0396	16	0.0304	16	0.0305	25	0.0432
harkerp2	42	0.0607	29	0.0513	18	0.0335	71	0.0428
hart6	23	0.0244	23	0.0165	26	0.0106	30	0.0184
hatflda	127	0.0771	127	0.0647	127	0.0646	58	0.0358
hatfldb	91	0.0504	86	0.0393	86	0.0392	55	0.0332
hatfldc	24	0.0300	24	0.0199	24	0.0161	32	0.0254
himmelp1	14	0.0135	14	0.0048	14	0.0049	14	0.0110
hs1	24	0.0203	24	0.0118	24	0.0118	24	0.0174
hs110	2	0.0245	2	0.0214	2	0.0214	2	0.0076
hs2	27	0.0181	27	0.0094	27	0.0094	27	0.0149
hs25	1	0.0018	1	0.0003	1	0.0003	1	0.0059
hs3	9	0.0118	9	0.0031	9	0.0031	9	0.0088
hs38	47	0.0351	47	0.0243	47	0.0244	47	0.0298
hs3mod	10	0.0128	10	0.0040	10	0.0040	10	0.0095
hs4	2	0.0113	2	0.0011	2	0.0012	2	0.0074
hs45	4	0.0117	4	0.0016	4	0.0017	4	0.0078
hs5	8	0.0126	8	0.0040	8	0.0040	8	0.0096

Table 6.10: RH-B methods in MATLAB (Problems 38–74)

Problem	LRHB(m,i) $m = 5$		LRHB(m,e) $m = 5$		LRHB(m,e) $m = 50$		RHSLB	
	nfg	time	nfg	time	nfg	time	nfg	time
jnlbrng1	401	6.5789	402	34.4906	385	278.2951	–	10.9944
jnlbrng2	660	7.5114	599	8.6981	545	35.0408	879	11.1295
jnlbrnga	274	3.0307	285	5.2304	279	29.1972	841	8.4151
jnlbrngb	–	11.2684	–	14.3636	–	67.0133	–	11.8498
koebhelb	225	0.1145	225	0.1081	225	0.1083	211	0.1081
*linverse**	399	4.3964	413	21.9481	327	326.7529	–	2.5575
logros	104	0.0573	104	0.0492	104	0.0492	104	0.0543
maxlika	574	0.5749	794	0.8078	228	0.1581	247	0.1796
mccormck	23	0.1913	23	0.2265	26	0.2348	23	0.1734
mdhole	82	0.0441	81	0.0356	81	0.0356	82	0.0412
minsurfo	364	2.2614	392	3.0272	503	11.0020	675	39.6562
ncvxbqp1	7	1.5784	7	16.8360	7	16.8539	7	0.0379
*ncvxbqp2**	72	2.0718	72	18.2701	70	19.1208	104	0.4452
*ncvxbqp3**	150	2.2089	155	15.9895	73	17.1137	157	0.6845
nobndtor	196	2.9246	195	7.5756	174	39.0531	400	4.2823
*nonscomp**	29	0.8938	29	22.6322	32	278.2514	45	0.2503
obstclae	160	3.8033	162	27.8334	166	347.0124	741	6.5923
obstclal	102	1.3634	102	3.9991	102	15.0837	225	1.9617
obstclbl	115	2.6517	115	23.2306	100	76.5024	502	4.0530
obstclbm	103	2.0725	103	15.9116	103	52.1298	289	2.4378
obstclbu	110	2.0808	110	15.3836	102	56.0347	360	2.9475
oslbqp	2	0.0110	2	0.0008	2	0.0009	2	0.0074
*palmer1**	41	0.0246	41	0.0161	41	0.0161	41	0.0221
palmer1a	–	1.2163	–	1.0556	74	0.0351	76	0.0409
palmer1b	42	0.0289	42	0.0204	42	0.0204	39	0.0241
palmer1e	–	1.2357	–	1.0649	–	0.6893	223	0.1242
*palmer2**	76	0.0378	87	0.0365	87	0.0365	31	0.0204
palmer2a	480	0.4023	973	0.8650	127	0.0615	128	0.0671
palmer2b	37	0.0256	37	0.0171	37	0.0171	40	0.0236
palmer2e	–	1.2120	–	1.0511	–	0.6825	201	0.1085
*palmer3**	46	0.0297	46	0.0213	46	0.0213	54	0.0308
palmer3a	674	0.5690	549	0.4792	168	0.0796	164	0.0852
palmer3b	122	0.0638	126	0.0574	126	0.0573	78	0.0425
palmer3e	–	1.2109	–	1.0522	–	0.6876	171	0.0974
*palmer4**	180	0.0877	143	0.0632	143	0.0631	54	0.0310
palmer4a	598	0.4961	687	0.5917	–	3.1710	–	0.0942
palmer4b	78	0.0428	78	0.0345	78	0.0345	79	0.0430

Table 6.11: RH-B methods in MATLAB (Problems 75–111)

Problem	LRHB(m,i) $m = 5$		LRHB(m,e) $m = 5$		LRHB(m,e) $m = 50$		RHSLB	
	nfg	time	nfg	time	nfg	time	nfg	time
palmer4e	–	1.2092	–	1.0479	–	0.6791	138	0.0802
palmer5a	–	1.2254	–	1.0518	–	0.6597	–	0.1124
palmer5b	–	1.2052	–	1.0463	–	0.6929	270	0.1476
palmer5d	1254	0.5784	427	0.1975	427	0.1981	22	0.0174
palmer5e	–	1.2156	–	1.0576	–	0.7291	–	0.7485
palmer6a	936	0.7645	–	1.0442	219	0.1081	222	0.1143
palmer6e	–	1.2132	–	1.0499	–	0.6897	166	0.0948
palmer7a	–	1.2043	–	1.0473	–	0.6938	–	0.0933
palmer7e	–	1.2343	–	1.0634	–	0.6752	–	0.7316
palmer8a	383	0.2973	505	0.4241	192	0.0898	117	0.0614
palmer8e	–	1.2290	–	1.0609	–	0.6926	110	0.0647
pentdi	4	0.0170	4	0.0101	4	0.0100	5	0.0159
*powellbc**	635	0.8152	925	1.2490	558	2.6354	–	0.1694
probpenl	4	0.0130	4	0.0022	4	0.0022	4	0.0079
pspdoc	11	0.0148	11	0.0062	11	0.0062	11	0.0118
qrtquad	–	5.1077	–	6.6562	–	6.3330	–	0.8471
*qudlin**	11	0.5199	11	4.0586	11	4.0553	10	0.0325
*s368**	26	0.0809	26	0.0751	30	0.0843	21	0.0576
scond1ls	–	6.3469	–	7.7594	–	30.9209	–	42.8666
sim2bqp	2	0.0113	2	0.0010	2	0.0011	2	0.0074
simbqp	5	0.0134	5	0.0023	5	0.0023	6	0.0082
*sineali**	60	0.0948	60	0.0934	60	0.1054	83	0.1357
specan	190	0.6803	198	0.7167	82	0.2691	36	0.1262
torsion1	177	2.8578	167	10.1151	98	50.4656	365	3.8645
torsion2	211	5.5237	232	45.2581	136	511.0080	894	9.2958
torsion3	95	1.4441	95	3.9395	68	12.0678	173	1.9552
torsion4	128	4.8412	128	52.2369	88	278.2089	557	5.6152
torsion5	46	0.6579	46	1.8326	37	2.4831	79	0.9740
torsion6	65	3.5616	65	42.5975	57	103.0646	294	2.9179
torsiona	149	2.5587	149	9.2392	97	37.0632	419	5.2731
torsionb	229	6.6760	209	56.4760	189	371.6227	1134	13.0725
torsionc	98	1.5798	98	4.2102	69	17.6156	190	2.5443
torsiond	127	5.3441	131	54.2231	86	277.4768	472	5.1872
torsione	41	0.5709	41	0.8393	39	2.9091	92	1.1591
torsionf	56	3.3675	56	40.8027	45	61.4128	367	3.9654
*weeds**	54	0.0328	54	0.0244	54	0.0245	8	0.0096
yfit	84	0.0484	84	0.0403	84	0.0403	84	0.0457

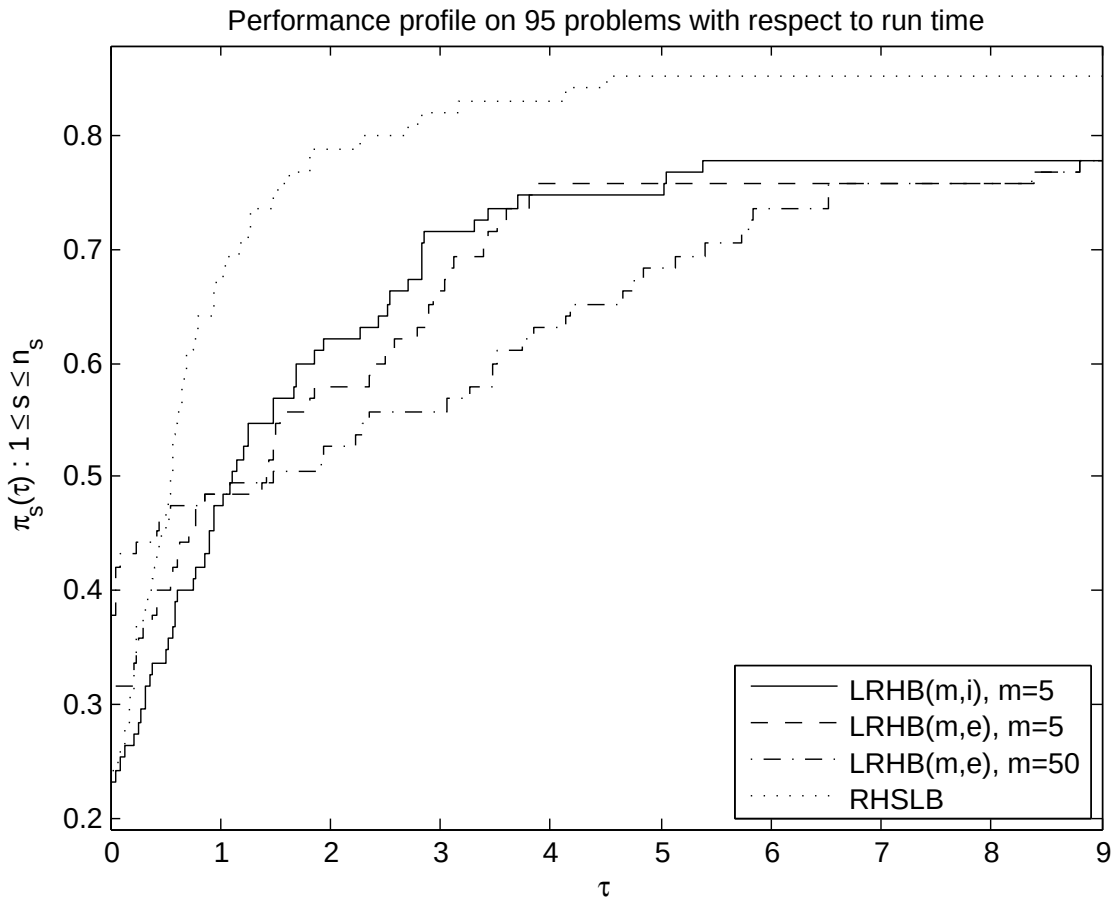

Figure 6.1: Performance profile for RH-B methods in MATLAB (time)

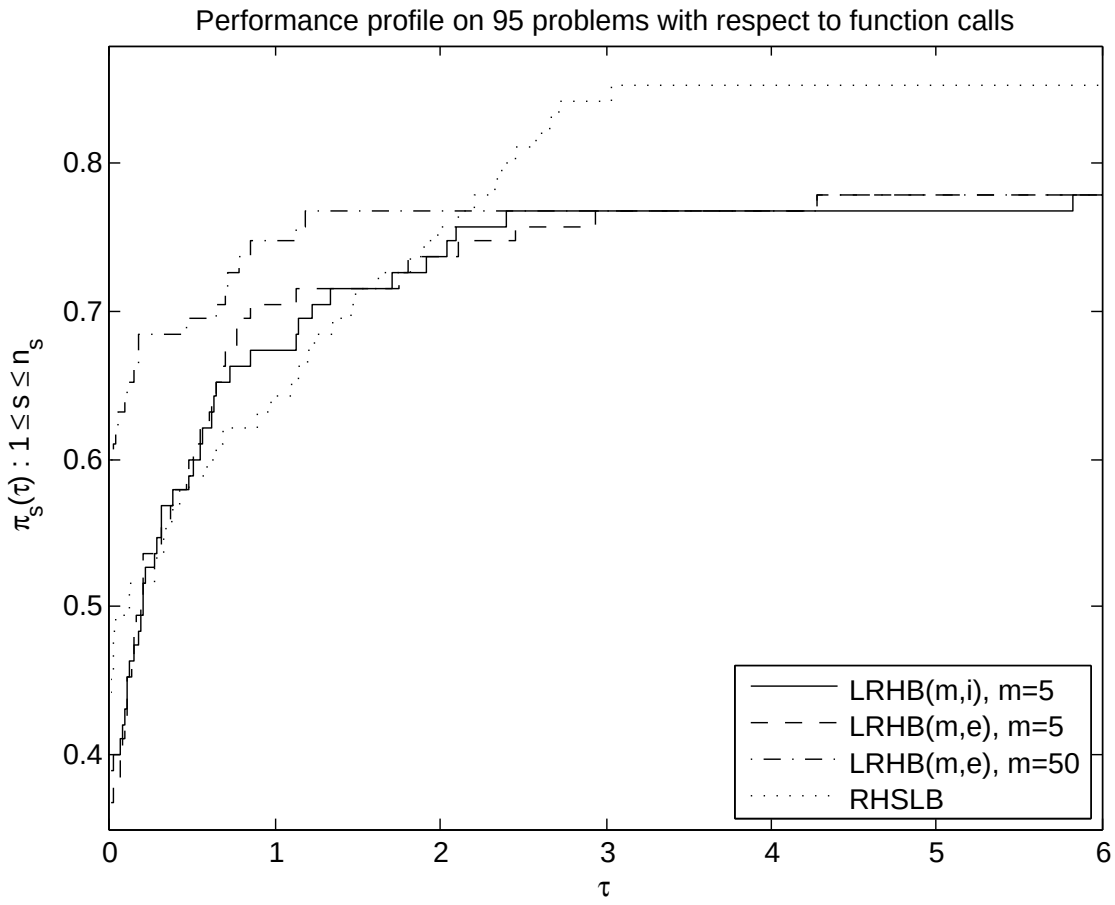

Figure 6.2: Performance profile for RH-B methods in MATLAB (nfg)

6.3.3 Known bugs and issues

A known bug with the Fortran 90 RH-B implementation is that, on occasion, the number of iterations and function calls needed is different for the first run than for all subsequent runs. The small numerical rounding errors introduced into the values of variables on the first run appear to be different than the rounding errors on later runs, even when input is identical. Since many of the problems are ill-conditioned near solutions, these small differences propagate and can occasionally cause dramatic differences in the number of functions called or, in some cases, even convergence before 1000 iterations. The issue seems to begin with a call to the BLAS routine DNRM2 on this machine. It is not known at this time if the error is machine specific. There is no variance detected in the variables on any given problem on subsequent calls after the first call has been made.

In order to demonstrate this behavior, one run (instead of six) was made per algorithm per problem in the following test. The first two algorithms displayed in Tables 6.12–6.14 are the same Fortran 90 implementation of an RH-B, with identical parameters. Although this bug influences the time it took to run each algorithm, there is also computational overhead involved when any optimization routine is first invoked on a new problem. The third algorithm displayed is the corresponding MATLAB implementation, also with identical parameters. The performance difference between the MATLAB and Fortran 90 versions can be explained by slight differences in the underlying code as well as error propagation caused by using inexact arithmetic. As mentioned before, the MATLAB implementation is slightly more developed than the Fortran 90 version, though this is primarily due to time constraints. No performance profile for time is given, since it does not give an accurate measure of actual performance under the circumstances.

Tables 6.12–6.14 and Figure 6.3 show that, although the Fortran implementation gave different results on the first two runs, they were fairly comparable overall. The marginally higher convergence percentage that the MATLAB implementation exhibited is likely due to its slightly more advanced code.

Table 6.12: Implicit implementations of LRHB (Problems 1–37)

Problem	LRHB(f) $m = 5$		LRHB(f) $m = 5$		LRHB(m,i) $m = 5$	
	nfg	time	nfg	time	nfg	time
3pk	–	0.1424	–	0.1252	–	1.2454
allinit	21	0.0061	21	0.0021	21	0.2720
antwerp	–	0.1908	–	0.1824	–	1.3167
bdexp	15	0.0413	15	0.0387	15	0.3288
biggsb1	–	2.3733	–	2.3638	–	4.3782
bleachng	6	8.5763	6	8.4470	6	8.6785
bqp1var	2	0.0036	2	0.0003	2	0.1794
bqpgabim	30	0.0074	30	0.0038	30	0.3411
bqpgasim	31	0.0078	31	0.0052	31	0.3316
bqpgauss	–	1.4575	–	1.4647	–	3.0363
camel6	14	0.0052	14	0.0014	14	0.2715
chardis0	4	0.0176	4	0.0125	4	0.2704
chebyqad	1018	4.1665	1021	4.1684	–	5.3287
chenhark	–	2.4796	–	2.4702	–	4.2289
cvxbqp1	10	0.0420	10	0.0373	11	1.1200
deconvb	113	0.0209	113	0.0169	113	0.4418
eg1	10	0.0049	10	0.0012	10	0.2780
*explin**	280	0.1466	254	0.1380	271	0.7765
explin2	87	0.0519	87	0.0497	87	0.5436
expquad	–	–	–	–	524	1.1424
hadamals	16	0.0248	16	0.0195	16	0.3276
harkerp2	26	0.0082	28	0.0049	42	0.3541
hart6	23	0.0062	23	0.0025	23	0.3029
hatflda	127	0.0157	127	0.0136	127	0.3323
hatfldb	–	–	–	–	91	0.3664
hatfldc	24	0.0065	24	0.0053	24	0.3064
himmelp1	14	0.0051	14	0.0016	14	0.2602
hs1	24	0.0061	24	0.0023	24	0.2889
hs110	1	0.0043	1	0.0006	2	0.2313
hs2	27	0.0064	27	0.0051	27	0.3182
hs25	1	0.0036	1	0.0004	1	0.0542
hs3	9	0.0048	9	0.0013	9	0.2583
hs38	47	0.0085	47	0.0047	47	0.2912
hs3mod	10	0.0052	10	0.0013	10	0.3040
hs4	2	0.0042	2	0.0003	2	0.2167
hs45	4	0.0042	4	0.0015	4	0.2198
hs5	8	0.0046	8	0.0011	8	0.2683

Table 6.13: Implicit implementations of LRHB (Problems 38–74)

Problem	LRHB(f) $m = 5$		LRHB(f) $m = 5$		LRHB(m,i) $m = 5$	
	nfg	time	nfg	time	nfg	time
jnlbrng1	407	4.1868	423	4.3397	401	7.0459
jnlbrng2	661	6.6109	728	7.1495	660	7.9985
jnlbrnga	288	2.6931	306	2.8510	274	3.4086
jnlbrngb	–	9.6333	–	9.6534	–	11.4474
koebhelb	216	0.0310	212	0.0278	225	0.3878
linverse	425	0.7674	411	0.7429	399	4.8888
logros	104	0.0136	104	0.0097	104	0.3143
maxlika	764	0.2258	591	0.1653	574	0.8962
mccormck	23	0.1521	23	0.1457	23	0.4926
mdhole	81	0.0113	81	0.0077	82	0.3438
minsurfo	365	1.6328	318	1.4006	364	2.6558
ncvxbqp1	7	0.0426	7	0.0376	7	1.8649
ncvxbqp2	72	0.3442	72	0.3407	72	2.3804
ncvxbqp3	144	0.6709	145	0.6768	150	2.5269
nobndtor	194	2.1877	194	2.1752	196	3.2249
nonscomp	29	0.1393	29	0.1347	29	1.2207
obstclae	163	1.5815	160	1.5450	160	4.1383
obstclal	102	0.9681	102	0.9691	102	1.7224
obstclbl	115	1.1110	115	1.1069	115	3.0042
obstclbm	103	0.9927	103	0.9933	103	2.4178
obstclbu	110	1.0606	110	1.0518	110	2.4336
oslbqp	2	0.0044	2	0.0003	2	0.1946
palmer1	41	0.0079	41	0.0041	41	0.2975
palmer1a	–	0.1356	–	0.1282	–	1.2310
palmer1b	42	0.0085	42	0.0051	42	0.2901
palmer1e	–	0.1377	–	0.1343	–	1.2602
*palmer2**	72	0.0107	72	0.0069	76	0.3544
palmer2a	620	0.0718	835	0.0932	480	0.7005
palmer2b	37	0.0074	37	0.0049	37	0.2867
palmer2e	–	0.1372	–	0.1280	–	1.2359
palmer3	46	0.0085	46	0.0046	46	0.3285
palmer3a	612	0.0715	–	0.1265	674	0.8696
palmer3b	129	0.0166	123	0.0118	122	0.3498
palmer3e	–	0.1332	–	0.1295	–	1.2287
*palmer4**	182	0.0215	180	0.0182	180	0.4052
palmer4a	503	0.0585	358	0.0399	598	0.7947
palmer4b	77	0.0113	77	0.0075	78	0.3372

Table 6.14: Implicit implementations of LRHB (Problems 75–111)

Problem	LRHB(f) $m = 5$		LRHB(f) $m = 5$		LRHB(m,i) $m = 5$	
	nfg	time	nfg	time	nfg	time
palmer4e	–	0.1330	–	0.1324	–	1.2416
palmer5a	–	0.1368	–	0.1324	–	1.2446
palmer5b	–	0.1279	–	0.1235	–	1.2220
palmer5d	1404	0.1516	577	0.0558	1254	0.8565
palmer5e	–	0.1334	–	0.1274	–	1.2401
palmer6a	–	0.1384	–	0.1281	936	1.0715
palmer6e	–	0.1307	–	0.1261	–	1.2312
palmer7a	–	0.1275	–	0.1273	–	1.2199
palmer7e	–	0.1302	–	0.1271	–	1.2497
*palmer8a**	748	0.0849	558	0.0584	383	0.6091
palmer8e	–	0.1305	–	0.1311	–	1.2603
pentdi	4	0.0121	4	0.0087	4	0.2685
*powellbc**	737	0.3944	773	0.4024	635	1.1399
probpenl	4	0.0050	4	0.0011	4	0.2583
pspdoc	11	0.0050	11	0.0013	11	0.2635
qrtquad	–	–	–	–	–	5.2863
*qudlin**	11	0.0194	11	0.0156	11	0.7888
s368	26	0.0580	26	0.0526	26	0.3908
scond1ls	–	4.8679	–	4.7358	–	6.5657
sim2bqp	2	0.0041	2	0.0003	2	0.2128
simbqp	5	0.0043	5	0.0007	5	0.2597
sineali	60	0.0435	60	0.0395	60	0.3926
specan	189	0.5546	211	0.6051	190	1.0039
torsion1	167	1.9045	165	1.8756	177	3.1939
torsion2	212	2.4513	208	2.3932	211	5.9096
torsion3	95	1.0684	95	1.0636	95	1.7670
torsion4	128	1.4631	128	1.4562	128	5.2179
torsion5	46	0.4971	46	0.4923	46	0.9616
torsion6	65	0.7454	65	0.7366	65	3.9030
torsiona	151	1.8267	149	1.7983	149	2.8878
torsionb	202	2.4896	232	2.8438	229	7.0656
torsionc	98	1.1970	98	1.1910	98	1.9153
torsiond	131	1.6506	129	1.6097	127	5.7448
torsione	41	0.4912	41	0.4816	41	0.8816
torsionf	56	0.6808	56	0.6746	56	3.7592
weeds	54	0.0106	54	0.0051	54	0.3000
yfit	84	0.0121	84	0.0083	84	0.3260

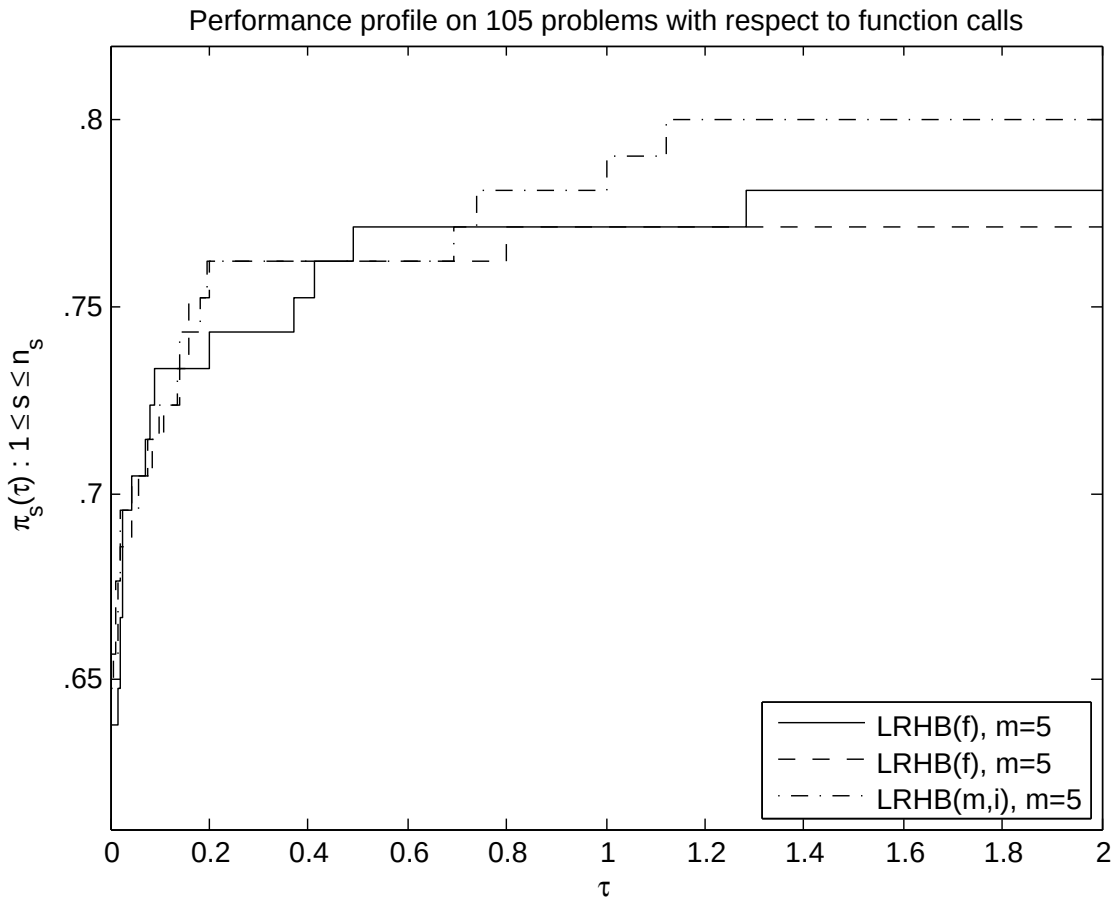

Figure 6.3: Performance profile for implicit implementations of LRHB (nfg)

6.3.4 Competitive algorithms

In order to compare how effective RH-B methods are against other algorithms, we tested three, including Algorithm LBFGSB-M, which uses a quasi-Wolfe line search:

- Algorithm LBFGSB, $m = 5$,
- Algorithm LBFGSB-M, $m = 5$,
- Algorithm LRHB, Fortran implementation, $m = 5$.

Summing over all problems where all solvers converged to the same solution, we get the following totals using Tables 6.16–6.17 (see following pages).

Table 6.15: Comparison of competitive algorithms (Sum Total)

Algorithm	nfg	Time (sec)	Failed
LBFGSB, $m = 5$	9002	54.83	35
LBFGSB-M, $m = 5$	7716	46.14	34
LRHB(f), $m = 5$	6999	40.57	24

Based on Table 6.15, we see that LRHB minimized the set of test problems where all three algorithms converge in less than 74% of the time that LBFGSB took. LRHB also needed less than 78% of the function calls that LBFGSB needed.

In addition to the two performance profiles (Figures 6.4 and 6.5) that include all problems such that all converging algorithms converged to the same solution, for completeness, two performance profiles (Figures 6.6 and 6.7) are shown that include the full test set.

It is clear from Tables 6.15–6.18 and Figures 6.4–6.7 that LRHB generally outperformed LBFGSB and LBFGSB-M in terms of time taken, function calls needed, and percentage of problems that were successfully solved. Using the same tables and figures, it is clear that replacing the line search used in LBFGSB with a quasi-Wolfe line search (as in LBFGSB-M) also resulted in increased performance with respect to time and function calls.

Table 6.16: Comparison of competitive algorithms (Problems 1–37)

Problem	LBFGSB $m = 5$		LBFGSB-M $m = 5$		LRHB(f) $m = 5$	
	nfg	time	nfg	time	nfg	time
3pk	–	0.2129	–	0.2321	–	0.1251
allinit	19	0.0035	19	0.0032	21	0.0019
antwerp	–	0.2227	–	0.2177	–	0.1813
*bdexp**	15	0.0331	15	0.0340	15	0.0379
biggsb1	–	1.2690	–	1.3111	–	2.3659
bleachng	–	67.8912	–	83.1788	6	8.4512
bqp1var	2	0.0003	2	0.0003	2	0.0002
bqpgabim	23	0.0046	23	0.0046	30	0.0036
bqpgasim	25	0.0048	25	0.0048	31	0.0036
bqpgauss	–	1.2131	–	1.2138	–	1.4394
camel6	14	0.0024	14	0.0023	14	0.0013
*chardis0**	4	0.0117	4	0.0118	4	0.0114
chebyqad	–	0.0198	–	0.0149	1021	4.1788
chenhark	–	1.5590	–	1.6026	–	2.4473
cvxbqp1	2	0.0069	2	0.0078	10	0.0361
*deconvb**	210	0.0477	136	0.0307	113	0.0165
eg1	14	0.0024	15	0.0025	10	0.0009
explin	–	0.0563	–	0.0572	254	0.1362
explin2	–	0.0254	–	0.0734	87	0.0464
expquad	–	0.0073	–	0.0033	–	–
hadamals	24	0.0308	26	0.0337	16	0.0191
harkerp2	31	0.0075	32	0.0078	28	0.0046
hart6	19	0.0034	19	0.0035	23	0.0023
hatflda	39	0.0065	39	0.0067	127	0.0116
hatfldb	34	0.0056	32	0.0054	–	–
hatfldc	23	0.0042	23	0.0042	24	0.0026
himmelp1	12	0.0019	11	0.0017	14	0.0012
hs1	29	0.0047	29	0.0048	24	0.0022
hs110	2	0.0006	3	0.0007	1	0.0005
hs2	19	0.0030	21	0.0031	27	0.0024
hs25	1	0.0003	1	0.0002	1	0.0002
hs3	4	0.0008	5	0.0008	9	0.0008
hs38	26	0.0046	26	0.0046	47	0.0043
hs3mod	9	0.0014	13	0.0019	10	0.0009
hs4	2	0.0004	3	0.0005	2	0.0002
hs45	11	0.0016	5	0.0007	4	0.0004
hs5	8	0.0014	8	0.0014	8	0.0008

Table 6.17: Comparison of competitive algorithms (Problems 38–74)

Problem	LBFGSB $m = 5$		LBFGSB-M $m = 5$		LRHB(f) $m = 5$	
	nfg	time	nfg	time	nfg	time
jnlbrng1	913	7.5231	641	5.2212	423	4.3701
jnlbrng2	574	4.4512	558	4.3707	728	7.1763
jnlbrnga	341	2.3823	293	2.0813	306	2.8579
jnlbrngb	–	7.1282	–	7.2592	–	9.6754
koebhelb	–	0.0066	–	0.0036	212	0.0262
*linverse**	336	0.4593	332	0.4662	411	0.7370
logros	120	0.0199	111	0.0185	104	0.0094
maxlika	–	0.3748	–	0.3792	591	0.1651
mccormck	15	0.0839	15	0.0853	23	0.1458
mdhole	85	0.0144	85	0.0143	81	0.0072
minsurfo	326	1.2027	338	1.2430	318	1.3975
ncvxbqp1	2	0.0071	3	0.0093	7	0.0369
ncvxbqp2	–	0.3645	–	0.3104	72	0.3398
ncvxbqp3	–	0.8392	–	0.7918	145	0.6724
nobndtor	225	2.0512	212	1.9370	194	2.1223
*nonscomp**	51	0.1912	51	0.1944	29	0.1283
obstclae	668	4.7566	502	3.5620	160	1.5003
obstclal	162	1.0631	157	1.0486	102	0.9386
obstclbl	261	1.8963	249	1.8166	115	1.0615
obstclbm	118	0.8522	121	0.8737	103	0.9472
obstclbu	154	1.0945	151	1.0757	110	1.0093
oslbqp	2	0.0004	2	0.0004	2	0.0002
palmer1	–	0.0113	–	0.0439	41	0.0038
palmer1a	–	0.2157	600	0.1070	–	0.1259
palmer1b	78	0.0138	–	0.0499	42	0.0040
palmer1e	–	0.2491	–	0.2509	–	0.1303
palmer2	–	0.0131	–	0.0113	72	0.0066
palmer2a	553	0.0990	554	0.0989	835	0.0918
palmer2b	62	0.0103	60	0.0099	37	0.0034
palmer2e	–	0.2278	–	0.2371	–	0.1234
palmer3	–	0.0491	–	0.0136	46	0.0043
palmer3a	760	0.1373	493	0.0885	–	0.1243
palmer3b	57	0.0095	61	0.0100	123	0.0114
palmer3e	–	0.2261	–	0.2226	–	0.1258
*palmer4**	25	0.0039	25	0.0039	180	0.0168
palmer4a	611	0.1107	381	0.0678	358	0.0375
palmer4b	69	0.0116	69	0.0117	77	0.0071

Table 6.18: Comparison of competitive algorithms (Problems 75–111)

Problem	LBFGSB $m = 5$		LBFGSB-M $m = 5$		LRHB(f) $m = 5$	
	nfg	time	nfg	time	nfg	time
palmer4e	–	0.2231	–	0.2219	–	0.1264
palmer5a	–	0.2275	–	0.2221	–	0.1290
palmer5b	–	0.2093	–	0.2090	–	0.1215
palmer5d	24	0.0039	24	0.0040	577	0.0545
palmer5e	–	0.2253	–	0.2228	–	0.1254
palmer6a	985	0.1734	576	0.1009	–	0.1225
palmer6e	–	0.2261	–	0.2383	–	0.1233
palmer7a	–	0.2238	–	0.2203	–	0.1237
palmer7e	–	0.2245	–	0.2201	–	0.1242
*palmer8a**	405	0.0718	234	0.0410	558	0.0573
palmer8e	–	0.2438	–	0.2139	–	0.1241
pentdi	3	0.0046	3	0.0049	4	0.0062
powellbc	–	0.0147	–	0.0108	773	0.3902
probpenl	3	0.0009	3	0.0009	4	0.0009
pspdoc	11	0.0020	11	0.0020	11	0.0010
qrtquad	–	2.8505	–	2.3754	–	–
*qudlin**	2	0.0035	3	0.0044	11	0.0142
*s368**	21	0.0444	16	0.0337	26	0.0510
scond1ls	–	3.6306	–	3.7331	–	4.5519
sim2bqp	2	0.0004	2	0.0004	2	0.0002
simbqp	5	0.0008	5	0.0008	5	0.0005
sineali	–	0.0206	–	0.0272	60	0.0372
specan	157	0.4592	152	0.4600	211	0.6038
torsion1	198	1.6654	185	1.5611	165	1.8180
torsion2	497	4.4410	313	2.7168	208	2.3171
torsion3	76	0.5865	76	0.5957	95	1.0319
torsion4	420	3.4901	268	2.1807	128	1.4136
torsion5	40	0.2925	40	0.2984	46	0.4782
torsion6	341	2.7342	308	2.4881	65	0.7116
torsiona	203	1.9165	199	1.8759	149	1.7392
torsionb	398	3.9027	363	3.5590	232	2.7660
torsionc	89	0.7712	89	0.7825	98	1.1374
torsiond	375	3.4416	353	3.2817	129	1.5375
torsione	38	0.3079	38	0.3134	41	0.4586
torsionf	341	3.0509	264	2.3449	56	0.6479
weeds	–	0.0203	47	0.0079	54	0.0049
yfit	105	0.0182	93	0.0161	84	0.0079

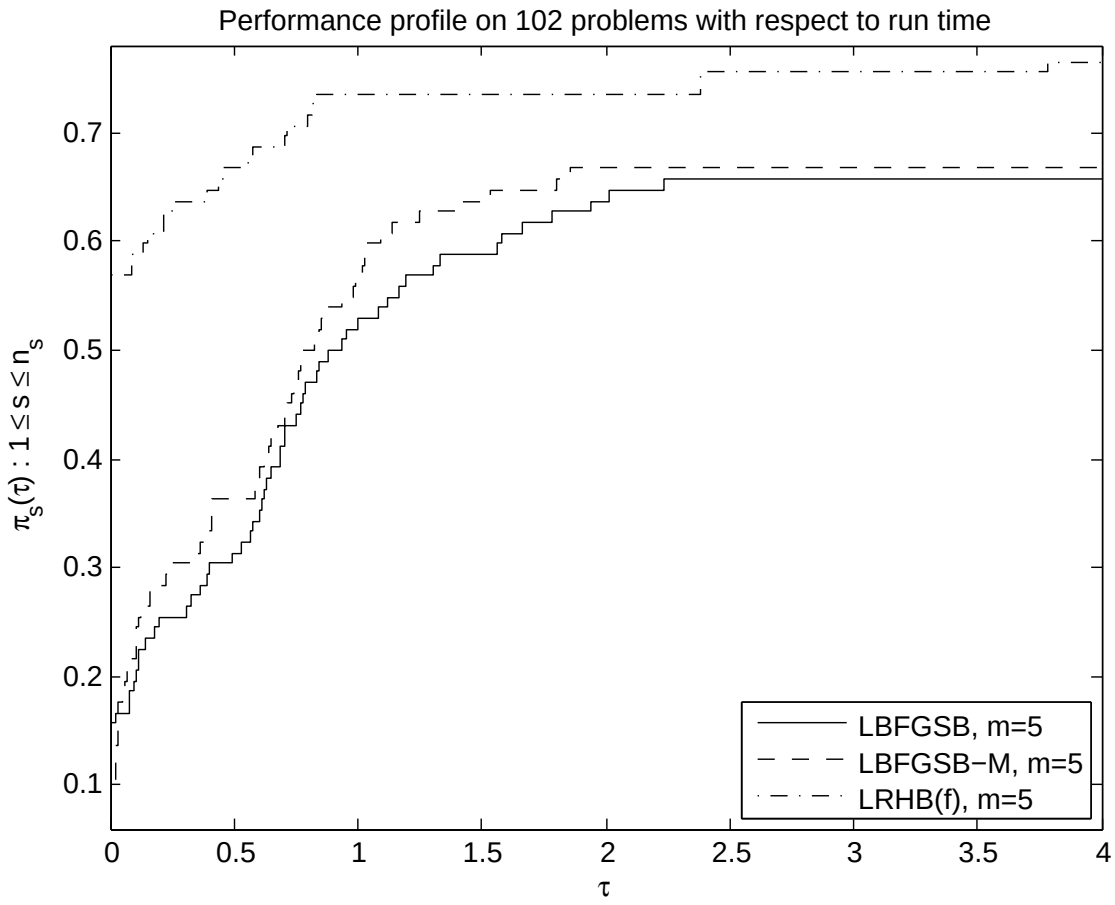

Figure 6.4: Performance profile for competitive solvers (time)

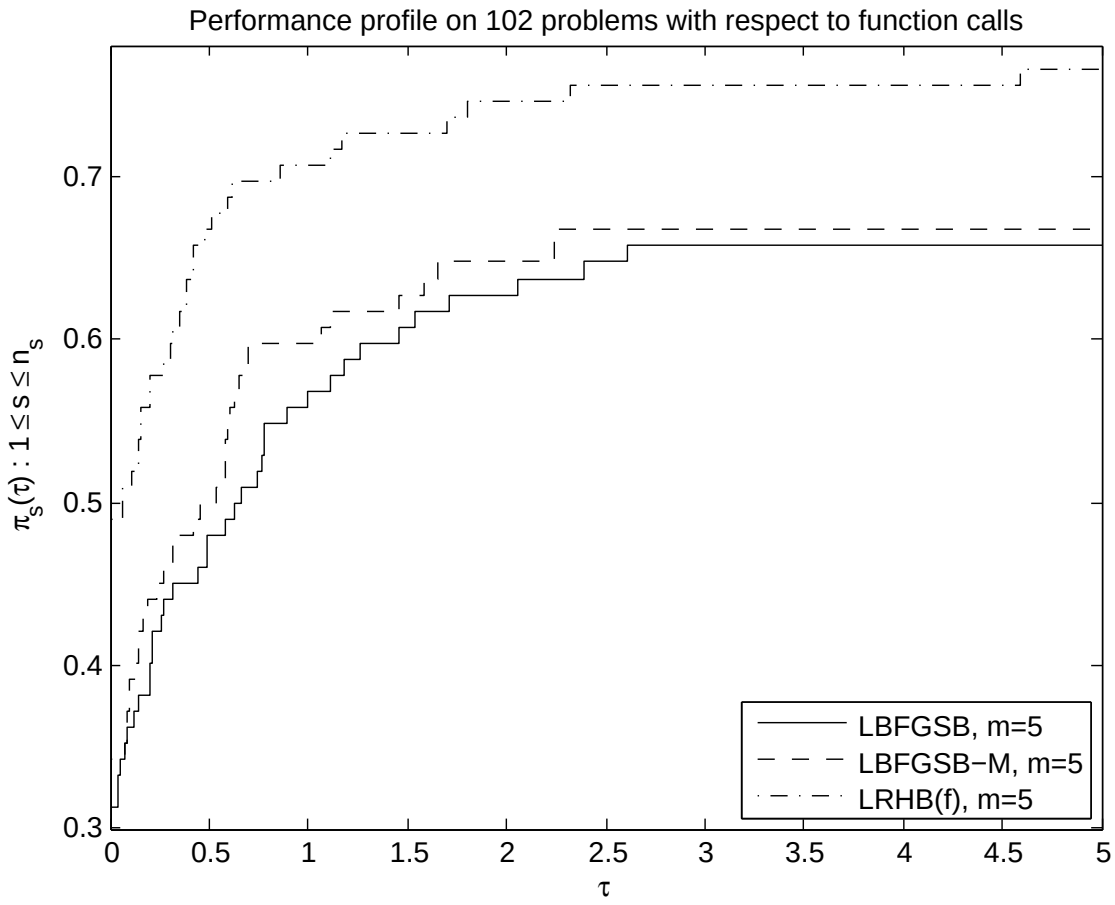

Figure 6.5: Performance profile for competitive solvers (nfg)

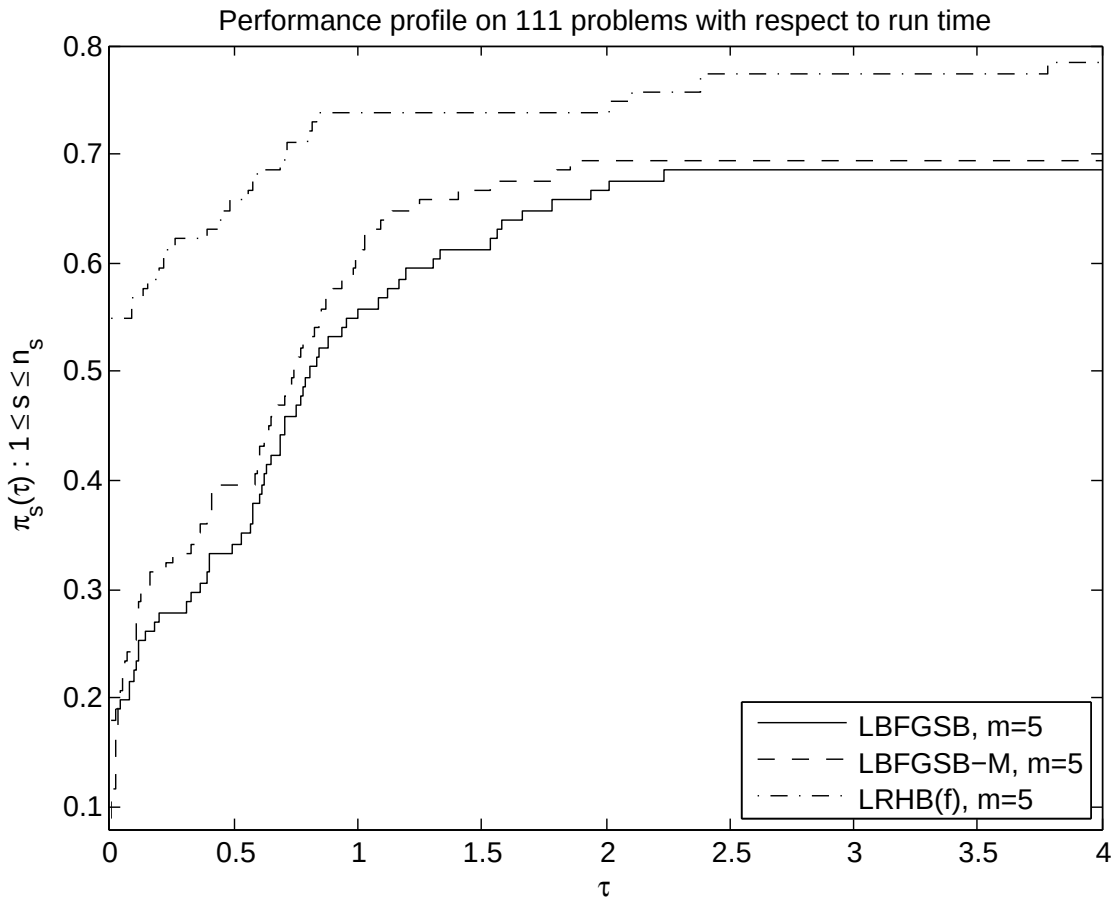

Figure 6.6: Performance profile for competitive solvers on full set (time)

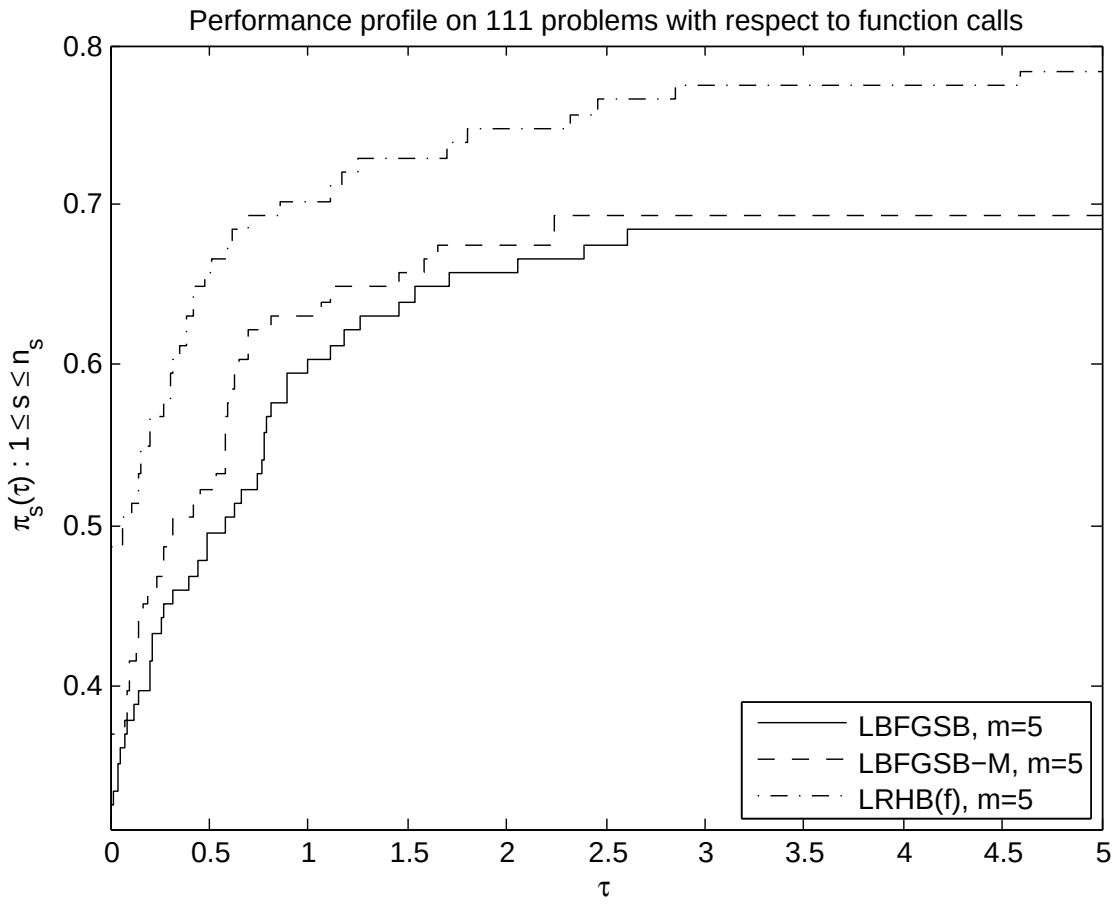

Figure 6.7: Performance profile for competitive solvers on full set (nfg)

Bibliography

[Ber76] D. P. Bertsekas. On the Goldstein-Levitin-Polyak gradient projection method. *IEEE Trans. Automatic Control*, AC-21(2):174–184, 1976.

[Ber82] D. P. Bertsekas. Projected Newton methods for optimization problems with simple constraints. *SIAM J. Control Optim.*, 20(2):221–246, 1982.

[BLNZ95] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16:1190–1208, 1995.

[Car07] P. Carbonetto. A MATLAB interface for L-BFGS-B. Dept of Computer Science, University of British Columbia, 2007.

[CGT00] A. R. Conn, N. I. M. Gould, and P. L. Toint. *Trust-Region Methods*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000.

[CM87] P. H. Calamai and J. J. Moré. Projected gradient methods for linearly constrained problems. *Math. Program.*, 39:93–116, 1987.

[Dav59] W. C. Davidon. Variable metric methods for minimization, a. e. c. research and development. Report ANL-5990, Argonne National Laboratory, Argonne, IL, 1959.

[DGKS76] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart. Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization. *Math. Comput.*, 30:772–795, 1976.

[DM02] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2, Ser. A):201–213, 2002.

[Fen81] M. C. Fenelon. *Preconditioned Conjugate-Gradient-Type Methods for Large-Scale Unconstrained Optimization*. PhD thesis, Department of Operations Research, Stanford University, Stanford, CA, 1981.

[GL01] P. E. Gill and M. W. Leonard. Reduced-Hessian quasi-Newton methods for unconstrained optimization. *SIAM J. Optim.*, 12(1):209–237, 2001.

[GL03] P. E. Gill and M. W. Leonard. Limited-memory reduced-Hessian methods for large-scale unconstrained optimization. *SIAM J. Optim.*, 14:380–401, 2003.

[GMS97] P. E. Gill, W. Murray, and M. A. Saunders. SNOPT: an SQP algorithm for large-scale constrained optimization. Numerical Analysis Report 97-2, Department of Mathematics, University of California, San Diego, La Jolla, CA, 1997.

[Gol64] A. A. Goldstein. Convex programming in Hilbert space. *Bulletin of the American Mathematical Society*, 70(5):709–710, 1964.

[GOT03] N. I. M. Gould, D. Orban, and P. L. Toint. CUTEr and SifDec: A constrained and unconstrained testing environment, revisited. *ACM Trans. Math. Software*, 29(4):373–394, 2003.

[Leo95] M. W. Leonard. *Reduced Hessian Quasi-Newton Methods for Optimization.* PhD thesis, Department of Mathematics, University of California, San Diego, 1995.

[LP66] E. S. Levitin and B. T. Polyak. Constrained minimization methods. *U.S.S.R. Comput. Math. and Math. Physics*, 6(5):1–50, 1966.

[McC69] G. P. McCormick. Anti-zig-zagging by bending. *Management Science*, 15(5):315–320, 1969.

[MS84] J. J. Moré and D. C. Sorensen. Newton's method. In G. H. Golub, editor, *Studies in Mathematics, Volume 24. MAA Studies in Numerical Analysis*, pages 29–82. Math. Assoc. America, Washington, DC, 1984.

[MT91] J. J. Moré and G. Toraldo. On the solution of large quadratic programming problems with bound constraints. *SIAM J. Optim.*, 1(1):93–113, 1991.

[MT94] J. J. Moré and D. J. Thuente. Line search algorithms with guaranteed sufficient decrease. *ACM Trans. Math. Software*, 20(3):286–307, 1994.

[NW99] J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer-Verlag, New York, 1999.

[NY97] Q. Ni and Y. Yuan. A subspace limited memory quasi-Newton algorithm for large-scale nonlinear bound constrained optimization. *Mathematics of Computation*, 66(220):1509–1520, 1997.

[Sie94] D. Siegel. Modifying the BFGS update by a new column scaling technique. *Mathematical Programming*, 66:45–78, 1994. Ser. A.

[SP78] D. F. Shanno and K. Phua. Matrix conditioning and nonlinear optimization. *Math. Program.*, 14:149–160, 1978.

[Wol69] P. Wolfe. Convergence conditions for ascent methods. *SIAM Review*, 11:226–235, 1969.

[Wol72] P. Wolfe. On the convergence of gradient methods under constraint. *IBM J. Res. Dev.*, 16:407–411, 1972.

[WY06] Z. Wang and Y. Yuan. A subspace implementation of quasi-newton trust region methods for unconstrained optimization. *Numerische Mathematik*, 104:241–269, 2006.

[ZBLN97] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Software*, 23(4):550–560, 1997.