A Review of Methods for Unconstrained Optimization: Theory, Implementation and Testing

Master's Thesis
Seppo Pulkkinen
University of Helsinki
Department of Mathematics and Statistics
November 2008

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Tiedekunta/Osasto – Fakultet/Sektion – Faculty Faculty of Science	Laitos – Institution – Department Department of Mathematics and Statistics
Tekijä – Författare – Author Seppo Pulkkinen	
Työn nimi – Arbetets titel – Title A Review of Methods for Unconstrained Optimization: Theory, Implementation and Testing	
Oppiaine – Läroämne – Subject Applied mathematics	

Työn laji – Arbetets art – Level Master's thesis	Aika – Datum – Month and year November 2008	Sivumäärä – Sidoantal – Number of pages 89 + 9 pages

Tiivistelmä – Referat – Abstract

This thesis discusses minimizing real-valued n-dimensional functions. The unconstrained minimization problem of a function $f:\mathbb{R}^n \rightarrow \mathbb{R}$ is formulated as finding a point **x*** such that

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

This thesis surveys the most commonly used methods for unconstrained minimization. The following methods are covered in this thesis:

- The Nelder and Mead simplex method
- The steepest descent method
- The Fletcher-Reeves and Polak-Ribière conjugate gradient methods
- The Newton method
- The quasi-Newton BFGS method

This thesis is divided into three parts: theory, implementation and testing. The theoretical background of each method with the relevant convergence results is disussed in detail in the first part. Detailed algorithm listings are also provided.

The second part of this thesis discusses implementation of these methods. The main contribution of this thesis is the implementation of GSL++, a C/C++ library for unconstrained minimization. This software library extends the GNU Scientific Library by providing additional algorithms and utilities. It also implements a GNU Octave interface for processing and plotting the results of minimization algorithms.

The third part of this thesis consists of testing the implemented algorithms. A detailed performance comparison of these algorithms is provided. Some of the interesting characteristic properties of these algorithms are also illustrated.

Avainsanat – Nyckelord – Keywords
unconstrained, optimization, minimization, numerical, GSL, Octave

Säilytyspaikka – Förvaringställe – Where deposited
Kumpula Science Library

Muita tietoja – Övriga uppgifter – Additional information

Acknowledgements

First and foremost, I wish to thank my supervisor prof. Matti Vuorinen for his advice, inspiration, and patience during this long project which has at times been with no clear direction. He was the one who introduced me this fascinating field of research, and without his inspiration, I would never have started writing my master's thesis on this topic. My second supervisor prof. Marko Mäkelä also made many helpful suggestions.

At the time of writing this thesis, I was working at the Finnish Meteorological Insititute. I wish to thank the people of FMI for sharing their expertise on the field of numerical computing. At FMI I also became familiar with practical applications of the algorithms studied in this thesis.

As any difficult project, this one also needed creative breaks in order to proceed. I thank all my friends and fellow students who have shared those much-needed moments with me. I also wish to express my deepest gratitude to my parents for all their support during this project.

Helsinki, November 2008

Seppo Pulkkinen

Contents

1 Introduction **1**

1.1 About this thesis 1

1.2 Problem definition and motivation 2

2 Mathematical background **3**

2.1 Preliminaries 3

2.1.1 Notations and basic definitions 3

2.1.2 Conditions for minima 5

2.1.3 Method definition 6

2.1.4 Convergence rates 6

2.1.5 Order of complexity 7

2.1.6 Invariance under transformations 8

2.2 Direct search methods 8

2.2.1 The Nelder and Mead simplex method 9

2.3 Gradient descent methods 13

2.3.1 Line search conditions and global convergence 14

2.3.2 The Moré and Thuente line search algorithm 16

2.3.3 The method of steepest descents 24

2.3.4 Conjugate gradient methods 25

2.3.5 The Newton method 30

2.3.6 Quasi-Newton methods 32

3 Implementation **37**

3.1 Introduction to GSL and BLAS 37

3.1.1 Minimization algorithms implemented in GSL 37

3.1.2 The GSL minimization interface 39

3.1.3 GSL linear algebra routines 41

3.1.4 Overview of BLAS 41

3.2 Overview of GSL++ 42

3.2.1 Proposed algorithms 43

3.2.2 GNU Octave utilities implemented in GSL++ 45

4 Numerical results **51**
4.1 Overview of testing procedures 51
4.2 Qualitative tests 52
4.2.1 The choice of starting point 52
4.3 Algorithm-specific tests 54
4.3.1 The Nelder and Mead simplex algorithm 54
4.3.2 The steepest descent algorithm 56
4.3.3 Conjugate gradient algorithms 57
4.3.4 Quasi-Newton BFGS algorithms 59
4.4 Comparison of algorithms 61
4.4.1 Sensitivity to line search parameters 61
4.4.2 Convergence rates 63
4.4.3 Function and gradient evaluation counts 69
4.4.4 Scale-dependency 71
4.4.5 Performance profiles 73
4.4.6 Asymptotic complexity 81

5 Conclusions and discussion **83**

References **85**

Appendices **90**

A Supplementary algorithms **91**
A.1 The backtracking algorithm 91
A.2 Fletcher's line search algorithm 92
A.3 Fletcher's initial step length selection 95
A.4 The modified $\mathbf{LDL}^T$ factorization 96
A.5 Finite-difference approximations 98

Chapter 1

Introduction

1.1 About this thesis

This thesis gives an in-depth review of the classical methods for unconstrained minimization of real-valued functions in arbitrary dimensions. The mathematical theory is discussed and a sample C/C++ framework for implementing and testing these methods is described. An extensive set of numerical test results is also provided. This thesis covers the Nelder and Mead simplex method, the steepest descent method, the Fletcher-Reeves and Polak-Ribière conjugate gradient methods, the Newton method and the quasi-Newton BFGS method. In addition, four different line search methods and a modification of the Newton method are covered.

Chapter 2 surveys the theoretical background of the methods reviewed in this thesis. An outline of the derivation of each method is given with algorithm listings. The characteristic properties of these methods and their implications to practical implementations are discussed. The proofs of these results are omitted due to the pragmatic approach taken in this thesis.

The main contribution of this thesis is the implementation of GSL++, a C/C++ library for unconstrained minimization in the GNU/Linux environment. This library extends the functionality of GSL, a general purpose numerical software library for scientific computing written in C. GSL++ implements several revised versions of the GSL algorithms with corrections suggested by the author of this thesis. In addition, it implements an interface for invoking these algorithms from GNU Octave, a MATLAB-compatible numerical software environment. An overview of these software packages is given in Chapter 3.

Due to the computational nature of solving minimization problems, testing of algorithms is an essential part of this thesis. Different approaches for evaluating performance of minimization algorithms are presented in Chapter 4, and a comprehensive performance comparison of the reviewed algorithms is given. Also the specific characteristics of each algorithm are analyzed experimentally with illustrations. Some of their theoretical results are also experimentally verified. The existing GSL implementations are compared to the GSL++ implementations in order to provide a set of reference results. Finally, Chapter 5 summarizes this thesis and points out some areas of future research.

The documented source code of the GSL++ library with installation instructions is available at `http://www.cs.helsinki.fi/u/sjipulkk/GSL++`.

1.2 Problem definition and motivation

This thesis discusses minimizing vector-valued functions $f : \mathbb{R}^n \to \mathbb{R}$ over $\mathbb{R}^n$, i.e. finding a point $\mathbf{x}^* \in \mathbb{R}^n$ such that

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

This is an *unconstrained minimization problem*, with *objective function* f and variables $\mathbf{x} = (x_1, \ldots, x_n)$. The need to solve such problems arises frequently in practical applications. Many physical systems involve potential functions whose minima describe specific states of the system. Problems of this type are also typical in statistical applications, where it is often necessary to fit functions to given data sets.

The primary goal of this thesis is to review a representative set of minimization methods, implement them and compare their performance. The emphasis is on implementing provably convergent methods with well-established theoretical foundations. Several of these methods are notoriously difficult to implement correctly, and thus their implementation details are emphasized. Some of them also exhibit behaviour that is not completely explained by their convergence theory. Thus, the numerical experiments carried out in this thesis are aimed for complementing these theoretical results.

As GSL implements only a basic set of minimization algorithms, the need to extend it with more advanced algorithms with stronger convergence theory motivated this thesis. It has yet a simple and powerful interface on which additional algorithms can be built. GSL neither implements any tools for analyzing and plotting the results of its algorithms, which motivated the development of a GNU Octave-based interface for it. Such an interface greatly facilitates analyzation and visualization of the results of minimization algorithms.

The choice of open-source software libraries in the GNU/Linux environment was motivated by the rapid development on this front during recent years. In particular, GNU Octave has become an increasingly capable non-commercial replacement of MATLAB. All software libraries and applications utilized in this thesis fall under the GNU general public license (GPL).

Chapter 2

Mathematical background

2.1 Preliminaries

2.1.1 Notations and basic definitions

Throughout this thesis we use boldface letters to denote vectors and matrices. A vector in $\mathbb{R}^n$ is always assumed to be a column vector, i.e. an $n \times 1$ matrix. Rows and columns of matrices are denoted by subscripts and superscripts, respectively. If not stated otherwise, the vector norm $\|\cdot\|$ denotes the l_2-norm

$$\|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

The *inner product* $\cdot : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ of two vectors $\mathbf{u}$ and $\mathbf{v}$ is defined as

$$\mathbf{u} \cdot \mathbf{v} \equiv \mathbf{u}^T \mathbf{v} = \sum_{i=1}^{n} u_i v_i.$$

The elements of the *outer product* $\otimes : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^{n \times n}$ of two vectors $\mathbf{u}$ and $\mathbf{v}$ are defined as

$$[\mathbf{u} \otimes \mathbf{v}]_{ij} \equiv [\mathbf{u}\mathbf{v}^T]_{ij} = u_i v_j.$$

We use the notation $f \in C^i(\mathbb{R}^n, \mathbb{R})$ for a function $f : \mathbb{R}^n \to \mathbb{R}$ with continuous ith order partial derivatives with respect to all of its components. The gradient of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \frac{\partial f}{\partial x_2}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\mathbf{x}) \end{bmatrix}.$$

The Taylor series approximation of a C^2-function is one of the cornerstones of minimization methods. The second-order Taylor series approximation of $f(\mathbf{x}+\mathbf{h})$, where $f : \mathbb{R}^n \to \mathbb{R}$ is a C^2-function, is given by

$$f(\mathbf{x}+\mathbf{h}) \approx f(\mathbf{x}) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(\mathbf{x}) h_i + \frac{1}{2} \sum_{i,j=1}^{n} \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}) h_i h_j. \tag{2.1.1}$$

The third term in the above Taylor series represents the *Hessian* matrix. Its elements are given by

$$[\mathbf{H}_f(\mathbf{x})]_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}), \quad i, j = 1, ..., n.$$

Note that since f is a C^2-function, $\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}$, and thus the Hessian matrix is symmetric. By using the above definition of Hessian matrix, expression (2.1.1) can be given equivalently in matrix form, that is

$$f(\mathbf{x}+\mathbf{h}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^T \mathbf{h} + \frac{1}{2} \mathbf{h}^T \mathbf{H}_f(\mathbf{x}) \mathbf{h}. \tag{2.1.2}$$

Positive definiteness of a matrix is a very important property in the context of function minimization. We say that a matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$ is positive definite if

$$\mathbf{x}^T \mathbf{M} \mathbf{x} > 0$$

for all nonzero vectors $\mathbf{x} \in \mathbb{R}^n$. If only weak inequality holds, we say that $\mathbf{M}$ is *positive semidefinite*. Note that a positive definite matrix is invertible. In addition, a matrix is positive definite if and only if its inverse is positive definite [HJ85, Chap. 7].

The theoretical foundations of most minimization methods are built on *quadratic functions*. The form of a quadratic function $f : \mathbb{R}^n \to \mathbb{R}$ we use is

$$f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{b}^T \mathbf{x} + c, \tag{2.1.3}$$

where $\mathbf{A} \in \mathbb{R}^{n \times n}$ is symmetric, $\mathbf{b} \in \mathbb{R}^n$ and $c \in \mathbb{R}$. By applying the rule of differentiating a product to (2.1.3), we obtain

$$\nabla f(\mathbf{x}) = \frac{1}{2}(\mathbf{A} + \mathbf{A}^T)\mathbf{x} - \mathbf{b} = \mathbf{A}\mathbf{x} - \mathbf{b}. \tag{2.1.4}$$

Several convergence results we will state in this thesis require that the objective function is *Lipschitz-continuous* in a given set.

Definition 2.1.5. Let $f : \mathbb{R}^n \to \mathbb{R}$, $f \in C^1(\mathcal{D}, \mathbb{R})$, where $\mathcal{D} \subset \mathbb{R}^n$ is an open set. We say that f is locally Lipschitz-continuous in $\mathcal{D}$, if there exists $L \geq 0$ such that

$$\|f(\mathbf{x}) - f(\mathbf{y})\| \leq L \|\mathbf{x} - \mathbf{y}\|$$

for all $\mathbf{x}, \mathbf{y} \in \mathcal{D}$.

It follows from the *mean-value theorem* that every C^1-function in a compact set X is also locally Lipschitz-continuous in X with $L = \sup_{\mathbf{x} \in X} \|\nabla f(\mathbf{x})\|$. Therefore the requirement of Lipschitz-continuity in the convergence results given in the subsequent chapters is usually implied by additional assumptions of C^1- and C^2-continuity.

Convexity is another key property in the theory of minimization methods. Convex sets and functions are defined as follows.

Definition 2.1.6. [Ber99, Definition B.1] A set $X \subset R^n$ is convex, if for any $\mathbf{x}_1, \mathbf{x}_2 \in X$ and for any $t \in [0, 1]$, we have

$$t\mathbf{x}_1 + (1-t)\mathbf{x}_2 \in X.$$

Definition 2.1.7. [Ber99, Proposition B.4] A function $f : \mathbb{R}^n \to \mathbb{R}$, $f \in C^2(X, \mathbb{R})$, is convex in a set $X \subset \mathbb{R}^n$ if X is convex and $\mathbf{H}_f(\mathbf{x})$ is positive semidefinite for all $\mathbf{x} \in X$.

We say that a function is *strictly convex*, if positive semidefiniteness is replaced with positive definiteness in the above definition.

2.1.2 Conditions for minima

A minimum of a function can be either *local* or *global*. Their standard definitions given in the literature are stated below.

Definition 2.1.8. A point $\mathbf{x}^* \in \mathbb{R}^n$ is a *local minimizer* of $f : \mathbb{R}^n \to \mathbb{R}$, if there exists a neighbourhood

$$U = \{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x} - \mathbf{x}^*\| < \epsilon\}$$

such that f attains its minimum value in U at $\mathbf{x}^*$, i.e.

$$f(\mathbf{x}^*) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in U.$$

In general, a function may have several distinct local minima. In a *global minimum*, a function attains a value smaller than any of its other minima. This is formally stated in the following definition.

Definition 2.1.9. A point $\mathbf{x}^* \in \mathbb{R}^n$ is a global minimizer of $f : \mathbb{R}^n \to \mathbb{R}$, if f attains there its smallest value in $\mathbb{R}^n$, i.e.

$$f(\mathbf{x}^*) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathbb{R}^n.$$

In the absence of equalities in definitions 2.1.8 and 2.1.9, and with the assumption that $\mathbf{x} \neq \mathbf{x}^*$, we say that a minimizer is *strict*.

The *sufficient* second-order conditions for a given point $\mathbf{x}^* \in \mathbb{R}^n$ to be a strict local minimizer of a C^2-function $f : \mathbb{R}^n \to \mathbb{R}$ are that

$$\nabla f(\mathbf{x}^*) = \mathbf{0}, \quad \mathbf{x}^T \mathbf{H}_f(\mathbf{x}^*)\mathbf{x} > 0 \quad \forall \mathbf{x} \neq \mathbf{0}. \tag{2.1.10}$$

In the absence of the second condition, we say that $\mathbf{x}^*$ is a *stationary point*, which is not guaranteed to be an extremal point. Also, if positive definiteness of the Hessian is replaced by positive semidefiniteness, these conditions are not sufficient but *necessary* conditions for a local minimizer.

Strict convexity of the objective function in a given convex set implies that a local minimizer is also its unique global minimizer in this set [Ber99, Propositions B.4. and B.10.]. For this reason, convexity is often a tacitly made assumption in theory of function minimization.

2.1.3 Method definition

This thesis discusses iterative methods for finding local minima. These methods produce a sequence of sets of iterates (X_k), $X_k = \{\mathbf{x}_k^i\}_{i=1}^m \subset \mathbb{R}^n$, starting from a given set of starting points $\{\mathbf{x}_0^i\}_{i=1}^m \subset \mathbb{R}^n$, where m denotes the number of iterates generated at each step. The most general form of such an iteration step is given by

$$\mathbf{x}_{k+1}^i = \mathbf{x}_k^i + \mathbf{s}_k^i(X_k, F_k, G_k, H_k), \tag{2.1.11}$$

where

$$X_k = \{\mathbf{x}_k^i\}, \quad F_k = \{f(\mathbf{x}_k^i)\}, \quad G_k = \{\nabla f(\mathbf{x}_k^i)\}, \quad H_k = \{\mathbf{H}_f(\mathbf{x}_k^i)\}$$

and $i = 1, \ldots, m$. The steps $\mathbf{s}_k^i$ always depend at least on the current iterates X_k and their associated function values F_k but not necessarily on function derivatives. Also note that they may depend on the previous iteration history, which is not stated in this definition for notational simplicity. As methods of this type are formulated for finding local minima, it should also be emphasized that the result may depend to a great extent on the choice of starting points.

2.1.4 Convergence rates

One of the key measures for comparing different minimization methods is their theoretical rate of convergence. Our emphasis is on the *asymptotic* convergence rate in the neighbourhood of a solution. A commonly used approach is to compare the improvement of each estimate $\mathbf{x}_{k+1}$ over the previous estimate $\mathbf{x}_k$ by measuring their distances to the assumed solution. The following standard definitions of convergence rates are given in the literature [NW99, p. 28-29]. [1]

[1] We assume that $\mathbf{x}_k \neq \mathbf{x}^*$ for all $k \in \mathbb{N}$.

Definition 2.1.12. Let $(\mathbf{x}_k)$, where $\mathbf{x}_k \in \mathbb{R}^n$ for all k, be a sequence that converges to an $\mathbf{x}^* \in \mathbb{R}^n$. The convergence rate of the sequence $(\mathbf{x}_k)$ is

1. *linear*, if there exists $r \in]0,1[$ and $k_0 \in \mathbb{N}$ such that
$$\tfrac{\|\mathbf{x}_{k+1}-\mathbf{x}^*\|}{\|\mathbf{x}_k-\mathbf{x}^*\|} \leq r \quad \text{for all } k > k_0. \tag{2.1.13}$$

2. *superlinear*, if
$$\lim_{k\to\infty} \frac{\|\mathbf{x}_{k+1}-\mathbf{x}^*\|}{\|\mathbf{x}_k-\mathbf{x}^*\|} = 0. \tag{2.1.14}$$

3. *quadratic*, if there exists $M > 0$ and $k_0 \in \mathbb{N}$ such that
$$\tfrac{\|\mathbf{x}_{k+1}-\mathbf{x}^*\|}{\|\mathbf{x}_k-\mathbf{x}^*\|^2} \leq M \quad \text{for all } k > k_0. \tag{2.1.15}$$

2.1.5 Order of complexity

For theoretical considerations, it is useful to state the asymptotic upper bound of a function in terms of another function. In this manner, simpler functions such as polynomials can be used to divide functions into classes according to their asymptotic behaviour. The following definition is used throughout this thesis.

Definition 2.1.16. [Ueb97, p. 185-186] A function $f : \mathbb{N} \to \mathbb{N}$ is of order $g : \mathbb{N} \to \mathbb{N}$, if there exists constants $c \in \mathbb{R}$ and $n_0 \in \mathbb{N}$ such that
$$f(n) \leq cg(n) \quad \text{for all} \quad n \geq n_0 \tag{2.1.17}$$

This is usually expressed by using the *Landau-O notation.* We say that [2]
$$f = \mathcal{O}(g(n)),$$

if f is of order g. The function classes of particular importance are $\mathcal{O}(1)$, $\mathcal{O}(n)$, $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, which are referred to as *constant*, *linear*, *quadratic* and *cubic*, respectively. In this thesis, the above definition refers to the order of complexity, defined as a function of n, where n is the problem dimension. By the notion of complexity we refer to as computational and space complexity of an algorithm. The former gives a measure of used floating-point operations, and the latter measures storage requirements.

[2]This is a notational convention and not an equation in the strict sense. A more correct, but less frequently used notation is $f \in \mathcal{O}(g(n))$.

2.1.6 Invariance under transformations

The variables of a minimization problem are often scaled in such a way that they have different ranges of magnitude. This can have a substantial effect on the rate of convergence. Thus, a highly desired property of a minimization method is that, excluding errors due to limited numerical precision, its iterates remain invariant under transformations of variables.

We denote the original variables by $\mathbf{x}$ and the transformed variables by $\mathbf{y}$. Consider a linear transformation $T : \mathbb{R}^n \to \mathbb{R}^n$,

$$\mathbf{y} = T(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b} \tag{2.1.18}$$

with a nonsingular matrix $\mathbf{A}$. Its inverse transformation T^{-1} is given by

$$\mathbf{x} = T^{-1}(\mathbf{y}) = \mathbf{A}^{-1}(\mathbf{y} - \mathbf{b}). \tag{2.1.19}$$

We denote the transformed objective function by $\tilde{f}$,

$$\tilde{f}(\mathbf{y}) = f(T^{-1}(\mathbf{y})). \tag{2.1.20}$$

Lemma 2.1.21. [Fle80, p. 46] *The gradient and Hessian of a function $f : \mathbb{R}^n \to \mathbb{R}$, $f \in C^2(\mathbb{R}^n, \mathbb{R})$, in the coordinates transformed according to (2.1.18) are given by*

$$\nabla f(\mathbf{x}) = \mathbf{A}^T \nabla \tilde{f}(\mathbf{y}), \tag{2.1.22}$$

$$\mathbf{H}_f(\mathbf{x})^{-1} = \mathbf{A}^{-1}\tilde{\mathbf{H}}_f(\mathbf{y})^{-1}\mathbf{A}^{-T}. \tag{2.1.23}$$

Definition 2.1.24. A minimization method is *invariant* under transformations of the form (2.1.18), if iteration formula (2.1.11) is equivalent to

$$\mathbf{y}_{k+1}^i = \mathbf{y}_k^i + \mathbf{s}_k^i(Y_k, \tilde{F}_k, \tilde{G}_k, \tilde{H}_k), \tag{2.1.25}$$

where

$$Y_k = \{\mathbf{y}_k^i\}, \quad \tilde{F}_k = \{\tilde{f}(\mathbf{y}_k^i)\}, \quad \tilde{G}_k = \{\nabla \tilde{f}(\mathbf{y}_k^i)\}, \quad \tilde{H}_k = \{\tilde{\mathbf{H}}_f(\mathbf{y}_k^i)\}$$

via a transformation of the form (2.1.18). That is, for all $k \in \mathbb{N}$, $i = 1, \ldots, m$,

$$\mathbf{y}_{k+1}^i = T(\mathbf{x}_{k+1}^i) \quad \text{if} \quad \mathbf{y}_k^i = T(\mathbf{x}_k^i).$$

2.2 Direct search methods

The *direct search methods* do not require differentiability or even continuity of the objective function. Thus, they are applicable to broadest range of problems. These methods typically exhibit slower convergence rates than gradient-descent methods. On the other hand, they tend to be more reliable on noisy functions or functions with multiple local minima. The only representative of this class discussed in this thesis is the Nelder and Mead simplex method.

2.2.1 The Nelder and Mead simplex method

Despite its age, The Nelder and Mead simplex method [NM65] is still used in modern numerical software packages such as MATLAB and GSL. The original paper by Nelder and Mead does not state the algorithm in a very strict sense, which has led to numerous different interpretations. The variation of the algorithm reviewed in this thesis is based on a more formally stated version by Lagarias et. al. [LRWW98].

A simplex in $\mathbb{R}^n$ is a $n+1$-element set of n-dimensional *vertices* with a nondegenerate set of *edges*.

Definition 2.2.1. A simplex $S \subset \mathbb{R}^n$ is a set of $n+1$ vertices, $\{\mathbf{x}_i\}_{i=1}^{n+1} \subset \mathbb{R}^n$, connected by edges

$$\mathbf{E} = (\mathbf{x}_2 - \mathbf{x}_1, \mathbf{x}_3 - \mathbf{x}_1, \ldots, \mathbf{x}_{n+1} - \mathbf{x}_1) \tag{2.2.2}$$

that form a basis of $\mathbb{R}^n$, i.e. $\operatorname{span}(\mathbf{E}) = \mathbb{R}^n$.

The Nelder and Mead simplex algorithm applied to a function $f : \mathbb{R}^n \to \mathbb{R}$ maintains an ordered set of simplex vertices $(\mathbf{x}_1^k, \ldots, \mathbf{x}_{n+1}^k)$, where k denotes the kth iteration step. The ordering of simplex vertices is [3]

$$f(\mathbf{x}_1) \leq f(\mathbf{x}_2) \leq \cdots \leq f(\mathbf{x}_{n+1}). \tag{2.2.3}$$

We refer the vertex $\mathbf{x}_1$ with the lowest function value to as the *lowest* vertex. Likewise, we refer the vertex with the highest function value to as the *highest* vertex.

At the beginning of each iteration, the *centroid* point

$$\bar{\mathbf{x}} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i \tag{2.2.4}$$

is computed for all subsequent simplex operations. Each step following the computation of the centroid point computes a *trial point*. Each iteration has two possible outcomes: 1. one of the computed trial points with a lower function value than the highest vertex is accepted and the highest vertex is replaced with it, or 2. a *shrink* operation is performed.

The Nelder and Mead simplex algorithm uses four different trial points. These trial points with their associated scalar parameters are given in the following definition. They are also illustrated in Figure 1.

Definition 2.2.5. Trial points of the Nelder and Mead simplex algorithm

$$\text{Reflection}(\rho > 0): \quad \mathbf{x}_r = \bar{\mathbf{x}} + \rho(\bar{\mathbf{x}} - \mathbf{x}_{n+1}) \tag{2.2.6}$$

$$\text{Expansion}(\chi > \max\{1, \rho\}): \quad \mathbf{x}_e = \bar{\mathbf{x}} + \chi(\mathbf{x}_r - \bar{\mathbf{x}}) \tag{2.2.7}$$

$$\text{Outside contraction}(0 < \gamma < 1): \quad \mathbf{x}_{oc} = \bar{\mathbf{x}} + \gamma(\mathbf{x}_r - \bar{\mathbf{x}}) \tag{2.2.8}$$

$$\text{Inside contraction}(0 < \gamma < 1): \quad \mathbf{x}_{ic} = \bar{\mathbf{x}} - \gamma(\bar{\mathbf{x}} - \mathbf{x}_{n+1}) \tag{2.2.9}$$

[3] We omit the superscript k for notational convenience.

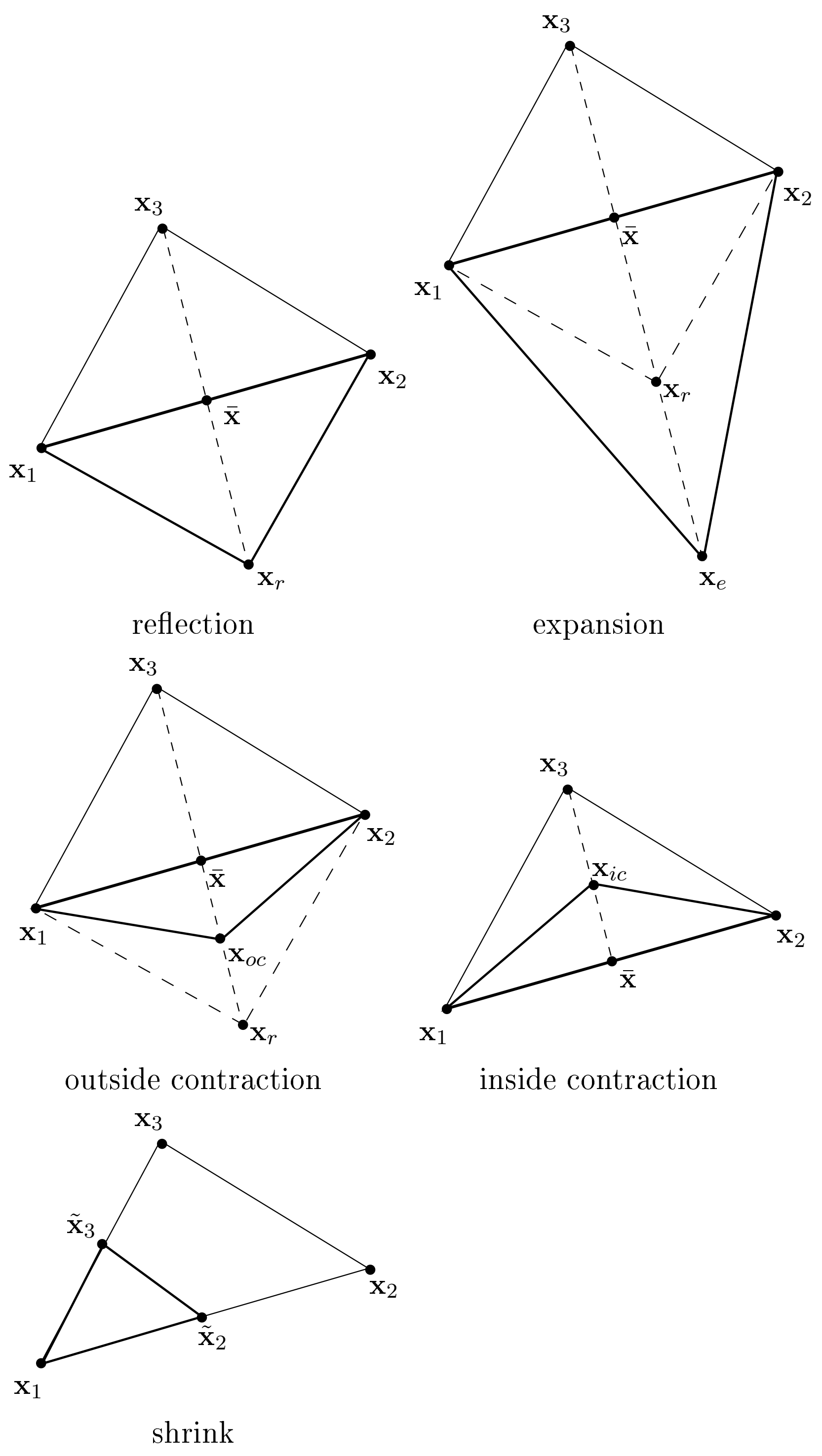


Figure 1: The operations performed on a simplex in $\mathbb{R}^2$.

If none of the trial points yield improvement, a *shrink* operation is performed.

Definition 2.2.10. A shrink operation transforms the simplex vertices $\{\mathbf{x}_i\}_{i=2}^{n+1}$ according to

$$\tilde{\mathbf{x}}_i = \mathbf{x}_1 + \sigma(\mathbf{x}_i - \mathbf{x}_1), \quad 0 < \sigma < 1. \tag{2.2.11}$$

A formal statement of the Nelder and Mead simplex algorithm with the heuristic rules for modifying the simplex is given in Algorithm 1. The rationale of this algorithm is that successive decreases in function values at the simplex vertices eventually lead to convergence to a minimizer. However, its convergence has been established only in low dimensions [LRWW98], and there exists counterexamples showing where it instead converges to a non-stationary point [McK98].

Stopping criteria

Since the simplex method is based on comparison of function values, it makes sense to use the stopping criterion

$$|f(\mathbf{x}_{n+1}) - f(\mathbf{x}_1)| < \epsilon,$$

which also was the original stopping criterion suggested by Nelder and Mead [NM65]. Another stopping criterion is to test $V(S_k) < \epsilon$, where $V(S_k)$ is the simplex volume [LRWW98, p. 6]

$$V(S_k) = \frac{|det(\mathbf{E}_k)|}{n!}. \tag{2.2.12}$$

The choice of initial simplex

A practical choice of the initial simplex vertices is [4]

$$\mathbf{x}_{i+1}^0 = \mathbf{x}_1^0 + \lambda_i \mathbf{e}_i, \quad i = 1, \ldots, n, \tag{2.2.13}$$

where $\mathbf{x}_1^0$ denotes the given starting point $\mathbf{x}_0$ and $\mathbf{e}_i$ denotes a unit vector along the ith coordinate axis. The parameters λ_i are supplied by user. This choice simplifies computation of the initial simplex volume, since the edge matrix $\mathbf{E}_0$ in formula (2.2.12) reduces to a diagonal matrix. Consequently, formula (2.2.12) then reduces to

$$V(S_0) = \frac{1}{n!} \prod_{i=2}^{n+1} \|\mathbf{x}_i - \mathbf{x}_1\|$$

which requires only $\mathcal{O}(n)$ operations.

[4]The vertices of the initial simplex are not assumed to be ordered.

Algorithm 1: The Nelder and Mead simplex algorithm, one iteration.

```
Order simplex vertices according to (2.2.3).                 /* 1. Order */
Compute $\bar{\mathbf{x}}$ from (2.2.4).
Compute $\mathbf{x}_r$ from (2.2.6).                         /* 2. Reflect */
if $f(\mathbf{x}_1) \leq f(\mathbf{x}_r) < f(\mathbf{x}_n)$ then
    $\mathbf{x}_{n+1} \leftarrow \mathbf{x}_r$
    Terminate this iteration step.
if $f(\mathbf{x}_r) < f(\mathbf{x}_1)$ then                   /* 3. Expand */
    Compute $\mathbf{x}_e$ from (2.2.7).
    if $f(\mathbf{x}_e) < f(\mathbf{x}_r)$ then
        $\mathbf{x}_{n+1} \leftarrow \mathbf{x}_e$
    else
        $\mathbf{x}_{n+1} \leftarrow \mathbf{x}_r$
    Terminate this iteration step.
/* 4. Contract                                               */
if $f(\mathbf{x}_r) \geq f(\mathbf{x}_n)$ then
    if $f(\mathbf{x}_n) \leq f(\mathbf{x}_r) < f(\mathbf{x}_{n+1})$ then /* 4a. Contract outside */
        Compute $\mathbf{x}_{oc}$ from (2.2.8).
        if $f(\mathbf{x}_{oc}) \leq f(\mathbf{x}_r)$ then
            $\mathbf{x}_{n+1} \leftarrow \mathbf{x}_{oc}$
            Terminate this iteration step.
        else
            Go to step 5.
    if $f(\mathbf{x}_r) \geq f(\mathbf{x}_{n+1})$ then          /* 4b. Contract inside */
        Compute $\mathbf{x}_{ic}$ from (2.2.9).
        if $f(\mathbf{x}_{ic}) < f(\mathbf{x}_{n+1})$ then
            $\mathbf{x}_{n+1} \leftarrow \mathbf{x}_{ic}$
            Terminate this iteration step.
        else
            Go to step 5.
Apply (2.2.11) to simplex vertices.                          /* 5. Shrink */
```

Properties of the Nelder and Mead algorithm

Lagarias et. al. proved formally several properties of the Nelder and Mead algorithm. Firstly, they state that the Nelder and Mead simplex algorithm cannot produce a degenerate simplex, provided that the initial simplex is nondegenerate [LRWW98, Lemma 3.1].

The following result relaxes the requirement of expensive recomputation of the simplex volume at each iteration, provided that the initial simplex volume is known.

Theorem 2.2.14. [LRWW98, Lemma 3.1.] *Let S be a simplex defined by 2.2.1. Suppose that Algorithm 1 is applied to S. The volume of the modified simplex S' is given by*

$$\begin{cases} V(S') = |\tau| V(S), & \textit{following a nonshrink step} \\ V(S') = \sigma^n V(S), & \textit{following a shrink step,} \end{cases}$$

where τ is given by

$$\begin{array}{llll} \tau = \rho, & \textit{reflection} & \tau = \rho\chi, & \textit{expansion} \\ \tau = \rho\gamma, & \textit{outside contraction} & \tau = -\gamma, & \textit{inside contraction.} \end{array}$$

The Nelder and Mead simplex method is also invariant under linear transformations of the form (2.1.18). Lagarias et. al. essentially proved the following result. Provided that the initial simplex S_0 is also transformed according to (2.1.18), this result holds for the entire sequence of generated simplices.

Theorem 2.2.15. [LRWW98, Lemma 3.2.] *Suppose that the simplices $S_{k+1} = \{\mathbf{x}_i^{k+1}\}_{i=1}^{n+1}$ and $\tilde{S}_{k+1} = \{\mathbf{y}_i^{k+1}\}_{i=1}^{n+1}$ are obtained by applying Algorithm 1 to simplices $S_k = \{\mathbf{x}_i^k\}_{i=1}^{n+1}$ and $\tilde{S}_k = \{\mathbf{y}_i^k\}_{i=1}^{n+1}$, respectively. If*

$$\mathbf{y}_i^k = \mathbf{A}\mathbf{x}_i^k + \mathbf{b} \quad \forall i = 1, \ldots, n+1,$$

then

$$\mathbf{y}_i^{k+1} = \mathbf{A}\mathbf{x}_i^{k+1} + \mathbf{b} \quad \forall i = 1, \ldots, n+1.$$

2.3 Gradient descent methods

For the rest of the theoretical part of this thesis we discuss minimization methods that use function derivatives and require the objective function to strictly decrease each iteration, i.e. $f(\mathbf{x}_{k+1}) < f(\mathbf{x}_k)$ for all k. The general iteration formula for this class of methods is given by

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k, \tag{2.3.1}$$

where $\mathbf{d}_k$ denotes the given *search direction* and α_k denotes the *step length.*

The step lengths α_k are obtained by one-dimensional *line minimization*. Minimizing the objective function along the search direction is formulated as finding a step length α_k such that

$$\alpha_k = \arg\min_{\alpha>0} f(\mathbf{x}_k + \alpha \mathbf{d}_k), \tag{2.3.2}$$

where $\mathbf{d}_k$ is a *descent direction*, i.e.

$$\frac{d}{d\alpha_k} f(\mathbf{x}_k + \alpha_k \mathbf{d}_k)|_{\alpha_k=0} < 0. \tag{2.3.3}$$

The next iterate $\mathbf{x}_{k+1}$ is assumed to be a local minimizer along the search direction when [5]

$$\begin{aligned} \tfrac{d}{d\alpha_k} f(\mathbf{x}_k + \alpha_k \mathbf{d}_k) &= \nabla f(\mathbf{x}_k + \alpha_k \mathbf{d}_k)^T \mathbf{d}_k \\ &= \nabla f(\mathbf{x}_{k+1})^T \mathbf{d}_k = 0. \end{aligned} \tag{2.3.4}$$

The following stopping criteria for gradient descent methods have been suggested in the literature:

$$\|\nabla f(\mathbf{x}_k)\| < \epsilon, \quad \|\mathbf{x}_{k+1} - \mathbf{x}_k\| < \epsilon, \quad f(\mathbf{x}_k) - f(\mathbf{x}_{k+1}) < \epsilon.$$

Testing the gradient norm follows from conditions (2.1.10), and it is the most commonly used stopping criterion with these methods.

2.3.1 Line search conditions and global convergence

From the practical point of view, an exact line minimization is too expensive, and a tradeoff between performance and accuracy is usually done by employing some sort of *inexact* line search. Based on the *Wolfe conditions* [NW99, p. 39], a rigorous mathematical framework with strong convergence results has been developed. In what follows, we will simplify the notation by defining the objective function along the current search direction as

$$\phi(\alpha) \equiv f(\mathbf{x}_k + \alpha \mathbf{d}_k). \tag{2.3.5}$$

We also assume that

$$\begin{cases} f \in C^1(\mathcal{L}, \mathbb{R}) \\ f \text{ is bounded from below in } \mathcal{L}, \end{cases} \tag{2.3.6}$$

where $\mathcal{L} = \{\mathbf{x}_k + \alpha \mathbf{d}_k \mid \alpha \geq 0\}$ is the search half-line extending from $\mathbf{x}_k$.

The *strong Wolfe conditions* for the step lengths α_k are stated as

$$\phi(\alpha_k) \leq \phi(0) + \mu \alpha_k \phi^{'}(0) \tag{2.3.7}$$

$$|\phi'(\alpha_k)| \leq \eta |\phi'(0)|, \tag{2.3.8}$$

[5] Line search procedures do not typically enforce second-order conditions, and in general, they only guarantee convergence of the multidimensional algorithm to a stationary point.

where $0 < \mu < \frac{1}{2}$ and $\mu < \eta < 1$. The first condition guarantees *sufficient decrease* of the objective function, and the second condition restricts the step length to the neighbourhood of a minimizer. Assuming (2.3.6), it can be shown that there always exists a step length interval $I =]\alpha_l, \alpha_u[$ such that any $\alpha \in I$ satisfies conditions (2.3.7) and (2.3.8) [DS83, Theorem 6.3.2]. A step length interval satisfying these conditions is illustrated in Figure 2.

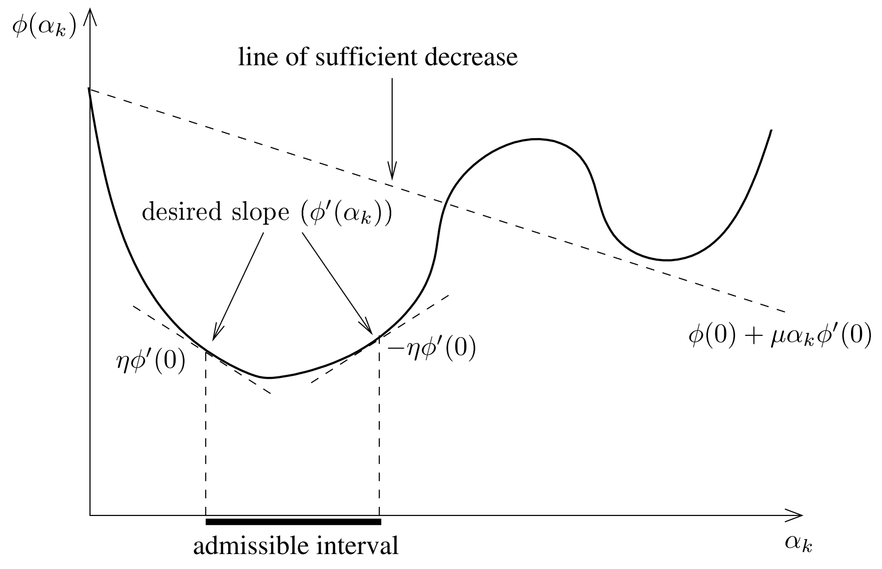

$\phi(\alpha_k)$
line of sufficient decrease
desired slope ($\phi'(\alpha_k)$)
$\eta\phi'(0)$
$-\eta\phi'(0)$
$\phi(0) + \mu\alpha_k\phi'(0)$
admissible interval
α_k

Figure 2: A step length interval satisfying the strong Wolfe conditions.

In some cases, condition (2.3.8) is replaced with a weaker condition

$$\phi'(\alpha_k) \geq \eta\phi'(0). \tag{2.3.9}$$

A line search satisfying condition (2.3.8) approaches an exact line search as $\eta \to 0$. The weaker condition does not obey this limit, since ϕ' is not restricted from the right-hand side. Also note that condition (2.3.8) implies condition (2.3.9). One can also show that the Wolfe conditions are scale-invariant [NW99].

The cosine of the angle between the search directions $\mathbf{d}_k$ and the negative gradient directions $-\nabla f(\mathbf{x}_k)$, that is

$$\cos\theta_k = \frac{-\nabla f(\mathbf{x}_k)^T \mathbf{d}_k}{\|\nabla f(\mathbf{x}_k)\| \|\mathbf{d}_k\|} \tag{2.3.10}$$

is a fundamental concept in the convergence theory of gradient descent methods. A sufficient condition for convergence of a subsequence of $(\mathbf{x}_k)$ to a stationary point, and thus the existence of an iterate that satisfies the stopping criterion $\|\nabla f(\mathbf{x}_k)\| < \epsilon$, is that a subsequence of $(\cos\theta_k)$ is bounded away from 90°. This ensures that the iteration does not stagnate on a contour line. Based on the earlier results due to Wolfe [Wol69], [Wol71], Dennis and Schnabel essentially give the following result.

Theorem 2.3.11. [DS83, Theorem 6.3.3] *Consider a sequence* $(\mathbf{x}_k)$ *produced by an iteration of the form (2.3.1) with step lengths* α_k *that satisfy conditions (2.3.7) and (2.3.9). Suppose that* $f : \mathbb{R}^n \to \mathbb{R}$ *satisfies the assumptions*

$$\begin{cases} f \in C^1(\mathcal{D}, \mathbb{R}) \\ f \text{ is bounded from below in } \mathcal{D} \\ \nabla f \text{ is Lipschitz-continuous in } \mathcal{D}, \end{cases} \tag{2.3.12}$$

where $\mathcal{D} = \{\mathbf{x} \in \mathbb{R}^n | f(\mathbf{x}) < f(\mathbf{x}_0)\}$ *and* $\mathbf{x}_0$ *is the starting point of the iteration. If*

$$\limsup_{k\to\infty} \cos\theta_k > 0, \tag{2.3.13}$$

then

$$\liminf_{k\to\infty} \|\nabla f(\mathbf{x}_k)\| = 0.$$

This result states very general conditions for convergence of gradient descent methods, and thus it is a *global convergence* result. However, these conditions do not give any guarantee about the actual rate of convergence, which is covered by the *local convergence* theory. It should be also emphasized that this result only guarantees convergence to a stationary point.

2.3.2 The Moré and Thuente line search algorithm

The Moré and Thuente algorithm [MT94] is implemented in several numerical software packages such as Mathematica [Wol03] and OPT++ [Mez94]. Within a finite number of iterations, this algorithm generates step lengths α_k that satisfy conditions (2.3.7) and (2.3.8). In addition, conditions for convergence to an α^* such that $\phi'(\alpha^*) = 0$ are given in [MT94]. [6]

For the statement of this algorithm, we define the auxiliary function

$$\psi(\alpha) = \phi(\alpha) - \phi(0) - \mu\alpha\phi'(0)$$

and the set of admissible step lengths

$$T(\mu) = \{\alpha \in \mathbb{R} \mid \alpha \in T_s(\mu), \alpha \in T_c(\mu)\},$$

where

$$\begin{aligned} T_s(\mu) &= \{\alpha > 0 \mid \phi(\alpha) \le \phi(0) + \mu\alpha\phi'(0)\}, \\ T_c(\eta) &= \{\alpha > 0 \mid |\phi'(\alpha)| \le \eta|\phi'(0)|\} \end{aligned}$$

denote the sets of step lengths satisfying conditions (2.3.7) and (2.3.8), respectively. We also introduce the notation

$$T'(\mu, \eta) = \{\alpha > 0 \mid \alpha \in T_s(\mu), \alpha \in T_c(\eta)\}$$

for the case $\mu \neq \eta$. Note that $T(\mu) \subseteq T'(\mu, \eta)$ if $\mu \le \eta$. An outline of the Moré and Thuente algorithm is given in Algorithm 2.

[6] The step length bounds α_{min} and α_{max} introduced in [MT94] are omitted in this description. In what follows, we assume that that $\alpha_{min} = 0$ and $\alpha_{max} = \infty$.

Algorithm 2: Outline of the Moré and Thuente line search algorithm.

```
1 Choose the parameters $\mu, \eta \in ]0,1[$.
2 $I_0 \leftarrow [0, \infty]$.
3 for $k = 0, 1, \dots$ do
4     Terminate if $\alpha_t^k \in T'(\mu, \eta)$.
5     Choose $\alpha_t^{k+1} \in ]\alpha_l^k, \alpha_u^k[$ by interpolating $\psi$ (S1) or $\phi$ (S2).
6     If $\psi(\alpha_t^k) \leq 0$ and $\phi'(\alpha_t^k) > 0$, switch to S2.
7     Generate $I_{k+1}$ by using Algorithm 3 (S1) or Algorithm 4 (S2).
```

This algorithm generates a sequence of intervals $I_k \equiv [\alpha_l^k, \alpha_u^k]$ and a sequence of trial steps α_t^k such that $I_{k+1} \subseteq I_k$ and $\alpha_t^k \in I_k$ for all k in two stages. The first stage is aimed at finding a step length $\alpha^* \in T(\mu)$ such that $\psi'(\alpha^*) = 0$. If

$$\psi(\alpha_t^k) \leq 0, \quad \phi'(\alpha_t^k) > 0 \tag{2.3.14}$$

holds for some iterate α_t^k, the algorithm starts its second stage using α_t^k and I_k. At this stage, the algorithm attempts to find a step length $\alpha^* \in T'(\mu, \eta) \cap I_k$ such that $\phi'(\alpha^*) = 0$.

The *bracketing* algorithms for updating the intervals I_k are given in Algorithms 3 and 4 that correspond to stages 1 and 2, respectively. The secondary bracketing algorithm is otherwise identical to the first one, but with ψ replaced by ϕ. This algorithm is also illustrated in Figure 3. Note that we do not assume that α_l^k and α_u^k are ordered, and precisely speaking, $I_k \equiv [\alpha_l^k, \alpha_u^k]$ denotes an interval with unordered endpoints. [7]

Algorithm 3: The Moré and Thuente bracketing algorithm (S1).

```
1 if $\psi(\alpha_t) > \psi(\alpha_l)$ then $\alpha_l^+ \leftarrow \alpha_l$; $\alpha_u^+ \leftarrow \alpha_t$              /* Case 1. */
2 else
3     if $\psi'(\alpha_t)(\alpha_t - \alpha_l) < 0$ then $\alpha_l^+ \leftarrow \alpha_t$; $\alpha_u^+ \leftarrow \alpha_u$   /* Case 2. */
4     if $\psi'(\alpha_t)(\alpha_t - \alpha_l) > 0$ then $\alpha_l^+ \leftarrow \alpha_t$; $\alpha_u^+ \leftarrow \alpha_l$   /* Case 3. */
```

Algorithm 4: The Moré and Thuente bracketing algorithm (S2).

```
1 if $\phi(\alpha_t) > \phi(\alpha_l)$ then $\alpha_l^+ \leftarrow \alpha_l$; $\alpha_u^+ \leftarrow \alpha_t$              /* Case 1. */
2 else
3     if $\phi'(\alpha_t)(\alpha_t - \alpha_l) < 0$ then $\alpha_l^+ \leftarrow \alpha_t$; $\alpha_u^+ \leftarrow \alpha_u$   /* Case 2. */
4     if $\phi'(\alpha_t)(\alpha_t - \alpha_l) > 0$ then $\alpha_l^+ \leftarrow \alpha_t$; $\alpha_u^+ \leftarrow \alpha_l$   /* Case 3. */
```

[7] We omit superscripts k and use $+$ to denote $k+1$ for notational convenience.

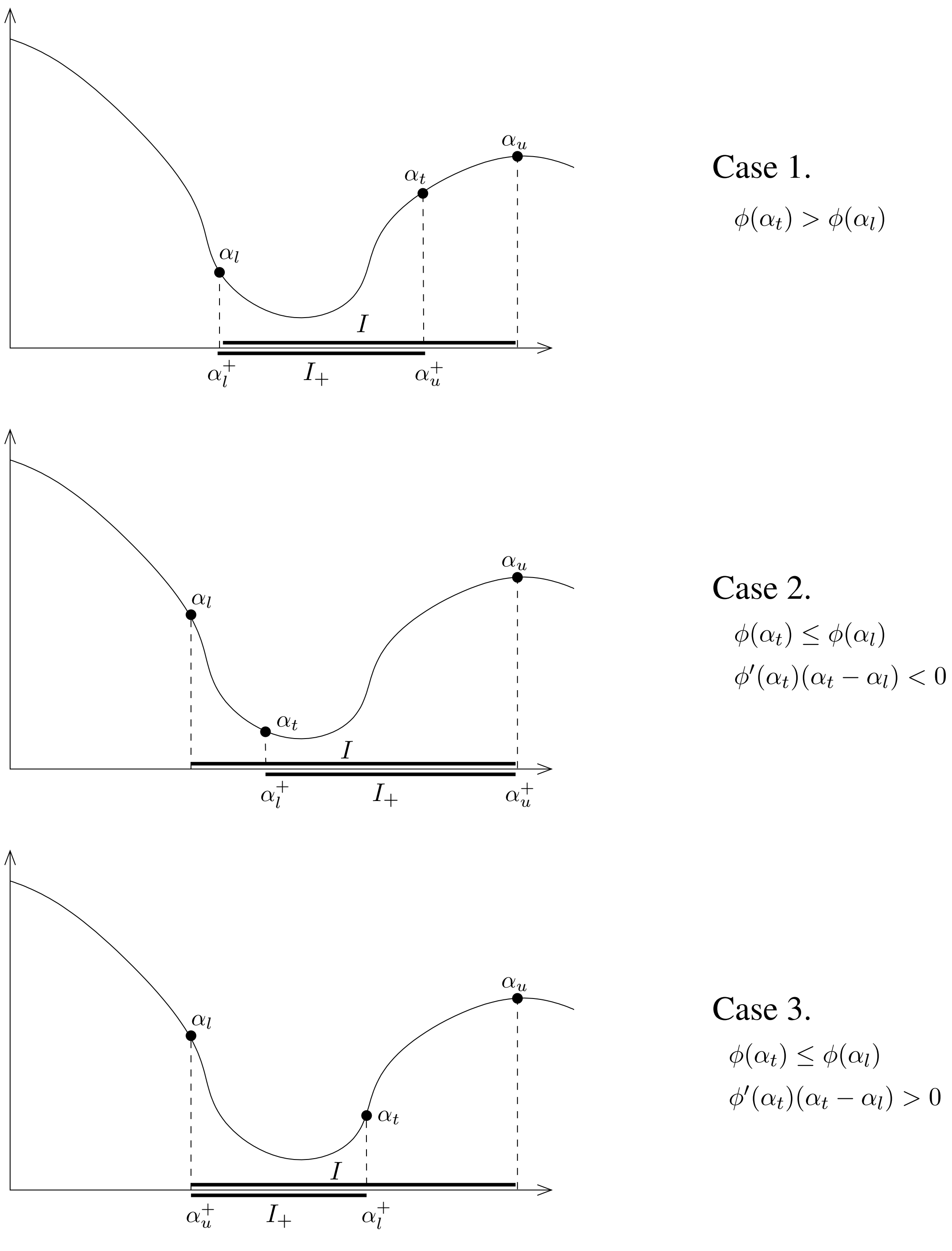


Figure 3: Bracketing steps of the Moré and Thuente algorithm.

Convergence results

Given an interval I satisfying the conditions stated below, the Moré and Thuente algorithm is motivated by the existence of a step length $\alpha^* \in I$ such that $\psi'(\alpha^*) = 0$ and $\alpha^* \in T(\mu)$. A similar result can also be established for ϕ. In particular, it can be shown that Algorithms 3 and 4 preserve conditions (2.3.16) and (2.3.18), respectively if I_0 satisfies them [MT94, p. 291].

Theorem 2.3.15. [MT94, Theorem 2.1] *Let $\mu \in]0,1[$ and let I be a closed interval with endpoints α_l and α_u. If the endpoints satisfy*

$$\psi(\alpha_l) \leq \psi(\alpha_u), \quad \psi(\alpha_l) \leq 0, \quad \psi'(\alpha_l)(\alpha_u - \alpha_l) < 0, \tag{2.3.16}$$

then there exists an $\alpha^* \in I$ such that $\psi(\alpha^*) \leq \psi(\alpha_l)$ and $\psi'(\alpha^*) = 0$. In particular, $\alpha^* \in T(\mu) \cap I$.

Theorem 2.3.17. [MT94, Theorem 3.2] *Let I be a closed interval with endpoints α_l and α_u. If the endpoints satisfy*

$$\phi(\alpha_l) \leq \phi(\alpha_u), \quad \phi'(\alpha_l)(\alpha_u - \alpha_l) < 0, \tag{2.3.18}$$

then there exists an $\alpha^* \in I$ such that $\phi(\alpha^*) \leq \phi(\alpha_l)$ and $\phi'(\alpha^*) = 0$.

The emphasis of the convergence results in [MT94] is on showing that with properly *safeguarded* trial steps, the lengths of the intervals I_k converge to zero when Algorithms 3 or 4 are successively applied to them. Consequently,

$$\lim_{k \to \infty} I_k = \{\alpha^*\},$$

where α^* satisfies the conditions stated in Theorem 2.3.15 or 2.3.17, respectively. [8]

For the following, we introduce the notations Algorithm 2/3 and 2/4 for stages 1. and 2., respectively. Moré and Thuente essentially proved the following result that motivates switching between bracketing algorithms when an iterate α_t^k satisfies conditions (2.3.14).

Theorem 2.3.19. [MT94, Theorem 3.1] *Let $\mu, \eta \in]0,1[$. Let (α_t^k) and (I_k), where $I_k \equiv [\alpha_l^k, \alpha_u^k]$ be the sequences produced by Algorithm 2/3. If α_t^k is the first iterate that satisfies*

$$\psi(\alpha_t^k) \leq 0, \quad \phi'(\alpha_t^k) > 0 \tag{2.3.20}$$

then $\alpha_l^k < \alpha_u^k$. Moreover, the interval

$$I^* \equiv [\alpha_l^k, \alpha_t^k]$$

contains an α^* that satisfies (2.3.7) and $\phi'(\alpha^*) = 0$, and thus $\alpha^* \in T'(\mu, \eta)$. Moreover, any $\alpha \in I^*$ with $\phi(\alpha) \leq \phi(\alpha_t^k)$ also satisfies (2.3.7).

[8] The term safeguarding refers to the rules that force the trial steps α_t^{k+1} to lie within the intervals I_k such that they are sufficiently far from the endpoints.

Remark 2.3.21. If conditions (2.3.14) hold for an iterate α_t^k, Case 2. in Algorithm 4 cannot hold because $\phi'(\alpha_t^k) > 0$. Hence, the interval I_{k+1} generated by Algorithm 4 from the interval I_k is the interval I^* of Theorem 2.3.19.

Remarks 2.3.22. Conditions (2.3.18) are satisfied by the interval I^* assuming that conditions (2.3.16) are satisfied by the interval I_k. The condition $\psi(\alpha_l^k) \leq \psi(\alpha_u^k)$ implies that $\phi(\alpha_l^k) < \phi(\alpha_u^k)$, since $\phi'(0) < 0$ and $\alpha_l^k < \alpha_u^k$. Moreover, the third condition in (2.3.16) with the assumption $\alpha_l^k < \alpha_u^k$ is equivalent to $\phi'(\alpha_l^k) < \mu\phi'(0)$, which implies that $\phi'(\alpha_l^k) < 0$, and thus $\phi'(\alpha_l^k)(\alpha_u^k - \alpha_l^k) < 0$. These conditions are also satisfied by $I_{k+1} \equiv I^*$, because Algorithm 4 preserves them.

The main convergence result in [MT94] is stated below. This result guarantees convergence of the Moré and Thuente line search to an $\alpha^* \in T'(\mu, \eta)$ such that $\phi'(\alpha^*) = 0$.

Theorem 2.3.23. [MT94, Theorem 3.3] *Let $\mu, \eta \in]0, 1[$. If Algorithm 2/3 generates an interval $I_k \equiv [\alpha_l^k, \alpha_u^k]$ and an iterate α_t^k satisfying conditions (2.3.14), then Algorithm 2/4 started from I_k and α_t^k terminates at an $\alpha_t^k \in T'(\mu, \eta)$, and the sequence (α_t^k) converges to a limit α^* such that $\phi'(\alpha^*) = 0$.*

Remark 2.3.24. Conditions (2.3.14) are in some cases not satisfied by any iterate. However, finite termination to an $\alpha^* \in T(\mu)$ is guaranteed under more general conditions [MT94, Theorem 2.3]. This is the desired result when $\mu \leq \eta$, and convergence to an α^* such that $\phi'(\alpha^*) = 0$ is not required.

Trial step selection

In this section we describe the rules for obtaining a new trial point α_t^+ by using the previous trial point α_t and the current interval I with endpoints α_l and α_u [MT94, p. 298-300]. In what follows, we will use f to denote the interpolated function (ψ or ϕ) and f' to denote its derivative. We introduce the following notation for the values of f and its derivative at the endpoints:

$$\begin{cases} f_l \equiv f(\alpha_l), & f_t \equiv f(\alpha_t), \quad f_u \equiv f(\alpha_u) \\ f'_l \equiv f'(\alpha_l), & f'_t \equiv f'(\alpha_t), \quad f'_u \equiv f'(\alpha_u) \end{cases}$$

In order to approximately locate the minimizer of f along the search direction, the trial step selection employs four types of interpolation polynomials:

1. a cubic polynomial that interpolates f_l, f_t, f'_l and f'_t
2. a quadratic polynomial that interpolates f_l, f_t and f'_l
3. a quadratic polynomial that interpolates f'_l and f'_t
4. a cubic polynomial that interpolates f_t, f_u, f'_t and f'_u

We denote the minimizers of these interpolation polynomials by α_c, α_q, α_s and α_e, respectively. The trial step selection of the Moré and Thuente algorithm is divided into four different cases:

Case 1. : $f_t > f_l$. Compute α_c and α_q and set

$$\alpha_t^+ = \begin{cases} \alpha_c, & \text{if } |\alpha_c - \alpha_l| < |\alpha_q - \alpha_l| \\ \frac{1}{2}(\alpha_q + \alpha_c), & \text{otherwise.} \end{cases}$$

Case 2. : $f_t \le f_l$ and $f_t' f_l' < 0$. Compute α_c and α_s and set

$$\alpha_t^+ = \begin{cases} \alpha_c, & \text{if } |\alpha_c - \alpha_t| \ge |\alpha_s - \alpha_t| \\ \alpha_s, & \text{otherwise.} \end{cases}$$

Case 3. : $f_t \le f_l$, $f_t' f_l' \ge 0$ and $|f_t'| \le |f_l'|$. Compute α_c and α_s and set

$$\alpha_t^+ = \begin{cases} \alpha_c, & \text{if } |\alpha_c - \alpha_t| < |\alpha_s - \alpha_t| \\ \alpha_s, & \text{otherwise.} \end{cases}$$

Restrict α_t^+ to the interval with endpoints α_t and α_u so that α_t^+ is sufficiently far from α_u. This is done by setting

$$\alpha_t^+ = \begin{cases} \min\{\alpha_t + \delta(\alpha_u - \alpha_t), \alpha_t^+\}, & \text{if } \alpha_t > \alpha_l, \\ \max\{\alpha_t + \delta(\alpha_u - \alpha_t), \alpha_t^+\}, & \text{otherwise,} \end{cases}$$

where $\delta = 0.66$.

Case 4. : $f_t \le f_l$, $f_t' f_l' \ge 0$ and $|f_t'| > |f_l'|$. Compute α_e and and set $\alpha_t^+ = \alpha_e$.

Choosing a trial step between α_l and α_t is referred to as *interpolation*. Cases 1. and 2., in which a local minimizer exists within this interval, fall into this category. On the other hand, choosing a trial value α_t^+ that lies between α_t and α_u is referred to as *extrapolation* [MT94]. Cases 3. and 4. fall into this category. In these cases, the algorithm extends the trial step length, because their conditions do not guarantee the existence of a minimizer. The use of these polynomials is illustrated in Figures 4-5.

Polynomial interpolation formulas

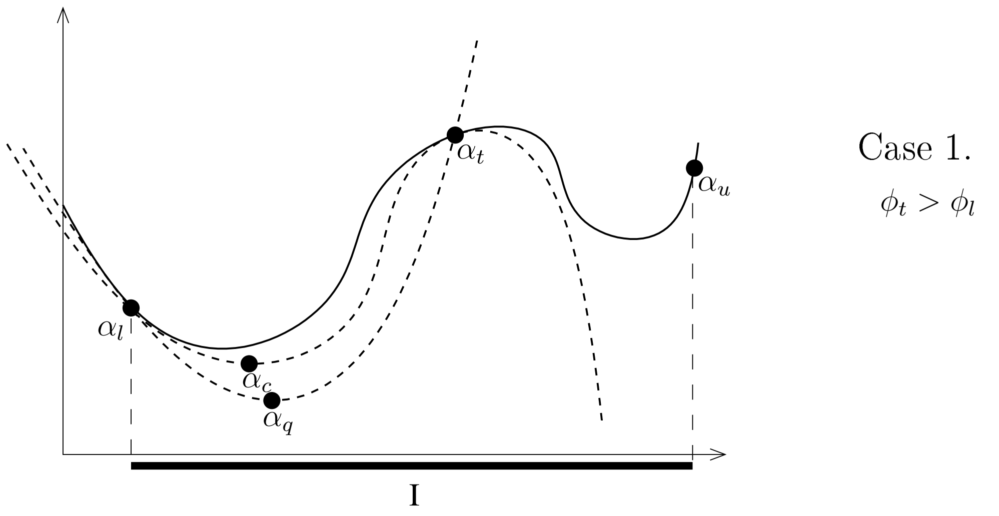
The trial step selection described in [MT94] is based on Fletcher's line search algorithm that employs a quadratic polynomial of the form

$$q(\alpha) = c_2\alpha^2 + c_1\alpha + c_0$$

to approximate ϕ [Fle80, p. 27]. As in the previous section, we denote the interpolated function (ϕ or ψ) by f.

Fletcher describes a formula that can be used for interpolating $f(a)$, $f'(a)$ and $f(b)$ within the given interval $[a, b]$. Requiring that $q(a) = f(a)$, $q'(a) = f'(a)$ and $q(b) = f(b)$ gives rise to the linear system of equations

$$\begin{bmatrix} a^2 & a & 1 \\ 2a & 1 & 0 \\ b^2 & b & 1 \end{bmatrix} \begin{bmatrix} c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} f(a) \\ f'(a) \\ f(b) \end{bmatrix}.$$



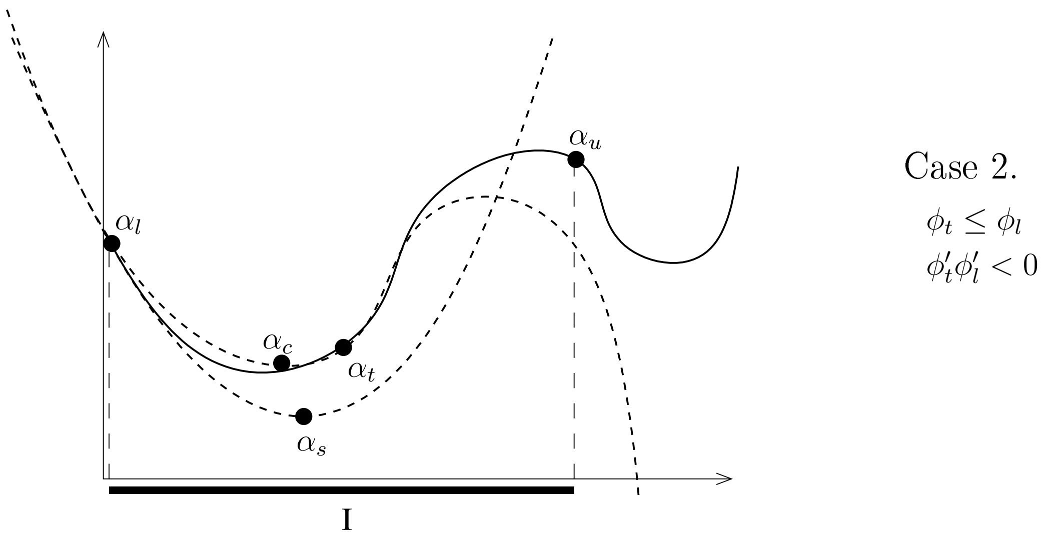
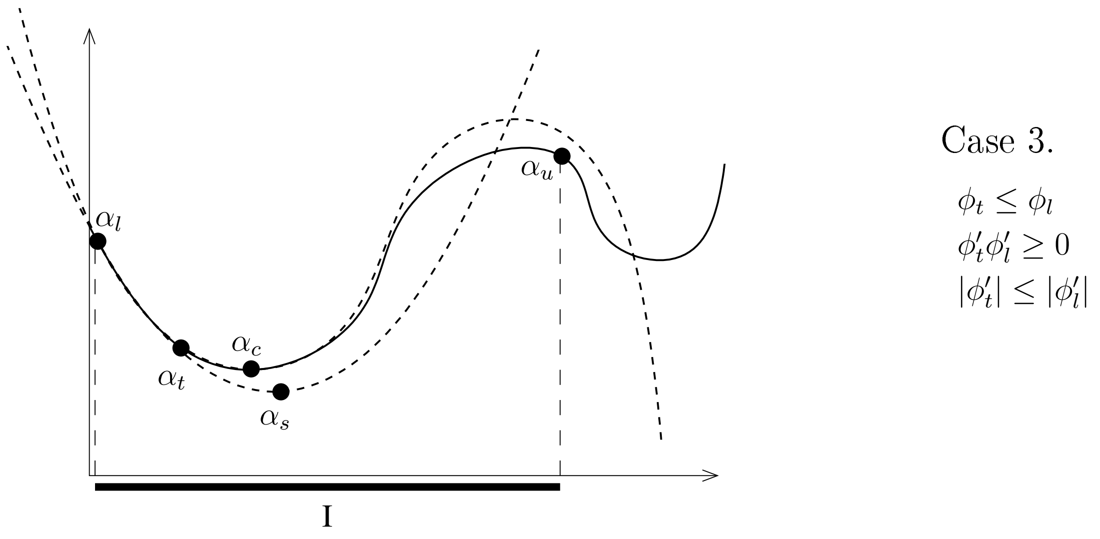


Figure 4: The Moré and Thuente interpolation polynomials: Case 1. above and Case 2. below.



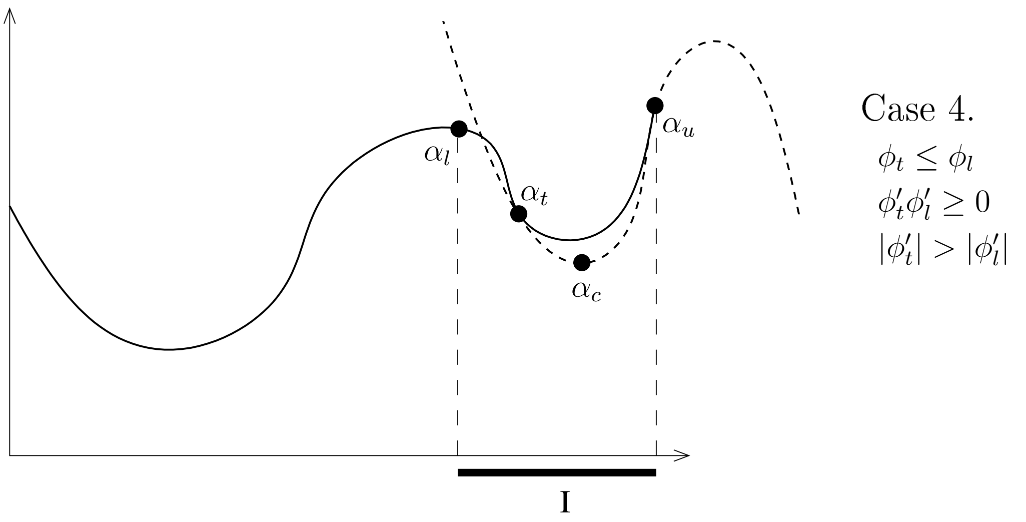


Figure 5: The Moré and Thuente extrapolation polynomials: Case 3. above and Case 4. below.

By substituting the solved coefficients to the expression

$$\alpha_q = -\frac{c_1}{2c_2} \tag{2.3.25}$$

that minimizes q, a straightforward calculation yields

$$\alpha_q = a + \frac{(b-a)^2 f'(a)}{2[f(a) - f(b) + (b-a)f'(a)]}. \tag{2.3.26}$$

Fletcher also describes a formula that can be used for interpolating $f'(a)$ and $f'(b)$. By requiring that $q'(a) = f'(a)$ and $q'(b) = f'(b)$, we obtain [9]

$$\begin{bmatrix} 2a & 1 \\ 2b & 1 \end{bmatrix} \begin{bmatrix} c_2 \\ c_1 \end{bmatrix} = \begin{bmatrix} f'(a) \\ f'(b) \end{bmatrix}$$

from which the coefficients, and thus the minimizer

$$\alpha_s = b + \frac{(b-a)f'(b)}{f'(a) - f'(b)} \tag{2.3.27}$$

for the polynomial q is obtained by substituting the solved coefficients to equation (2.3.25).

The minimizer of the cubic polynomial that interpolates $f(a)$, $f'(a)$, $f(b)$ and $f'(b)$ within the given interval $]a, b]$ is given by

$$\alpha_c = b - \frac{f'(b) + w - z}{f'(b) - f'(a) + 2w}(b-a),$$

where

$$z = \frac{3(f(a) - f(b))}{b-a} + f'(a) + f'(b)$$

and

$$w = \sqrt{z^2 - f'(a)f'(b)}.$$

This formula is described in [Ber99, p. 742-744].

2.3.3 The method of steepest descents

The method of steepest descents by Cauchy [Cau48] is based on the property that f decreases most rapidly along the direction of negative gradient. This gives rise to the iteration formula

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k),$$

where α_k is obtained by line minimization.

The steepest descent method is very sensitive to scaling of variables. Luenberger gives the following result for linear convergence rate of objective function values.

[9]The coefficient c_0 is left undetermined, since we are only interested in the location of the minimizer, and not the actual minimum value.

Theorem 2.3.28. [Lue84, p. 218-219] *Suppose that $f : \mathbb{R}^n \to \mathbb{R}$ is a strictly convex quadratic function of the form (2.1.3) with a minimizer $\mathbf{x}^*$. Then*

$$\frac{\|f(\mathbf{x}_{k+1}) - f(\mathbf{x}^*)\|}{\|f(\mathbf{x}_k) - f(\mathbf{x}^*)\|} \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^2, \tag{2.3.29}$$

where $\kappa = \frac{\lambda_{max}}{\lambda_{min}}$ is the condition number of matrix $\mathbf{A}$ with largest eigenvalue λ_{max} and smallest eigenvalue λ_{min}.

Luenberger also gives an extension of Theorem 2.3.28 for nonlinear functions [Lue84, p. 342-343]. Although these results account for the worst case, they show that the achievable convergece rates can be severely limited. To justify this claim, Akaike [Aka59, p. 12] gives examples in which this worst-case behaviour occurs on quadratic functions. On the other hand, steepest descent directions trivially satisfy condition (2.3.13), which guarantees global convergence.

2.3.4 Conjugate gradient methods

The *conjugate gradient methods* were originally developed by Hestenes and Stiefel [HS52] for solving linear equations. The development of *nonlinear conjugate gradient methods*, which is largely based on the theory of their linear counterparts, was pioneered by Fletcher and Reeves [FR64] and later complemented by Polak and Ribière [PR69].

The linear conjugate gradient methods are aimed for solving linear systems of the form $\mathbf{Ax} = \mathbf{b}$, or equivalently locating the minimizer $\mathbf{x}^*$ of a strictly convex quadratic function of the form (2.1.3) by solving the equation

$$\mathbf{r} \equiv \mathbf{b} - \mathbf{Ax}^* \equiv -\nabla f(\mathbf{x}^*) = \mathbf{0}, \tag{2.3.30}$$

where $\mathbf{r}$ is the *residual* of the linear system.

The primary motivation of these methods is that the vector space $\mathbb{R}^n$ can be spanned by n linearly independent search directions $\mathbf{d}_k$, i.e.

$$\mathbb{R}^n = \operatorname{span}\{\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_{n-1}\}.$$

If the difference vector between the starting point and the minimizer can be expressed as a linear combination of n linearly independent search directions such that

$$\mathbf{x}^* - \mathbf{x}_0 = \sum_{i=0}^{n-1} \delta_i \mathbf{d}_i \tag{2.3.31}$$

for a set of scalars $\{\delta_i\}_{i=0}^{n-1}$, the choice of search directions is optimal in that sense.

The key property that characterizes the linear conjugate gradient methods is that the search directions $\mathbf{d}_k$ are $\mathbf{A}$-*conjugated*, that is

$$\mathbf{d}_i^T \mathbf{A} \mathbf{d}_j = 0 \quad \forall i \neq j. \tag{2.3.32}$$

This property makes a construction of the form (2.3.31) possible, since $\mathbf{A}$-conjugated search directions are linearly independent [Lue84, p. 239].

The formulas for the iterates $\mathbf{x}_k$ and residual vectors $\mathbf{r}_k$ are given by

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + \alpha_k \mathbf{d}_k & (2.3.33) \\ \mathbf{r}_{k+1} &= \mathbf{r}_k + \alpha_k \mathbf{A} \mathbf{d}_k \equiv \mathbf{b} - \mathbf{A}\mathbf{x}_{k+1}, & (2.3.34) \end{aligned}$$

where α_k denotes the step length and the latter formula is obtained from the former by premultiplying by $\mathbf{A}$ and using the definition of residual.

By premultiplying equation (2.3.34) by $\mathbf{d}_k^T$ we obtain

$$\mathbf{d}_k^T \mathbf{r}_{k+1} = \mathbf{d}_k^T \mathbf{r}_k + \alpha_k \mathbf{d}_k^T \mathbf{A} \mathbf{d}_k$$

The minimizing step length α_k for a convex quadratic function is solved from the above equation by using identities (2.3.4) and (2.3.30), which yields

$$\alpha_k = -\frac{\mathbf{d}_k^T \mathbf{r}_k}{\mathbf{d}_k^T \mathbf{A} \mathbf{d}_k}. \tag{2.3.35}$$

The linear conjugate gradient method generates a set of search directions $\{\mathbf{d}_k\}$ and a set of residual vectors $\{\mathbf{r}_k\}$ that satisfy

$$\begin{cases} \mathbf{d}_i^T \mathbf{A} \mathbf{d}_j = 0 & \\ \mathbf{r}_i^T \mathbf{r}_j = 0 & \forall i < j \\ \mathbf{d}_i^T \mathbf{r}_j = 0 & \end{cases} \tag{2.3.36}$$

and $\operatorname{span}\{\mathbf{d}_k\} = \operatorname{span}\{\mathbf{r}_k\}$ [Lue84, p. 241-246]. In particular, it follows from this construction that the conjugate gradient method solves an n-dimensional quadratic problem in at most n steps, since $\operatorname{span}\{\mathbf{r}_i\}_{i=0}^{n-1} = \mathbb{R}^n$. These sets of vectors are constructed by applying (2.3.33), (2.3.34), (2.3.35) and the iteration formula for $\mathbf{d}_k$, which we will derive next. Instead of providing a formal proof by induction, we describe an outline of this construction.

The iteration formula for $\mathbf{d}_k$ is derived via *Gram-Schmidt conjugation* [Ber99, p. 134-138]. Assuming (2.3.36) for the residuals $\{\mathbf{r}_i\}_{i=0}^{k+1}$, they are orthogonal and thus linearly independent. Each new search direction is produced as a linear combination of the residual $\mathbf{r}_{k+1}$ computed from (2.3.34) and the previous search directions $\{\mathbf{d}_i\}_{i=0}^{k}$, that is

$$\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \sum_{i=0}^{k} \gamma_{ki} \mathbf{d}_i, \tag{2.3.37}$$

where $\mathbf{d}_0 = \mathbf{r}_0$. Postmultiplying the above equation by $\mathbf{A}\mathbf{d}_j$, where $j < k+1$, yields

$$\gamma_{k,j} = -\frac{\mathbf{r}_{k+1}^T \mathbf{A}\mathbf{d}_j}{\mathbf{d}_j^T \mathbf{A}\mathbf{d}_j} \tag{2.3.38}$$

with the assumption that the set of previously generated search directions $\{\mathbf{d}_i\}_{i=0}^{k}$ is $\mathbf{A}$-conjugated. On the other hand, assuming orthogonality of the vector set $\{\mathbf{r}_i\}_{i=0}^{k+1}$, premultiplying equation (2.3.34) by $\mathbf{r}_{j+1}$, where $j \leq k$, yields

$$\begin{aligned} & \mathbf{r}_{j+1}^T \mathbf{r}_{k+1} = \mathbf{r}_{j+1}^T \mathbf{r}_k + \alpha_k \mathbf{r}_{j+1}^T \mathbf{A}\mathbf{d}_k \\ \Longleftrightarrow\quad & \mathbf{r}_{j+1}^T \mathbf{A}\mathbf{d}_k = \tfrac{1}{\alpha_k}(-\mathbf{r}_{j+1}^T \mathbf{r}_k + \mathbf{r}_{j+1}^T \mathbf{r}_{k+1}) \end{aligned}$$

which is equivalent to

$$\mathbf{r}_{j+1}^T \mathbf{A}\mathbf{d}_k = \begin{cases} \frac{1}{\alpha_k}\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}, & j = k \\ 0, & \text{otherwise.} \end{cases}$$

By applying the above identity to equation (2.3.38) and substituting α_k from equation (2.3.35), we have

$$\gamma_{k+1} = \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{d}_k^T \mathbf{r}_k},$$

where $\gamma_{k+1} \equiv \gamma_{k,k}$. Finally, by premultiplying equation (2.3.37) by $\mathbf{r}_{k+1}^T$ and applying the assumed orthogonality condition $\mathbf{d}_i \mathbf{r}_j = 0$ for all $i < j$, we obtain the identity $\mathbf{r}_{k+1}^T \mathbf{d}_{k+1} = \mathbf{r}_{k+1}^T \mathbf{r}_{k+1}$, and thus

$$\gamma_{k+1} = \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}. \tag{2.3.39}$$

Consequently, equation (2.3.37) then reduces to the form

$$\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \gamma_{k+1}\mathbf{d}_k. \tag{2.3.40}$$

Two modifications are needed in order to apply this method to nonlinear functions. Firstly, formula (2.3.35) is replaced by an iterative line search procedure. Secondly, the residual vectors $\mathbf{r}_k$ are replaced by negative gradient vectors which we denote by $-\mathbf{g}_k$. Instead of applying (2.3.34), the function gradients are evaluated at each iterate $\mathbf{x}_k$. The *Fletcher-Reeves formula* [FR64] is obtained from (2.3.39) with the substitution $\mathbf{r}_k \equiv -\mathbf{g}_k$, that is

$$\gamma_{k+1} = \frac{\mathbf{g}_{k+1}^T \mathbf{g}_{k+1}}{\mathbf{g}_k^T \mathbf{g}_k}. \tag{2.3.41}$$

Polak and Ribière [PR69] later suggested a slightly modified formula

$$\gamma_{k+1} = \frac{(\mathbf{g}_{k+1} - \mathbf{g}_k)^T \mathbf{g}_{k+1}}{\mathbf{g}_k^T \mathbf{g}_k} \tag{2.3.42}$$

that generally exhibits faster convergence rates than its predecessor on non-linear functions. These formulas are equivalent for quadratic functions, since $\mathbf{g}_{k+1}^T\mathbf{g}_k = \mathbf{r}_{k+1}^T\mathbf{r}_k = 0$ by equation (2.3.36).

Periodic restarting is a commonly used strategy to accelerate convergence of conjugate gradient methods [Pow77]. Resetting the search direction back to the steepest descent direction $-\mathbf{g}_k$ every $n+1$ iterations was suggested by Fletcher and Reeves [FR64]. This gives rise to the modification of formula (2.3.41), that is

$$\gamma_{k+1} = \begin{cases} 0, & k = cn \text{ for some } c \in \mathbb{N} \\ \frac{\mathbf{g}_{k+1}^T\mathbf{g}_{k+1}}{\mathbf{g}_k^T\mathbf{g}_k}, & \text{otherwise.} \end{cases} \tag{2.3.43}$$

Powell [Pow83] constructed a function on which the Polak-Ribière method cycles infinitely with $\gamma_k < 0$. This gives rise to the modified update formula

$$\gamma_{k+1} = \max\{\frac{(\mathbf{g}_{k+1} - \mathbf{g}_k)^T\mathbf{g}_{k+1}}{\mathbf{g}_k^T\mathbf{g}_k}, 0\}. \tag{2.3.44}$$

Outline of implementation

The above derivations of the Fletcher-Reeves and Polak-Ribière conjugate gradient algorithms with restarting are summarized in Algorithms 5 and 6, respectively.

Algorithm 5: Fletcher-Reeves conjugate gradient iteration step.

```
1 Obtain an $\alpha_k > 0$ such that $\frac{d}{d\alpha_k}f(\mathbf{x}_k + \alpha_k\mathbf{d}_k) \approx 0$.
2 $\mathbf{g}_{k+1} \leftarrow \nabla f(\mathbf{x}_{k+1})$
3 if $c = n$ then $\gamma_{k+1} \leftarrow 0$; $c \leftarrow 0$
4 else $\gamma_{k+1} \leftarrow \frac{\mathbf{g}_{k+1}^T\mathbf{g}_{k+1}}{\mathbf{g}_k^T\mathbf{g}_k}$          /* eq. (2.3.43) */
5 $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k\mathbf{d}_k$          /* eq. (2.3.33) */
6 $\mathbf{d}_{k+1} \leftarrow -\mathbf{g}_{k+1} + \gamma_{k+1}\mathbf{d}_k$          /* eq. (2.3.40) */
7 $c \leftarrow c + 1$
```

Algorithm 6: Polak-Ribière conjugate gradient iteration step.

```
1 Obtain an $\alpha_k > 0$ such that $\frac{d}{d\alpha_k}f(\mathbf{x}_k + \alpha_k\mathbf{d}_k) \approx 0$.
2 $\mathbf{g}_{k+1} \leftarrow \nabla f(\mathbf{x}_{k+1})$
3 $\gamma_{k+1} \leftarrow \max\{\frac{(\mathbf{g}_{k+1}-\mathbf{g}_k)^T\mathbf{g}_{k+1}}{\mathbf{g}_k^T\mathbf{g}_k}, 0\}$          /* eq. (2.3.44) */
4 $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k\mathbf{d}_k$          /* eq. (2.3.33) */
5 $\mathbf{d}_{k+1} \leftarrow -\mathbf{g}_{k+1} + \gamma_{k+1}\mathbf{d}_k$          /* eq. (2.3.40) */
```

Convergence results

Based on the earlier results by Al-Baali [AB85] for the Fletcher-Reeves method, Gilbert and Nocedal [GN92] established global convergence theory of the Polak-Ribière method. They essentially proved the following global convergence result.

Theorem 2.3.45. [GN92, Theorems 3.2 and 4.3] *Consider a sequence* $(\mathbf{x}_k)$ *generated by Algorithm 5. Suppose the iteration and the objective function* $f : \mathbb{R}^n \to \mathbb{R}$ *satisfy the assumptions*

$$\begin{cases} \textit{The step lengths } \alpha_k \textit{ satisfy } (2.3.7) \textit{ and } (2.3.8). \\ \textit{Periodic restarting is not used.} \\ f \in C^1(\mathcal{L}, \mathbb{R}). \\ \mathcal{L} \textit{ is bounded.} \\ \nabla f \textit{ is Lipschitz-continuous in } \mathcal{L}. \end{cases} \tag{2.3.46}$$

where $\mathcal{L} = \{\mathbf{x} \in \mathbb{R}^n | f(\mathbf{x}) \leq f(\mathbf{x}_0)\}$ *and* $\mathbf{x}_0$ *is the starting point of the iteration. Then*

$$\liminf_{k\to\infty} \|\nabla f(\mathbf{x}_k)\| = 0. \tag{2.3.47}$$

Furthermore, if the sufficient descent condition

$$\mathbf{g}_k^T \mathbf{d}_k \leq -\chi \|\mathbf{g}_k\|^2, \quad \chi \in]0, 1[\tag{2.3.48}$$

is satisfied for all k, then (2.3.47) also holds for Algorithm 6.

Remarks 2.3.49. Condition (2.3.48) can be satisfied by any iterative line search algorithm that satisfies

$$\lim_{i\to\infty} \alpha_i = \alpha^*, \quad \phi'(\alpha^*) = 0.$$

Suppose that such a limit exists. By premultiplying (2.3.40) by $\mathbf{g}_{k+1}$, recalling the notation $\mathbf{g}_{k+1} \equiv -\mathbf{r}_{k+1} \equiv -\nabla f(\mathbf{x}_{k+1})$, using the definition of ϕ and taking the limit $i \to \infty$, we obtain

$$\begin{aligned} \lim_{i\to\infty} \mathbf{g}_{k+1}^T \mathbf{d}_{k+1} &= \lim_{i\to\infty} (-\|\mathbf{g}_{k+1}\|^2 + \gamma_{k+1} \mathbf{g}_{k+1}^T \mathbf{d}_k) \\ &\equiv \lim_{i\to\infty} (-\|\mathbf{g}_{k+1}\|^2 + \gamma_{k+1} \phi'(\alpha_i)), \\ &= -\|\mathbf{g}_{k+1}\|^2 \\ &< -\chi \|\mathbf{g}_{k+1}\|^2 \end{aligned}$$

since $\chi \in]0, 1[$ and $\phi'(\alpha_i) \equiv \frac{d}{d\alpha_i} f(\mathbf{x}_k + \alpha_i \mathbf{d}_k) = \nabla f(\mathbf{x}_{k+1})^T \mathbf{d}_k$. In particular, there exists an iterate α_i that satisfies (2.3.48). Also note that a sufficient condition for (2.3.48) is that $\phi'(\alpha^*) \leq 0$.

Unfortunately, the local convergence properties of nonlinear conjugate gradient methods remain poorly understood. Among the most important results is the following result by Cohen [Coh72], which states that the Fletcher-Reeves and Polak-Ribière conjugate gradient methods with periodic restarts converge n-step quadratically. This result is however based on the assumption that exact line minimization is used.

Theorem 2.3.50. [Coh72, p. 250-255] *Let $f : \mathbb{R}^n \to \mathbb{R}$, $f \in C^3(\mathbb{R}^n, \mathbb{R})$. Suppose that there exists constants $0 < m < M < \infty$ such that $\mathbf{H}_f$ satisfies*

$$m\|\mathbf{y}\|^2 \leq \mathbf{y}^T\mathbf{H}_f(\mathbf{x})\mathbf{y} \leq M\|\mathbf{y}\|^2 \quad \textit{for all} \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^n.$$

Then the sequence $(\mathbf{x}_k)$ produced by Algorithms 5-6 with exact line minimization is n-step quadratically convergent, i.e. there exists a constant $C > 0$ such that

$$\limsup_{k\to\infty} \frac{\|\mathbf{x}_{pk+n} - \mathbf{x}^*\|}{\|\mathbf{x}_{pk} - \mathbf{x}^*\|^2} \leq C,$$

if the iteration is restarted every p steps, where $p \geq n$.

2.3.5 The Newton method

The variant of the Newton method for unconstrained minimization is formulated for minimizing the quadratic approximation

$$f(\mathbf{x}_k + \mathbf{h}_k) \approx f(\mathbf{x}_k) + \mathbf{h}_k^T\nabla f(\mathbf{x}_k) + \frac{1}{2}\mathbf{h}_k^T\mathbf{H}_f(\mathbf{x}_k)\mathbf{h}_k \tag{2.3.51}$$

of a C^2-function with respect to $\mathbf{h}_k$. For the formulation of this method, we introduce the auxiliary function $F(\mathbf{h}_k) \equiv f(\mathbf{x}_k + \mathbf{h}_k)$.

Assuming positive definite Hessian, the auxiliary function F attains its minimum value when

$$\nabla F(\mathbf{h}_k) = \mathbf{0}.$$

By using the quadratic approximation of ∇F, that is

$$\nabla F(\mathbf{h}_k) \approx \nabla f(\mathbf{x}_k) + \mathbf{H}_f(\mathbf{x}_k)\mathbf{h}_k, \tag{2.3.52}$$

we obtain

$$\mathbf{H}_f(\mathbf{x}_k)\mathbf{h}_k = -\nabla f(\mathbf{x}_k), \tag{2.3.53}$$

which gives rise to the iteration formula

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + \mathbf{h}_k \\ &= \mathbf{x}_k - \mathbf{H}_f(\mathbf{x}_k)^{-1}\nabla f(\mathbf{x}_k), \end{aligned} \tag{2.3.54}$$

assuming that $\mathbf{H}_f(\mathbf{x})$ is nonsingular, which follows from its assumed positive definiteness.

Although the Newton method can exhibit quadratic convergence rate if the starting point is chosen sufficiently close to a minimizer [DS83, Theorem 5.2.1], its global convergence is not in general guaranteed. A common remedy is to use the step $\mathbf{h}_k$ obtained from (2.3.53) as a search direction and use a line search method to obtain the step length. This gives rise to a modification of iteration formula (2.3.54), that is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{H}_f(\mathbf{x}_k)^{-1} \nabla f(\mathbf{x}_k). \tag{2.3.55}$$

By following the derivation of equation (2.3.4) and assuming positive definiteness of $\mathbf{H}_f(\mathbf{x}_k)$, and consequently positive definiteness of its inverse, we obtain

$$\begin{aligned} & \tfrac{d}{d\alpha_k} f(\mathbf{x}_k - \alpha_k \mathbf{H}_f(\mathbf{x}_k)^{-1} \nabla f(\mathbf{x}_k))|_{\alpha_k=0} \\ = & -\nabla f(\mathbf{x}_k)^T \mathbf{H}_f(\mathbf{x}_k)^{-1} \nabla f(\mathbf{x}_k) < 0, \end{aligned}$$

from which the descent property of $\mathbf{h}_k$ follows. Unfortunately, this strategy alone is not successful in practice, since positive definiteness of the Hessian is not guaranteed without additional measures.

It can also be shown that the Newton method is invariant under linear transformations of the form (2.1.18) [Fle80, Theorem 3.3.1]. The Newton iteration (2.3.54) in the transformed variables $\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b}$ is given by

$$\mathbf{y}_{k+1} = \mathbf{y}_k - \tilde{\mathbf{H}}_f(\mathbf{y}_k)^{-1} \nabla \tilde{f}(\mathbf{y}_k). \tag{2.3.56}$$

By substiting equations (2.1.18) and (2.1.23) to (2.3.56), we obtain

$$\begin{aligned} \mathbf{y}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{b} - \mathbf{A}\mathbf{H}_f(\mathbf{x}_k)^{-1} \mathbf{A}^T \mathbf{A}^{-T} \nabla f(\mathbf{x}_k) \\ &= \mathbf{A}[\mathbf{x}_k - \mathbf{H}_f(\mathbf{x}_k)^{-1} \nabla f(\mathbf{x}_k)] + \mathbf{b} \\ &= \mathbf{A}\mathbf{x}_{k+1} + \mathbf{b}, \end{aligned}$$

which verifies the claim, provided that it holds for $k = 0$. This also holds for iterations of the form (2.3.55), if the used line search method is invariant.

The Cholesky factorization

The need to solve equations of the form (2.3.53) arises in the Newton method and its variants. Any symmetric positive definite matrix $\mathbf{A}$ has a *Cholesky factorization* of the form

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T, \tag{2.3.57}$$

where $\mathbf{L}$ is a nonsigular lower-triangular matrix with strictly positive diagonal elements. By equating the elements in (2.3.57), one can obtain

$$l_{ii} = \sqrt{a_{ii} - \sum_{j=1}^{i-1} l_{ij}^2}, \tag{2.3.58}$$

$$l_{ij} = \frac{1}{l_{jj}}(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}), \quad i > j. \tag{2.3.59}$$

By using this factorization, the solution of a symmetric positive definite linear system $\mathbf{Ax} = \mathbf{b}$ is split into a pair of triangular systems

$$\mathbf{Ly} = \mathbf{b} \quad \text{and} \quad \mathbf{L}^T\mathbf{x} = \mathbf{y}$$

that are solved by forward and back substitution. This method for solving linear systems is numerically more stable and efficient than the standard Gaussian elimination. [GMW91, p. 108-112]

2.3.6 Quasi-Newton methods

The class of *quasi-Newton* methods consists of numerous different methods based on the modified Newton iteration of the form (2.3.55). Instead of explicitly computing the Hessian, they maintain a first-order approximation of either the Hessian or its inverse which we denote by $\mathbf{B}_k$ and $\mathbf{S}_k$, respectively. We also introduce the notation

$$\mathbf{p}_k \equiv \mathbf{x}_{k+1} - \mathbf{x}_k \equiv \alpha_k \mathbf{h}_k \tag{2.3.60}$$

$$\mathbf{q}_k \equiv \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k), \tag{2.3.61}$$

where $\mathbf{h}_k = -\mathbf{H}_f(\mathbf{x}_k)^{-1}\nabla f(\mathbf{x}_k)$ is the Newton step from equation (2.3.54). By using approximation (2.3.52) and replacing $\mathbf{h}_k$ with $\mathbf{p}_k$, we obtain

$$\nabla f(\mathbf{x}_k + \mathbf{p}_k) - \nabla f(\mathbf{x}_k) \approx \mathbf{H}_f(\mathbf{x}_k)\mathbf{p}_k. \tag{2.3.62}$$

By using the definition of $\mathbf{q}_k$ and assuming that f is quadratic, equation (2.3.62) reduces to

$$\mathbf{H}_f(\mathbf{x}_k)\mathbf{p}_k = \mathbf{q}_k \quad \text{or} \quad \mathbf{p}_k = \mathbf{H}_f(\mathbf{x}_k)^{-1}\mathbf{q}_k. \tag{2.3.63}$$

The *Sherman-Morrison-Woodbury identity* [GvL89, p. 51] states that a rank-k update of a $n \times n$ matrix $\mathbf{A}$ of the form

$$\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{U}\mathbf{V}^T, \tag{2.3.64}$$

where $\mathbf{U}, \mathbf{V} \in \mathbb{R}^{n\times k}$, $k \leq n$, has an inverse given by

$$\tilde{\mathbf{A}}^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I} + \mathbf{V}^T\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^T\mathbf{A}^{-1}. \tag{2.3.65}$$

Via this identity, update formulas for the Hessian and its inverse can be derived equivalently, and thus they are treated identically in the literature. In what follows, we therefore formulate the updates for both $\mathbf{B}_k$ and $\mathbf{S}_k$ in a parallel fashion.

The Hessian or inverse Hessian approximation is updated by adding a rank-1 or rank-2 correction matrix such that

$$\begin{cases} \mathbf{B}_{k+1} = \mathbf{B}_k + \mathbf{C}_k \\ \mathbf{S}_{k+1} = \mathbf{S}_k + \mathbf{C}_k. \end{cases} \tag{2.3.66}$$

In analogy with equations (2.3.63), the resulting matrices $\mathbf{B}_{k+1}$ and $\mathbf{S}_{k+1}$ are required to satisfy the *quasi-Newton condition*

$$\begin{cases} \mathbf{B}_{k+1}\mathbf{p}_k = \mathbf{q}_k \\ \mathbf{p}_k = \mathbf{S}_{k+1}\mathbf{q}_k. \end{cases} \tag{2.3.67}$$

By using the approximations $\mathbf{B}_k$ or $\mathbf{S}_k$ from above, iteration step (2.3.55) is written to the form

$$\begin{cases} \mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{B}_k^{-1} \nabla f(\mathbf{x}_k) \\ \mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{S}_k \nabla f(\mathbf{x}_k). \end{cases} \tag{2.3.68}$$

In practice, the former iteration step is computed by solving the equation

$$\mathbf{B}_k \mathbf{d}_k = -\nabla f(\mathbf{x}_k) \tag{2.3.69}$$

for $\mathbf{d}_k$ by Cholesky factorization and then applying $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$.

An additional condition

$$\mathbf{p}_k^T \mathbf{q}_k > 0 \quad \forall k \in \mathbb{N} \tag{2.3.70}$$

is required, since $\mathbf{B}_{k+1}$ and $\mathbf{S}_{k+1}$ in equations (2.3.67) cannot be positive definite if $\mathbf{p}_k^T \mathbf{q}_k \leq 0$. However, this condition is satisfied if step lengths satisfying condition (2.3.9) are used [NW99, p. 195].

Derivation of update formulas

Without imposing additional constraints, the solutions to equations (2.3.67) are not unique. The *variational method* due to Greenstadt [Gre70] is formulated as seeking a minimal correction matrix that preserves symmetry and guarantees that $\mathbf{B}_{k+1}$ or $\mathbf{S}_{k+1}$ satisfies condition (2.3.67), i.e.

$$\left.\begin{array}{rl} \mathbf{C}_k = & \arg \min\limits_{\mathbf{C} \in \mathbb{R}^{n \times n}} \|\mathbf{C}\|_{F,W} \\ \text{s.t.} & \mathbf{B}_{k+1}\mathbf{p}_k = \mathbf{q}_k, \\ & \mathbf{C}_k = \mathbf{C}_k^T \end{array}\right\} \text{ or } \left\{\begin{array}{rl} \mathbf{C}_k = & \arg \min\limits_{\mathbf{C} \in \mathbb{R}^{n \times n}} \|\mathbf{C}\|_{F,W} \\ \text{s.t.} & \mathbf{S}_{k+1}\mathbf{q}_k = \mathbf{p}_k, \\ & \mathbf{C}_k = \mathbf{C}_k^T \end{array}\right. \tag{2.3.71}$$

where $\| \cdot \|_{F,W}$ denotes the *weighted Frobenius norm*

$$\|\mathbf{C}\|_{F,\mathbf{W}} = Tr(\mathbf{WCWC}^T). \tag{2.3.72}$$

The constrained minimization problem (2.3.71) can be solved by the method of *Lagrange multipliers* [Gre70, p. 4-5], and it has an unique solution up to the choice of the symmetric and positive definite weighting matrix $\mathbf{W}$. Greenstadt derived several commonly used update formulas with different choices of $\mathbf{W}$ [Gre70, eq. (2-25)].

The BFGS formula and the Broyden family

The BFGS formula is generally considered to be the most robust and best performing of the known quasi-Newton update formulas. In particular, Fletcher emphasizes its comparative robustness with inexact line searches [Fle80, Table 3.5.2]. It was discovered by Broyden [Bro70], Fletcher [Fle70], Goldfarb [Gol70], and Shanno [Sha70]. Goldfarb, for example, derived it by applying Greenstadt's method. The BFGS update formulas for Hessian and inverse Hessian approximations are given by [Kel99, p. 71-72]

$$\begin{cases} \mathbf{B}_{k+1} = \mathbf{B}_k + \frac{\mathbf{q}_k\mathbf{q}_k^T}{\mathbf{q}_k^T\mathbf{p}_k} - \frac{\mathbf{B}_k\mathbf{p}_k\mathbf{p}_k^T\mathbf{B}_k}{\mathbf{p}_k^T\mathbf{B}_k\mathbf{p}_k} \\ \mathbf{S}_{k+1} = (\mathbf{I} - \frac{\mathbf{p}_k\mathbf{q}_k^T}{\mathbf{q}_k^T\mathbf{p}_k})\mathbf{S}_k(\mathbf{I} - \frac{\mathbf{q}_k\mathbf{p}_k^T}{\mathbf{q}_k^T\mathbf{p}_k}) + \frac{\mathbf{p}_k\mathbf{p}_k^T}{\mathbf{q}_k^T\mathbf{p}_k}. \end{cases} \tag{2.3.73}$$

These two are equivalent via equation (2.3.65). For example, the BFGS formula for $\mathbf{B}_k$ can be written in the form (2.3.64) for rank-2 formulas,

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \begin{bmatrix} \mathbf{q}_k/\alpha & -\mathbf{B}_k\mathbf{p}_k/\beta \end{bmatrix} \begin{bmatrix} \mathbf{q}_k^T \\ (\mathbf{B}_k\mathbf{p}_k)^T \end{bmatrix} \equiv \mathbf{B}_k + \mathbf{U}\mathbf{V}^T, \tag{2.3.74}$$

where $\alpha = \mathbf{q}_k^T\mathbf{p}_k$ and $\beta = \mathbf{p}_k^T\mathbf{B}_k\mathbf{p}_k$. A straightforward but laborious computation by applying formula (2.3.65) gives the BFGS update for $\mathbf{S}_k$ by substituting $\mathbf{S}_{k+1} \equiv \mathbf{B}_{k+1}^{-1}$ and $\mathbf{S}_k \equiv \mathbf{B}_k^{-1}$.

The BFGS formula belongs to a more general class of formulas that is referred to as the *Broyden family* in the literature [Fle80, p. 48-54]. The generic form of a Broyden family update formula for $\mathbf{B}_k$ is given by

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{\mathbf{q}_k\mathbf{q}_k^T}{\mathbf{q}_k^T\mathbf{p}_k} - \frac{\mathbf{B}_k\mathbf{p}_k\mathbf{p}_k^T\mathbf{B}_k}{\mathbf{p}_k^T\mathbf{B}_k\mathbf{p}_k} + \phi(\mathbf{p}_k^T\mathbf{B}_k\mathbf{p}_k)\mathbf{v}_k\mathbf{v}_k^T,$$

where $\phi \in [0, 1]$ and

$$\mathbf{v}_k = \frac{\mathbf{q}_k}{\mathbf{q}_k^T\mathbf{p}_k} - \frac{\mathbf{B}_k\mathbf{p}_k}{\mathbf{p}_k^T\mathbf{B}_k\mathbf{p}_k}.$$

The BFGS formula is obtained for $\phi = 0$. The corresponding inverse Hessian BFGS update can be obtained by applying (2.3.65) to this formula, or making the interchanges $\mathbf{B}_{k+1} \leftrightarrow \mathbf{S}_{k+1}$, $\mathbf{p}_k \leftrightarrow \mathbf{q}_k$ and $\phi = 1$. As the Broyden family methods generate conjugated search directions, they have the quadratic termination property [Lue84, p. 267-268]. With exact line minimization, they also generate identical iterates on nonlinear functions [Dix72].

Assuming that $\mathbf{B}_0$ is positive definite, the BFGS formula preserves this property, which guarantees the descent property of search directions.

Theorem 2.3.75. [Kel99, Lemma 4.1.2] *Suppose that $\mathbf{B}_k$ is positive definite for some k, and $\mathbf{B}_{k+1}$ is obtained by the BFGS update formula (2.3.73). Then $\mathbf{B}_{k+1}$ is positive definite if and only if $\mathbf{p}_k^T\mathbf{q}_k > 0$.*

This result also applies to the inverse Hessian update via identity (2.3.65) and the property that the inverse of a positive definite matrix is also positive definite.

Outline of implementation

Generic quasi-Newton iteration steps using Hessian and inverse Hessian approximations are summarized in Algorithms 7 and 8, respectively.

Algorithm 7: Quasi-Newton iteration step, Hessian update.

```
1 Compute the Cholesky factorization $\mathbf{B}_k = \mathbf{L}\mathbf{L}^T$.
2 Solve $\mathbf{L}\mathbf{L}^T\mathbf{d}_k = -\nabla f(\mathbf{x}_k)$.        /* eq. (2.3.69) */
3 Obtain an $\alpha_k > 0$ such that $\frac{d}{d\alpha_k} f(\mathbf{x}_k + \alpha_k \mathbf{d}_k) \approx 0$.
4 $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k$
5 $\mathbf{p}_k \leftarrow \mathbf{x}_{k+1} - \mathbf{x}_k$        /* eq. (2.3.60) */
6 $\mathbf{q}_k \leftarrow \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$        /* eq. (2.3.61) */
7 Generate $\mathbf{B}_{k+1}$ such that $\mathbf{B}_{k+1}\mathbf{p}_k = \mathbf{q}_k$.        /* eq. (2.3.67) */
```

Algorithm 8: Quasi-Newton iteration step, inverse Hessian update.

```
1 $\mathbf{d}_k \leftarrow -\mathbf{S}_k \nabla f(\mathbf{x}_k)$
2 Obtain an $\alpha_k > 0$ such that $\frac{d}{d\alpha_k} f(\mathbf{x}_k + \alpha_k \mathbf{d}_k) \approx 0$.
3 $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \alpha_k \mathbf{S}_k \nabla f(\mathbf{x}_k)$        /* eq. (2.3.68) */
4 $\mathbf{p}_k \leftarrow \mathbf{x}_{k+1} - \mathbf{x}_k$        /* eq. (2.3.60) */
5 $\mathbf{q}_k \leftarrow \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$        /* eq. (2.3.61) */
6 Generate $\mathbf{S}_{k+1}$ such that $\mathbf{S}_{k+1}\mathbf{q}_k = \mathbf{p}_k$.        /* eq. (2.3.67) */
```

Solving equation (2.3.69) typically requires $O(n^3)$ floating-point operations per iteration, which makes the Hessian update less efficient than the inverse Hessian update with $O(n^2)$ floating-point operations per iteration. Some authors however favor the former method claiming its better numerical stability [GMW81, p. 122-123]. In particular, Cholesky factorizations allow incorporating additional modifications, see e.g. [GM74], that prevent Hessian approximations from losing their positive definiteness due to loss of precision.

Convergence results

Powell [Pow76] was the first to establish the conditions for global convergence of the BFGS method. Byrd, Nocedal and Yuan [BNY87] extended the earlier convergence theory of quasi-Newton methods to the Broyden family. They essentially proved the following global convergence result on strictly convex functions.

Theorem 2.3.76. [BNY87, Theorem 3.1] *Consider a sequence* $(\mathbf{x}_k)$ *produced by Algorithm 7 or 8. Suppose that the iteration and the objective function*

$f : \mathbb{R}^n \to \mathbb{R}$ satisfy

$$\begin{cases} f \in C^2(\mathbb{R}^n, \mathbb{R}). \\ \textit{The step lengths } \alpha_k \textit{ satisfy } (2.3.7) \textit{ and } (2.3.9). \\ \textit{A Broyden family formula with } \phi \in [0,1[\textit{ is used.} \\ \mathbf{B}_0 \textit{ or } \mathbf{S}_0 \textit{ is symmetric and positive definite.} \end{cases} \tag{2.3.77}$$

Also suppose that there exist $0 < m < M < \infty$ such that

$$m\|\mathbf{y}\|^2 \le \mathbf{y}^T \mathbf{H}_f(\mathbf{x})\mathbf{y} \le M\|\mathbf{y}\|^2 \quad \textit{for all } \mathbf{y} \in \mathbb{R}^n, \mathbf{x} \in \mathcal{L}, \tag{2.3.78}$$

where the set $\mathcal{L} = \{\mathbf{x} \in \mathbb{R}^n | f(\mathbf{x}) \le f(\mathbf{x}_0)\}$ is convex and $\mathbf{x}_0$ is the starting point of the iteration. Then

$$\liminf_{k\to\infty} \|\nabla f(\mathbf{x}_k)\| = 0.$$

As for the Newton method, rigorous local convergence theory has been established for the Broyden family methods. Dennis and Moré essentially proved the following result that also states an alternative definition of superlinear convergence. Based on this result, most implementations of Newton-based methods use initial step length of unity in their line searches. In particular, this result establishes superlinear convergence of the Newton method or any method with iteration steps converging to the Newton iteration (2.3.54).

Theorem 2.3.79. [DM74, Theorem 2.2 and Corollary 2.3] *Let $f : \mathbb{R}^n \to \mathbb{R}$, $f \in C^2(\mathbb{R}^n, \mathbb{R})$. Consider an iteration of the form $\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{S}_k \nabla f(\mathbf{x}_k)$ with nonsingular matrices $\mathbf{S}_k$. Suppose that the iteration converges to $\mathbf{x}^*$ such that $\mathbf{H}_f(\mathbf{x}^*)$ is nonsingular. Then the iteration converges superlinearly to $\mathbf{x}^*$ such that $\nabla f(\mathbf{x}^*) = \mathbf{0}$ if and only if*

$$\lim_{k\to\infty} \frac{\|[\mathbf{S}_k^{-1} - \mathbf{H}_f(\mathbf{x}^*)](\mathbf{x}_{k+1} - \mathbf{x}_k)\|}{\|\mathbf{x}_{k+1} - \mathbf{x}_k\|} = 0 \tag{2.3.80}$$

and

$$\lim_{k\to\infty} \alpha_k = 1.$$

Remark 2.3.81. It can also be shown that if (2.3.80) holds and $\mathbf{H}_f(\mathbf{x}^*)$ is positive definite, there exists k_0 such that $\alpha_k = 1$ satisfies conditions (2.3.7) and (2.3.9) for all $k \ge k_0$ [DM77, Theorem 6.4].

Based on Theorem 2.3.79, Byrd, Nocedal and Yuan also proved the following local superlinear convergence result for the Broyden family methods on strictly convex functions.

Theorem 2.3.82. [BNY87, Theorem 4.1] *Consider a sequence $(\mathbf{x}_k)$ produced by Algorithm 7 or 8. Suppose that the iteration and the objective function $f : \mathbb{R}^n \to \mathbb{R}$ satisfy the assumptions of Theorem 2.3.76. Also suppose that*

$$\begin{cases} \alpha_k = 1 \textit{ whenever it satisfies } (2.3.7) \textit{ and } (2.3.9). \\ \mathbf{H}_f \textit{ is Lipschitz-continuous in a neighbourhood } U \textit{ of } \mathbf{x}^*. \end{cases} \tag{2.3.83}$$

Then the sequence $(\mathbf{x}_k)$ converges superlinearly to $\mathbf{x}^*$ such that $\nabla f(\mathbf{x}^*) = \mathbf{0}$.

Chapter 3

Implementation

3.1 Introduction to GSL and BLAS

GSL (the GNU Scientific Library) is a general-purpose numerical software library for scientific computing written in C. It has been primarily developed on the GNU/Linux platform with gcc (the GNU C compiler), but it also supports a wide variety of other UNIX-based platforms. GSL is free software under GPL (GNU General Public License). BLAS (Basic Linear Algebra Subprograms) is a low-level library for matrix and vector computations.

3.1.1 Minimization algorithms implemented in GSL

The minimization algorithms implemented in GSL version 1.11 and their corresponding line search routines (if applicable) are listed in Table 1. [1]

GSL name	Algorithm	Line search
steepest_descent	steepest descent	backtracking
conjugate_fr	Fletcher-Reeves conjugate gradient	Brent
conjugate_pr	Polak-Ribière conjugate gradient	Brent
vector_bfgs	quasi-Newton L-BFGS	Brent
vector_bfgs2	quasi-Newton L-BFGS	Fletcher*
nmsimplex	Nelder and Mead simplex	-

Table 1: Minimization algorithms implemented in GSL.

Line search algorithms

GSL implements the Brent algorithm [Bre73, p. 79-80] which is also described in [PTVF07, p. 496-499]. This algorithm alternates between a slower *golden*

[1]The prefixes of the GSL algorithm names are omitted for brevity.

section search and a more rapidly converging, but less reliable quadratic interpolation scheme. This algorithm does not use function derivatives except for its stopping criterion, that is

$$\frac{\|\nabla f(\mathbf{x}_k + \alpha \mathbf{d}_k)^T \mathbf{d}_k\|}{\|\nabla f(\mathbf{x}_k + \alpha \mathbf{d}_k)\| \|\mathbf{d}_k\|} < \text{tol}, \tag{3.1.1}$$

where $\mathbf{d}_k$ is the search direction and `tol` is the given tolerance. This condition tests the angle between $\nabla f(\mathbf{x}_k + \alpha \mathbf{d}_k)$ and $\mathbf{d}_k$. It does not imply conditions (2.3.8) and (2.3.9), and to the knowledge of the author of this thesis, it is not sufficient for convergence in the same sense as discussed in Section 2.3.1. Unlike condition (2.3.7), the above stopping criterion neither guarantees sufficient decrease of the objective function. On the other hand, the Brent algorithm with this stopping criterion does not require that the given search direction is a descent direction in the sense of equation (2.3.3).

GSL also implements a refined version of the original Fletcher's algorithm described in [Fle80] and Appendix A.2. This algorithm uses cubic interpolation instead of quadratic. Several additional steps that guarantee the stronger condition (2.3.8) instead of (2.3.9) are also incorporated into it. For this algorithm, the user-specified tolerance `tol` corresponds to the parameter η of condition (2.3.8).

The steepest descent and conjugate gradient algorithms

GSL implements the steepest descent algorithm described in Section 2.3.3. This implementation uses the backtracking line search which is described in Appendix A.1. The user-specified tolerance `tol` corresponds to the parameter σ. For the other parameter ρ, GSL uses the value $\rho = 2$. Both conjugate gradient algorithms implemented in GSL use the Brent line search. They also use periodic restarting every n steps to the steepest descent direction. The Polak-Ribière algorithm also uses equation (2.3.42) instead of (2.3.44).

L-BFGS (limited-memory BFGS) algorithms

Two limited-memory variants of the classical BFGS algorithm using inverse Hessian approximations are implemented in GSL. They differ from the classical BFGS algorithm in such a way that only a single vector is stored instead of the full inverse Hessian approximation. This substantially reduces their storage requirements. These algorithms are similar to Wright's L-BFGS algorithm [Wri94] which is also described in [Kel99, p. 79-80]. The older implementation `vector_bfgs` uses the Brent line search. The newer implementation `vector_bfgs2` that uses the modified Fletcher's line search algorithm, was introduced in GSL version 1.9. The older implementation uses periodic restarting to the steepest descent direction every n steps, whereas the newer one does not.

The Nelder and Mead simplex algorithm

The GSL implementation of the Nelder and Mead algorithm is otherwise similar to the one given in Algorithm 1, but it omits the outside contraction step. Its stopping criterion tests the simplex size that is defined as

$$s(S) = \frac{1}{n+1} \sum_{i=1}^{n+1} \|\mathbf{x}_i - \bar{\mathbf{x}}\|, \tag{3.1.2}$$

where

$$\bar{\mathbf{x}} = \frac{1}{n+1} \sum_{i=1}^{n+1} \mathbf{x}_i$$

and S is a simplex in $\mathbb{R}^n$.

3.1.2 The GSL minimization interface

The GSL minimization algorithms are divided into two categories: direct-search and gradient-based algorithms, whose names begin with the prefixes

```
gsl_multimin_fminimizer
gsl_multimin_fdfminimizer,
```

respectively.

The objective function $f : \mathbb{R}^n \to \mathbb{R}$ is supplied to a GSL minimization algorithm via a *structure* [KR88, p. 127-149] of either of the types

```
gsl_multimin_f
gsl_multimin_fdf
```

that describe C^0- and C^1-functions, respectively. Both of these structures contain a *function pointer* [KR88, p. 118-121]

```
double (*f)(const gsl_vector *x, void *params)
```

to a function that returns $f(\mathbf{x})$. Additional parameters to the objective function can be specified with the pointer `params`. In addition, pointers to functions that evaluate the gradient $\nabla f(\mathbf{x})$ or both $f(\mathbf{x})$ and its gradient can be given via the function pointers

```
void (*df)(const gsl_vector *x, void *params, gsl_vector *g)
void (*fdf)(const gsl_vector *x, void *params, double *f,
            gsl_vector *g)
```

defined in `gsl_multimin_fdf`. The evaluated function value and gradient are stored to the pointers specified with `f` and `g`, respectively.

The usage of a GSL minimization algorithm consists of the following steps:

1. Initialize the algorithm.
2. Take one iteration step of the algorithm.
3. Test the stopping criterion. Return to 2. if it is not satisfied.

Depending on the type of the used algorithm, the initialization of a GSL minimization algorithm `s` with the given objective function `f` (or `fdf`) and the given starting point `x` is done by calling either of the following methods:

```
gsl_multimin_fminimizer_set(gsl_multimin_fminimizer *s,
                             gsl_multimin_function *f,
                             const gsl_vector *x,
                             const gsl_vector *step_size)
gsl_multimin_fdfminimizer_set(gsl_multimin_fdfminimizer *s,
                               gsl_multimin_function_fdf *fdf,
                               const gsl_vector *x,
                               double step_size,
                               double tol).
```

Each gradient-based algorithm has two adjustable parameters: `step_size` and `tol`. The line search parameter `step_size` is the initial step length, and `tol` is for adjusting accuracy of line searches. The direct-search algorithms, e.g. the simplex algorithm, have one parameter: `step_size` that specifies a vector defined by equation (2.2.13).

The functions for iterating the GSL minimization algorithms and obtaining information of the iteration are given in Table 2. Depending on the type of the algorithm, each function takes a pointer to a `gsl_multimin_fminimizer` or a `gsl_multimin_fdfminimizer` object.

Function	Purpose	Applicable to
x	return the current iterate $\mathbf{x}_k$	both
minimum	return $f(\mathbf{x}_k)$	both
gradient	return $\nabla f(\mathbf{x}_k)$	fdfminimizer
iterate	take one iteration step	both
restart	reset to the steepest descent direction	fdfminimizer
size	return the simplex size	fminimizer

Table 2: GSL functions for controlling and monitoring the iteration of minimization algorithms.

GSL implements the following stopping criteria for testing the given quantity against the given tolerance `epsabs`:

```
gsl_multimin_test_gradient (const gsl_vector *g, double epsabs)
gsl_multimin_test_size (const double size, double epsabs).
```

These are applicable to gradient-based and direct search algorithms, respectively. The first one tests the gradient norm $\|\nabla f(\mathbf{x}_k)\|$, and the second one tests the simplex size defined by equation (3.1.2).

3.1.3 GSL linear algebra routines

Basic data structures for matrices and vectors are implemented in GSL with names `gsl_matrix` and `gsl_vector`, respectively. By default, GSL provides a high-level interface for linear algebra operations written in C. A C interface for more optimized low-level BLAS operations is offered as an alternative.

Several matrix decomposition methods and their associated linear equation solvers are implemented in GSL. In particular, the Cholesky decomposition method and the Cholesky solver

```
gsl_linalg_cholesky_decomp
gsl_linalg_cholesky_solve
```

are utilized by the Newton-based minimization algorithms implemented in GSL++.

3.1.4 Overview of BLAS

BLAS is an interface for low-level linear algebra operations. It provides a set of highly-optimized routines. The functionality of BLAS is divided into three levels. Brief descriptions of these levels are given in Table 3.

	Typical operations	Complexity
Level 1	vector-vector operations	$\mathcal{O}(n)$
Level 2	matrix-vector operations	$\mathcal{O}(n^2)$
Level 3	matrix-matrix operations	$\mathcal{O}(n^3)$

Table 3: Operation levels defined in the BLAS interface.

In all levels, several operations are combined in one function call, which allows higher computational efficiency. In addition, separate functions with additional optimizations are implemented for symmetric and triangular matrices in levels 2 and 3. The emphasis of BLAS is on basic arithmetic matrix and vector operations, and it does not implement more advanced operations such as matrix decompositions.

3.2 Overview of GSL++

GSL++ is a collection of algorithms, interfaces and scripts written on top of GSL by the author of this thesis. It extends GSL by providing additional minimization algorithms. These are based on the descriptions given in Chapter 2. In addition, GSL++ provides driver routines for calling GSL-based minimization algorithms from GNU Octave and an interface for evaluating symbolic function expressions. The interaction of GSL++ with the user and external libraries is shown in Figure 6.

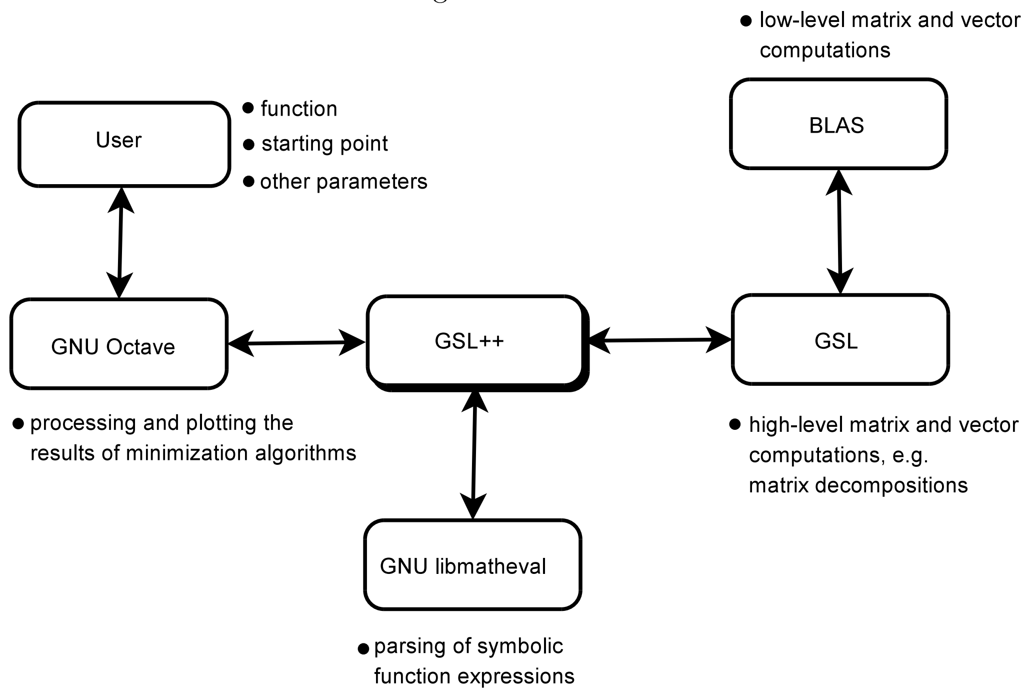

● low-level matrix and vector computations
BLAS
User
● function
● starting point
● other parameters
GNU Octave
GSL++
GSL
● processing and plotting the results of minimization algorithms
● high-level matrix and vector computations, e.g. matrix decompositions
GNU libmatheval
● parsing of symbolic function expressions

Figure 6: The interaction of GSL++ with the user and external libraries.

GNU Octave has builtin support for calling *dynamically linked* functions written in C/C++ [Eat02, Appendix A]. Via this interface, GSL++ implements the driver routines that allow calling any GSL-based minimization routine from GNU Octave by using the same calling syntax as functions written in the GNU Octave language. Supplying the input parameters, processing and plotting the results is done in the GNU Octave environment. A set of m-files was written in the GNU Octave language for this purpose. The actual computation is done in the C routines that implement the GSL minimization algorithm interfaces described in Section 3.1.2. In addition, GSL++ extends GSL by implementing support for functions with second-order derivatives and the finite-difference approximations described in Appendix A.5.

GSL++ employs the functionality of three external libraries: GSL, BLAS and GNU libmatheval. BLAS, which is used via its GSL interface, is for low-level matrix and vector computations. GSL provides higher-level operations such as matrix decompositions. It also provides the data structures for matrix and vector computations. GNU libmatheval is used for evaluating functions and derivatives from symbolic expressions. [2]

3.2.1 Proposed algorithms

In this section we discuss the revised versions of the existing GSL minimization algorithms implemented in GSL++. These algorithms are based on the descriptions given in Chapter 2. We summarize the practical importance of those theoretical results and discuss how they are taken into account in the implementations. All the algorithms described in this section are compliant with the GSL minimization interface described in Section 3.1.2.

The implemented algorithms with their associated line search algorithms and initial step selection methods are given in Table 4. These choices are based on theoretical considerations. The experimental results given in Chapter 4 provide some additional justification for these choices.

Name	Method	Line search method	Initial step
conjgrad_fr_mt	Fletcher-Reeves conjugate gradient	Moré and Thuente	Fletcher
conjgrad_pr_mt	Polak-Ribière conjugate gradient	Moré and Thuente*	Fletcher
mnewton	Newton/ modified Cholesky	Moré and Thuente	unity
bfgs_f	quasi-Newton BFGS/ inverse Hessian update	Fletcher	unity
bfgs_hess_f	quasi-Newton BFGS/ Hessian update	Fletcher	unity
bfgs_mt	quasi-Newton BFGS/ inverse Hessian update	Moré and Thuente	unity
bfgs_hess_mt	quasi-Newton BFGS/ Hessian update	Moré and Thuente	unity
lrwwsimplex	Nelder and Mead simplex	-	-

Table 4: Minimization algorithms implemented in GSL++.

Conjugate gradient algorithms

The GSL++ conjugate gradient implementations are based on Algorithms 5 and 6. Theorems 2.3.45 and 2.3.50 suggest using accurate line searches sat-

[2]Symbolic differentiation algorithms are extensively covered in [GC91].

isfying the strong Wolfe conditions. Hence, the Moré and Thuente algorithm is suggested as the default line search algorithm for these implementations. These implementations also default to Fletcher's initial step length selection described in Appendix A.3. The suggested line search parameters are

$$\mu = 0.001, \quad \eta = 0.05, \quad \chi = 0.01,$$

where the choice of χ was used by Gilbert and Nocedal [GN92, p. 21].

In particular, using the Moré and Thuente line search with the Polak-Ribière algorithm is motivated by Theorem 2.3.23 which gives the conditions for convergence to an α^* such that $\phi'(\alpha^*) = 0$. As suggested by Remarks 2.3.49, condition (2.3.48) is required as an additional stopping criterion in the line search of the GSL++ Polak-Ribière algorithm.

The modified Newton algorithm

GSL++ implements a modification of the Newton method (2.3.55) with Cholesky decompositions. This implementation guarantees positive definiteness of the Hessian by adding a multiple of identity to it, that is

$$\tilde{\mathbf{H}} = \mathbf{H} + \mu\mathbf{I},$$

where $\mu = 0$ if $\mathbf{H}$ is already positive definite [DS83, A5.5.1 and A5.5.2]. The first bound μ_1 is obtained by modifying the diagonal elements of the $\mathbf{LDL}^T$ factorization of $\mathbf{H}$ such that

$$\mathbf{L}\tilde{\mathbf{D}}\mathbf{L}^T = \mathbf{H} + \mathbf{E}$$

and choosing $\mu_1 = \max_{i=1\ldots n} \mathbf{E}_{ii}$. The used procedure for computing the diagonal matrix $\mathbf{E}$ is described in detail in Appendix A.4. The second bound μ_2 is obtained from the *Gerschgorin theorem* [DS83, p. 60,103]. Thus, the final choice of μ is given by

$$\mu = \min\{\mu_1, \mu_2\}.$$

Because of the need to evaluate and invert the Hessian, this algorithm has high computational complexity per iteration. This is however compensated by its rapid convergence rate. In order to further accelerate its convergence with accurate line searches, the Moré and Thuente line search is suggested as its default line search algorithm. As suggested by Theorem 2.3.79, initial step length of unity is used as the default choice.

Quasi-Newton BFGS algorithms

The GSL++ BFGS algorithm implementations are based on Algorithms 7 and 8. The default update formula used in the GSL++ BFGS implementation is the inverse Hessian BFGS update. Both implementations default to identity matrix as their initial approximation.

The convergence results stated in Theorems 2.3.76 and 2.3.82 require only the weaker Wolfer conditions. Fletcher's numerical results [Fle80, Table 3.5.1] also suggest that relaxing the strong Wolfe conditions does not cause a significant degradation in convergence rates. Hence, BFGS algorithms with Fletcher's line search described in Appendix A.2 were also implemented.

The suggested line search parameters for the GSL++ BFGS implementations with the Moré and Thuente line search algorithm are

$$\mu = 0.001, \quad \eta = 0.1.$$

Fletcher used the default values

$$\mu = 0.01, \quad \eta = 0.1, \quad \tau = 0.05, \quad \chi = 9$$

with his experiments on quasi-Newton algorithms. These are also the default parameters for the GSL++ BFGS algorithms with Fletcher's line search. As for the Newton algorithm, the default initial step length is unity.

All implemented BFGS algorithms include an additional safeguarding procedure for the case $\mathbf{p}_k^T \mathbf{q}_k \leq 0$. If this occurs, the matrix $\mathbf{B}_k$ or $\mathbf{S}_k$ is reset to identity matrix. This is also done in the algorithm with Hessian update if the Cholesky decomposition $\mathbf{B}_k = \mathbf{L}\mathbf{L}^T$ fails due to indefinite matrix $\mathbf{B}_k$. Both of these cases can occur due to numerical errors.

The revised simplex algorithm

The GSL++ implementation of the Nelder and Mead simplex algorithm follows Algorithm 1. It uses insertion sort [Cor01, p. 17] for sorting the simplex vertices. This algorithm has complexity of $\mathcal{O}(n^2)$ for arbitrarily sorted sets. For "almost-ordered" vertices, which is the most common case in the simplex algorithm, its complexity reduces to $\mathcal{O}(n)$ [Cor01, p. 24-25]. The insertion sort implementation is also consistent with the tie-breaking rules described in [LRWW98, p. 5-6]. The parameters of the trial point selection, as given by Lagarias et. al. [LRWW98], are

$$\rho = 1, \quad \chi = 2, \quad \gamma = \tfrac{1}{2}, \quad \text{and} \quad \sigma = \tfrac{1}{2}.$$

The initial simplex is chosen according to (2.2.13), and side lengths of unity is the suggested default choice.

3.2.2 GNU Octave utilities implemented in GSL++

GSL++ implements driver routines and several utility methods for invoking minimization algorithms from GNU Octave. As both GNU Octave and GSL++ are written in C++, they utilize its more advanced data structures and stream handling implemented in STL (the standard template library)

[Str00, p. 427-657]. We give only a brief overview of these routines. A complete Texinfo-based documentation is provided with their source code.

The driver routine `GSLpp_minimize` is for invoking GSL-based minimization algorithms from GNU Octave. It is used from the GNU Octave command prompt by typing a command of the following form:

```
octave:1> results = GSLpp_minimize(a, ap, lsp, f, fp, ep, sc,
                    scp, sp, v, t)
```

Input arguments

The input arguments for `GSLpp_minimize` are listed in Table 5. The mandatory arguments are printed in boldface. For the specifications of argument types, the reader is referred to the GNU Octave manual [Eat02, chap. 3-7].

Argument	Type	Description
a	string	minimization algorithm name
ap	struct	minimization algorithm parameters
lsp	struct	line search parameters (if applicable)
f	string	symbolic function expression or function name
fp	struct	additional objective function parameters
ep	struct	function evaluation type default: symbolic function and derivative evaluation
sc	string	stopping criterion name
scp	struct	stopping criterion parameters, e.g. threshold value
sp	vector	starting point
v	integer	verbosity level (the amount of output information)
t	boolean	measure the used computation time

Table 5: The input arguments of the GSL++ driver routine in the GNU Octave environment.

The names of the available minimization algorithms are those given in Tables 1 and 4. Most arguments are specified as *structs*. This data type is a collection that associates variable names with their values [Eat02, chap. 6]. For the GSL++ algorithms, each optional algorithm parameter is given the default value specified in Section 3.2.1 if not explicitly given. GSL algorithms invoked via the driver routine default to `step_size`$=(1,\dots,1)$ and `tol`$=0.1$.

Supplying the objective function

The objective function is supplied to the driver routine as a symbolic expression or as the name of a predefined test function. A symbolic expression may contain basic arithmetic operations $(+,-,*,/,\hat{})$ and basic elementary functions (absolute value, exponent, logarithm, square root, trigonometric functions). The predefined test functions are from the unconstrained optimization problem set given in [MGH81, p. 30]. They are programmed in the GSL++ test suite in two different ways. The first way is to generate the symbolic expression of the test function and parse it with GNU libmatheval. In addition, the evaluators of these functions are also written directly in C and linked to the GSL++ library, which mitigates the performance penalty of symbolic expression parsing. However, the lack of symbolic expressions in this case forces using finite-difference derivatives. The allowed combinations of function and derivative evaluation methods are listed in Table 6.

Function	Gradient	Hessian
symbolic	symbolic	symbolic
symbolic	symbolic	finite-difference
symbolic	finite-difference	finite-difference
precompiled	finite-difference	finite-difference

Table 6: The allowed combinations of function, gradient and Hessian evaluation methods.

Output

The GSL++ driver routine returns a variable of type `struct`. Its most relevant fields are specified in Table 7.

Field	Type	Description
converged	boolean	true if iteration converged, false if not
x	vector	estimated function minimizer
f	scalar	function value at x
g	vector	function gradient at x
H	matrix	the Hessian matrix of f at x
nfeval	scalar	the number of used function evaluations
ngeval	scalar	the number of used gradient evaluations
nHeval	scalar	the number of used Hessian evaluations
iterations	cell	a list of structs that contains information about each iteration
termval	scalar	the criterion value that is tested against the given threshold when the iteration terminates
time	scalar	the measured computation time

Table 7: The output produced by the GSL++ driver routine in the GNU Octave environment.

The following example invokes the `bfgs_f` algorithm with its default settings to find the minimizer of the Beale function starting from the point $(1,1)$. Setting the fields `f` and `g` to `sym` specifies symbolic derivatives to be used with symbolically parsed objective function. The stopping criterion is to test the condition $\|\nabla f(\mathbf{x}_k)\| < 10^{-8}$.

```
octave:1> results = GSLpp_minimize("bfgs_f", struct(), struct(),
                    "beale", struct("f", "sym", "g", "sym"),
                    struct(), "gradient", struct("eps", 1e-8),
                    [1,1]', 2, false)
```

When supplied with the above input arguments, the driver routine also prints into the Octave terminal the following output by using the C++ output streams [Str00, p. 605-656]. The progress of the iteration, the final estimates of the minimum point, minimum value and minimum gradient are printed. In addition, the driver routine keeps track of the evaluation counts of the objective function and its derivatives and prints them. [3]

```
k    x              y              f(x,y)
0    1              1              14.203
1    1              -0.3875        4.428
2    1.6222         -0.51691       2.6557
3    1.9691         0.31222        0.75726
4    2.1691         0.16779        0.33057
5    2.6852         0.42347        0.025106
6    2.8227         0.44087        0.0086544
7    2.9556         0.49583        0.0014262
8    2.9607         0.49044        0.00026025
9    2.9986         0.4994         1.68e-06
10   3              0.50002        2.5537e-09
11   3              0.5            1.4938e-12
12   3              0.5            9.4057e-15
13   3              0.5            3.4254e-23

minimizer:             [3, 0.5]
minimum value:         3.4254e-23
minimum gradient:      [1.089e-11, -5.1493e-11]

iterations:             14
function evaluations:   24
gradient evaluations:   20 (40)
```

[3]The number in parentheses is the total number of partial derivative evaluations: nm for symbolic gradients and $\frac{n(n+1)}{2}m$ for symbolic Hessians, where n is the problem dimension and m is the number of gradient or Hessian evaluations, respectively.

Several functions for illustrating the results of the iteration, e.g. the iteration path, are implemented in GSL++. `GSLpp_plot_results` can be used for plotting the iteration path of a minimization algorithm on the contour lines of the objective function in the two-dimensional case. It is invoked by typing a command of the following form:

```
octave:2> GSLpp_plot_results(results, {[0.25, 4.25], [-1, 1.5]},
                                      [false, false, true])
```

where the first argument refers to a `struct` returned by a previous call of `GSLpp_minimize`. The second and third arguments specify the plotting range and the choice of logarithmic scale for each axis. This function invokes the GNU Octave functions `plot` and `contour` for plotting the iteration path and contour lines, respectively. When supplied with the results of the driver routine call in the previous page, the above function call produces the plot shown in Figure 7.

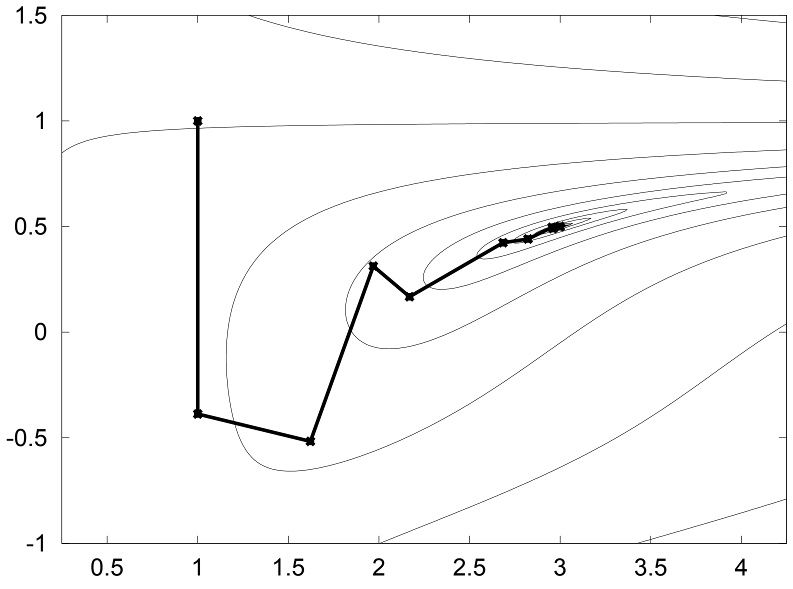
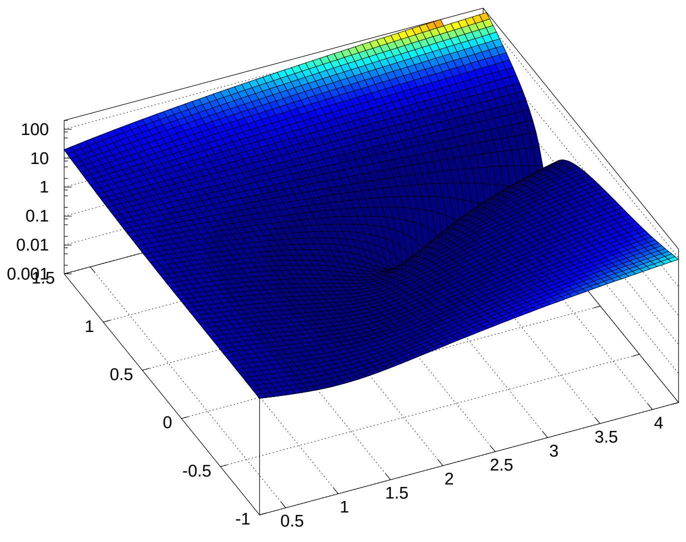


Figure 7: Iteration path of the `bfgs_f` algorithm on the Beale function.

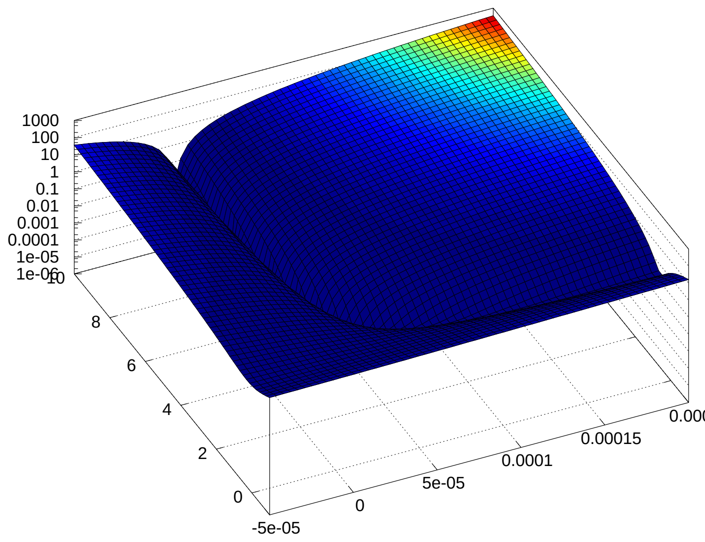
GSL++ also supports plotting 3D surface plots of two-dimensional functions with the same syntax as above. For example, the 3D surface plot of the Beale function can be plotted as follows:

```
octave:3> GSLpp_plot_mesh("beale", {[0.25, 4.25], [-1, 1.5],
                                   [1e-3, 500]}, [false, false, true])
```

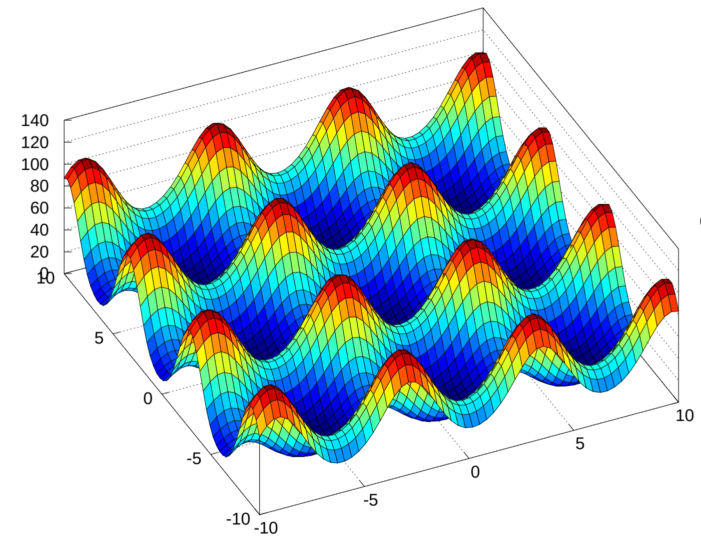
This command evaluates the values of the given test function in a uniform rectangular grid and plots the function by invoking the GNU Octave surface plotting function `surf`. Surface plots of the Beale function and some of the other test functions are shown in Figure 8.



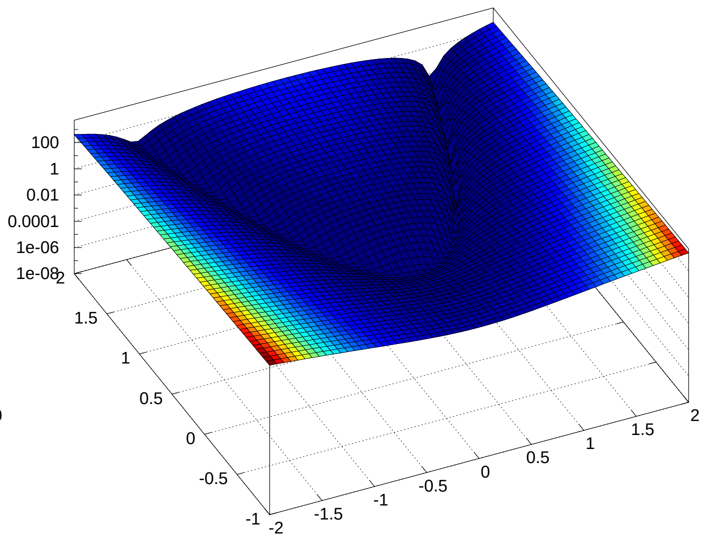
(a) Beale



(b) Powell badly scaled



(c) trigonometric ($n = 2$)



(d) extended Rosenbrock ($n = 2$)

Figure 8: Surface plots of predefined test functions programmed in the GSL++ test suite.

Chapter 4

Numerical results

4.1 Overview of testing procedures

In this chapter, we will provide an extensive comparison of the reviewed algorithms. We also analyze their characteristic properties in detail. The testing procedures carried out in this thesis were done on the test function set proposed by More, Garbow and Hillström [MGH81]. All functions in this set are given as least-squares problems of the form

$$f(\mathbf{x}) = \sum_{i=1}^{m} f_i(\mathbf{x})^2,$$

where $f : \mathbb{R}^n \to \mathbb{R}$. The tested algorithms are the existing GSL algorithms given in Table 1 and the proposed GSL++ algorithms given in Table 4. If not stated otherwise, the following is assumed:

- Symbolic gradients and Hessians are used.
- In the absence of symbolic expressions, precompiled test functions with central-difference gradients and forward-difference Hessians are used.
- The iterations are started from the standard starting points specified in [MGH81].
- GSL++ algorithms are run with their default parameters specified in Section 3.2.1.
- GSL algorithms are run with parameters `step_size`$=(1,\ldots,1)$ and `tol`$=0.1$.

All tests were run in 32-bit GNU/Linux environment on a computer with 2 GHz AMD processor and 1 GB system memory with double precision floating-point arithmetic (16 digits). [1]

[1]The range of representable numbers is $[2.225 \cdot 10^{-308}, 1.798 \cdot 10^{308}]$. The *machine epsilon* ϵ_m that is the smallest number yielding $1 + \epsilon_m > 1$ is approximately $2.220 \cdot 10^{-16}$.

4.2 Qualitative tests

4.2.1 The choice of starting point

Depending on the choice of starting point, the iteration can converge to different local minima. In order to demonstrate this, the `steepest_descent` and `bfgs_mt` algorithms were tested on the Himmelblau function

$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2,$$

which has four local minima:

$$\begin{cases} f(3,2) & = & 0 \\ f(-3.78,-3.28) & = & 0.0054 \\ f(-2.81,3.13) & = & 0.0085 \\ f(3.58,-1.85) & = & 0.0011. \end{cases}$$

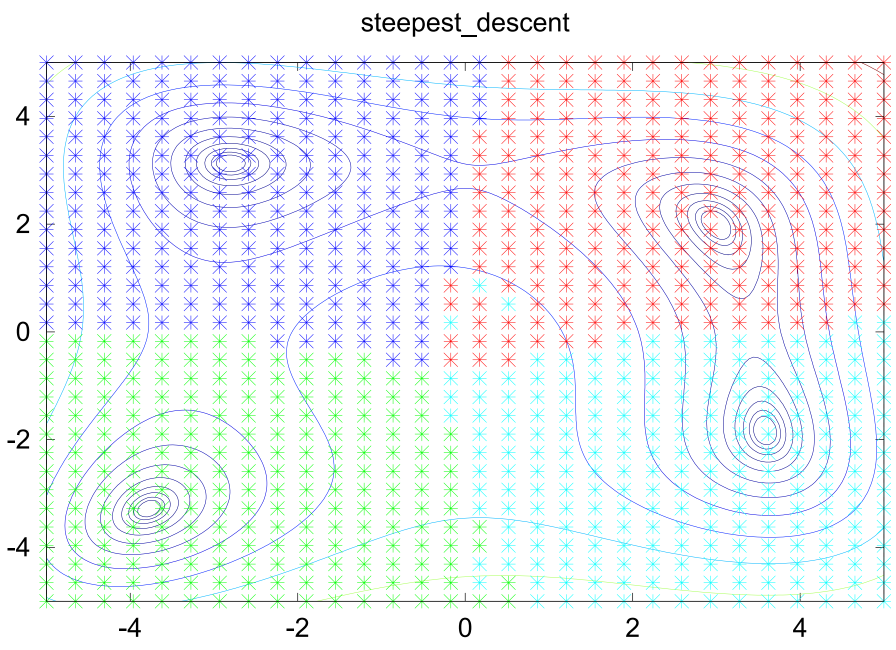
Each starting point (x_i, y_i) on the uniform rectangular grid

$$x_i = -5 + \tfrac{i-1}{29} \cdot 10 \quad y_i = -5 + \tfrac{i-1}{29} \cdot 10, \quad i = 1, \dots, 30$$

was classified according to the local minimizer the iteration started from that point converged to. Each iteration was tested against the stopping criterion $\|\nabla f(\mathbf{x}_k)\| < 10^{-8}$. These points were plotted on contour curves of the test function. The results are shown in Figure 9.

Observations

The steepest descent algorithm behaves in a very predictable manner and mostly converges to the nearest local minimizer. On the other hand, the BFGS algorithm exhibits very peculiar convergence behaviour. For a large number of starting points, it omits the nearest local minimizer and instead converges to a further one. The likely explanation is that the Wolfe (or strong Wolfe) conditions are not sufficient to distinguish different local minima. Since this test function has very variable curvature, this behaviour becomes more prominent with interpolation polynomials that rely on local function gradient information. Thus, the steepest descent algorithm with a very primitive line search routine does not exhibit this behaviour. To some extent, this can be avoided by trying different line search parameters. Of course, additional information for this such as 2D plots are not available for higher-dimensional problems, which makes adjusting the parameters even more a trial-and-error procedure.



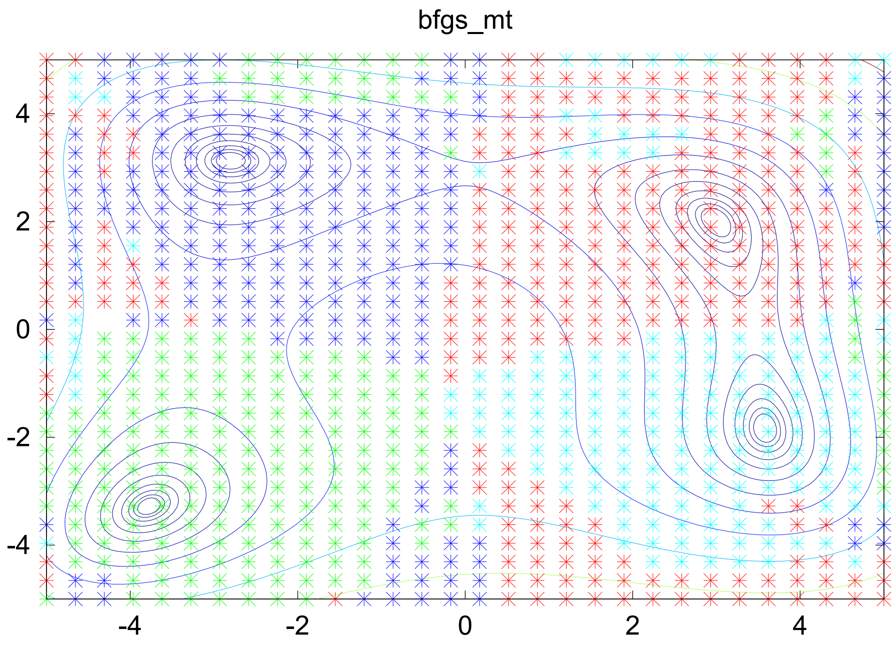


Figure 9: Convergence to different local minima from different starting points, steepest descent and BFGS algorithms.

4.3 Algorithm-specific tests

4.3.1 The Nelder and Mead simplex algorithm

The GSL++ simplex algorithm `lrwwsimplex` was tested on different test functions. Two test measures were used. The first was the condition number of the edge matrix $\mathbf{E}_k$, as defined by (2.2.2), that is

$$\kappa(\mathbf{E}_k) = \left|\frac{\lambda_{max}}{\lambda_{min}}\right|, \tag{4.3.1}$$

where λ_1 and λ_2 are the largest and smallest eigenvalues of $\mathbf{E}_k$, respectively. This quantity measures degeneracy of the simplex. If the simplex condition numbers become very large, it implies that the simplices become highly elongated. This can disable the ability of the simplex algorithm to obtain its trial steps. The second measure was the simplex volume (2.2.12). The distance $\|\mathbf{x}_k - \mathbf{x}^*\|$ to the known minimizer of each test function is also plotted in order to see the correlation of these test measures with the actual convergence of the iteration. Each iteration was tested against the stopping criterion $\|\mathbf{x}_k - \mathbf{x}^*\| < 10^{-8}$. The results are shown in Figure 10.

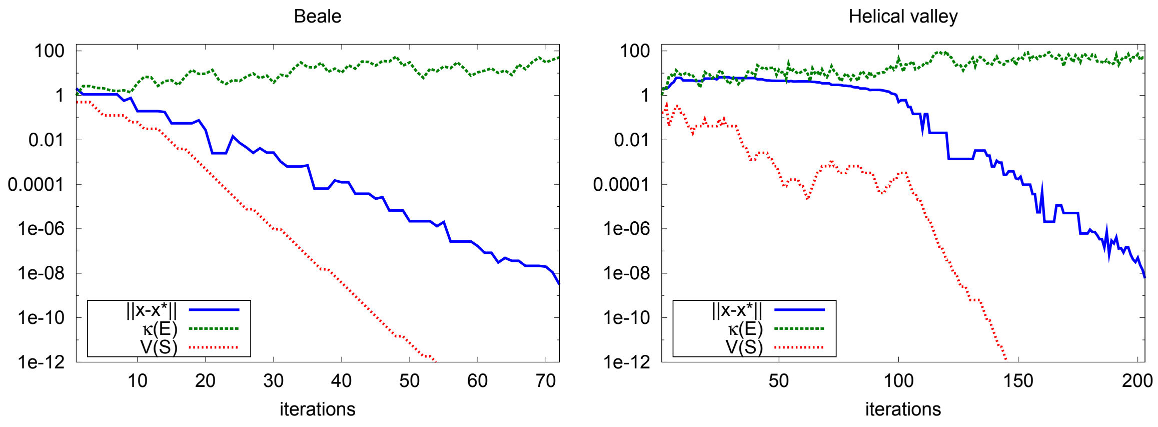


Figure 10: Convergence of the `lrwwsimplex` algorithm on the Beale function (left) and the helical valley function (right).

The `lrwwsimplex` algorithm performs well on the low-dimensional Beale and helical valley functions. The simplex condition number remains low, and the decrease in simplex volume is strongly correlated with convergence of the iteration. The higher-dimensional problems however expose a major weakness in this algorithm. Most n-dimensional test functions in the [MGH81] problem set caused this algorithm practically fail to converge. The effect of increasing problem dimension n on the simplex algorithm is shown in Tables 8- 10. If the stopping criterion was not satisfied after 50000 iterations, the iteration was terminated. The number of iterations i, the final gradient norm $\|\nabla f(\mathbf{x}_k)\|$, the final simplex volume $V(S_k)$, the final distance $\|\mathbf{x}_k - \mathbf{x}^*\|$ to the known minimizer and the final simplex condition number $cond(S_k)$ are listed in these tables.

n	i	$\|\nabla f(\mathbf{x}_k)\|$	$V(S_k)$	$\|\mathbf{x}_k - \mathbf{x}^*\|$	$cond(S_k)$
4	409	2.75075e-07	4.01235e-36	8.91199e-09	100.089
8	5255	6.44921e-07	9.19942e-73	6.99695e-09	240.863
12	31858	4.20566e-07	1.75309e-148	9.69522e-09	399.355
16	50000	4.7912	3.35222e-205	5.46229	8.16414e+13
20	50000	5.18912	0	3.87515	Inf

Table 8: Effect of problem dimension, extended Rosenbrock function.

n	i	$\|\nabla f(\mathbf{x}_k)\|$	$V(S_k)$	$\|\mathbf{x}_k - \mathbf{x}^*\|$	$cond(S_k)$
4	159	3.69402e-07	1.64346e-32	8.97569e-09	6.77109
8	951	1.48714e-07	1.47191e-71	9.90648e-09	58.8044
12	4092	2.62145e-07	5.82565e-113	9.69223e-09	243.077
16	25052	2.05804e-07	5.13727e-151	9.97749e-09	176.319
20	50000	2.90948	0	1.01255	2164.77

Table 9: Effect of problem dimension, variably dimensioned function.

n	i	$\|\nabla f(\mathbf{x}_k)\|$	$V(S_k)$	$\|\mathbf{x}_k - \mathbf{x}^*\|$	$cond(S_k)$
4	467	2.23181e-15	1.42547e-50	4.25208e-09	4.71885e+08
8	3136	7.61442e-15	1.45141e-102	7.21967e-09	1.34031e+09
12	14008	5.55417e-16	2.43291e-165	9.7844e-09	3.3855e+09
16	50000	1.03882e-15	2.01755e-240	5.24494e-08	1.00427e+10
20	50000	0.000293667	1.55445e-142	0.00756036	1.70638e+08

Table 10: Effect of problem dimension, extended Powell singular function.

The above results show a very rapid increase in the iteration counts as the problem dimension increases. Furthermore, the algorithm fails on all three test functions with $n = 20$. On the contrary, the simplex volume converges to zero in all cases. This collapsing of simplices with no observable decrease in gradient norms is associated with very high simplex condition numbers, which indicates that the simplices are practically degenerate. The fundamental reason for this is that although the simplex algorithm is guaranteed to generate nondegenerate simplices, there is no upper bound for the simplex condition number. Similar results have also been reported by Torczon [Tor89]. The observed failures basically render this algorithm unusable in higher dimensions. Similar results were also obtained with the corresponding GSL algorithm `nmsimplex`.

4.3.2 The steepest descent algorithm

The iteration path of steepest descent algorithm on the Extended Rosenbrock function ($n = 2$) demonstrates its characteristic behaviour. In order to better demonstrate this, a modification of the GSL `steepest_descent` algorithm was run with the more accurate Moré and Thuente line search ($\mu = 0.001$, $\eta = 0.01$) instead of backtracking. On this test function, the algorithm takes a large number of redundant steps, as it can be seen from Figure 11.

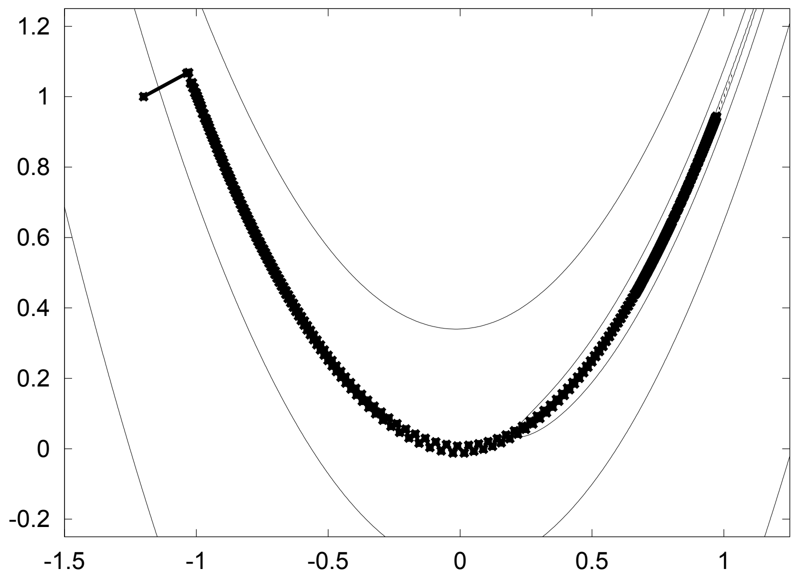


Figure 11: The iteration path of the steepest descent algorithm on the extended Rosenbrock function ($n = 2$).

This "zigzag-effect" is related to scaling of variables. In order to demonstrate this, the quadratic function

$$f(T^{-1}(\mathbf{x})) = (\alpha x_1)^2 + x_2^2, \quad T(\mathbf{x}) = (\tfrac{x_1}{\alpha}, x_2) \tag{4.3.2}$$

with the scale factor α was plotted with different values of α. The iteration was started from the point $\tilde{\mathbf{x}}_0 = T(\mathbf{x}_0) = (\frac{0.4}{\alpha}, 0.9)$. The results are shown in Figure 12. With symmetric scaling of axes, the algorithm finds the minimizer in a single step. However, even very small values of α can cause its performance to deteriorate significantly. [2]

[2]Note the different aspect ratios and scaling that does not preserve the orthogonality of negative gradient directions with contour lines.

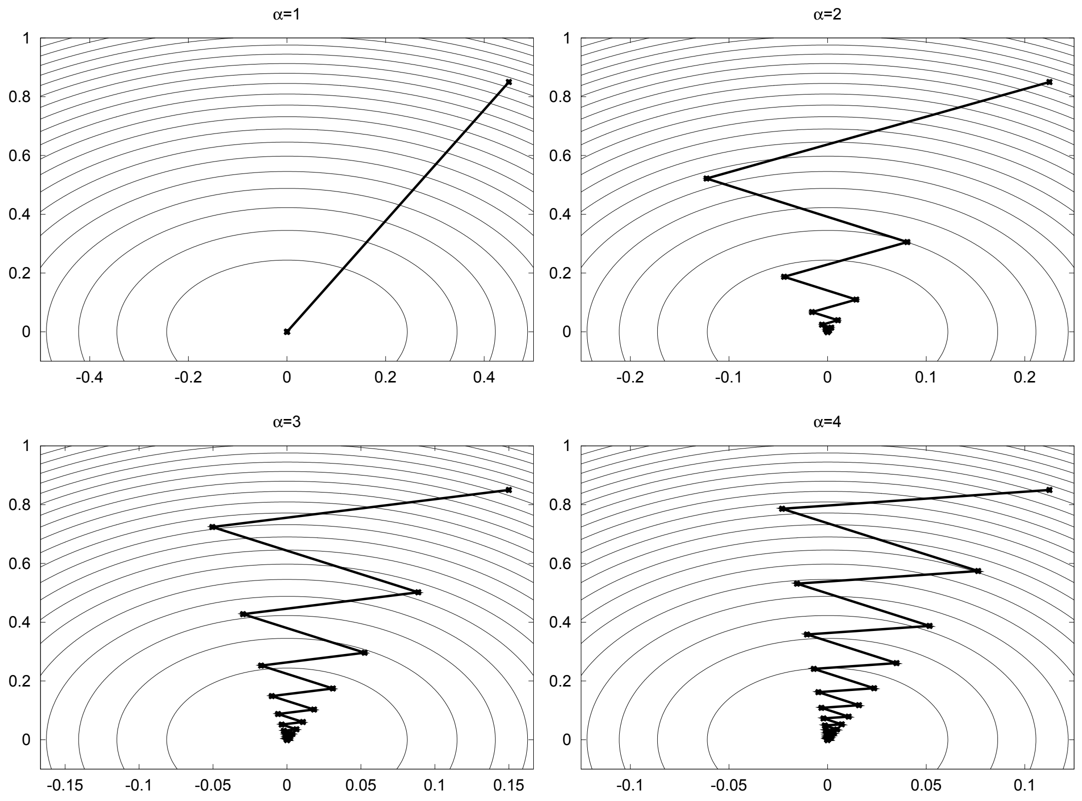


Figure 12: Iteration paths of the steepest descent algorithm on function (4.3.2) with different scale parameters.

4.3.3 Conjugate gradient algorithms

The Fletcher-Reeves and Polak-Ribière conjugate gradient algorithms can exhibit very different convergence behaviour on nonlinear functions. Perhaps the most significant difference is that the Fletcher-Reeves algorithm has a tendency to fall into a state of very slow convergence, which is rarely seen with the Polak-Ribière algorithm [GN92, p. 9,21-24]. Analyzing the effect of restarting procedures provides some insight on this aspect. The cosine of the angle between the steepest descent direction $-\nabla f(\mathbf{x}_k)$ and the current search direction $\mathbf{d}_k$, that is

$$\cos\theta_k = -\frac{\nabla f(\mathbf{x}_k)^T \mathbf{d}_k}{\|\nabla f(\mathbf{x}_k)\| \|\mathbf{d}_k\|} \tag{4.3.3}$$

was measured as a function of the number of iterations. The iteration was terminated when the stopping criterion $\|\mathbf{x}_k - \mathbf{x}^*\| < 10^{-8}$ was satisfied. The results for the extended Rosenbrock function with the `conjgrad_fr_mt` and `conjgrad_pr_mt` algorithms are shown in Figure 13. Gradient and search direction norms $\|\nabla f(\mathbf{x}_k)\|$ and $\|\mathbf{d}_k\|$ are also plotted in order to show their correlation with (4.3.3).

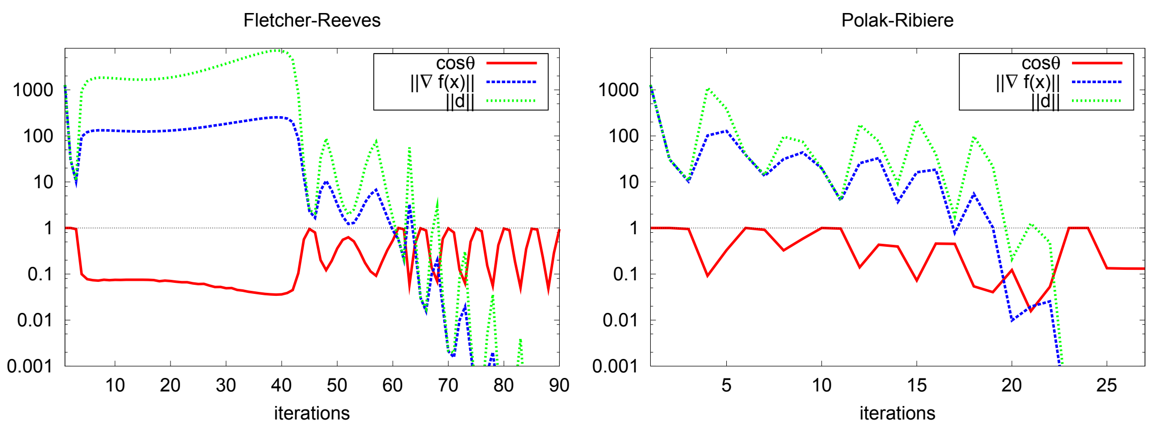


Figure 13: Convergence of the Fletcher-Reeves algorithm (left) and the Polak-Ribière algorithm (right) on the extended Rosenbrock function (n=60).

These results are very similar to those obtained by Gilbert and Nocedal [GN92]. They clearly show that the Fletcher-Reeves algorithm stagnates between 5 and 40 iterations. A possible explanation for this behaviour is based on three observations: [3]

$$\cos\theta_k < 0.1, \quad \|\mathbf{g}_{k+1}\| \approx \|\mathbf{g}_k\| \quad \text{and} \quad \|\mathbf{d}_k\| >> \|\mathbf{g}_k\|$$

starting from the 5th iteration step. Consequently, it follows from equation (2.3.41) that $\gamma_k \approx 1$. Recalling equation (2.3.40) for the nonlinear conjugate gradient method, that is

$$\mathbf{d}_{k+1} = -\mathbf{g}_{k+1} + \gamma_{k+1}\mathbf{d}_k,$$

the above observations imply that $\mathbf{d}_{k+1} \approx \mathbf{d}_k$. With the observation that $\cos\theta_k < 0.1$ ($\theta_k > 85°$), this implies that the algorithm enters a phase where it repeatedly uses search directions that are nearly orthogonal to steepest descent directions. This phase terminates only after 40 iterations. It should also be noted that periodic restarting takes place only after 60 iterations, which questions its usefulness in this particular case.

On the other hand, the restarting strategy incorporated in the Polak-Ribière algorithm seems to be much more efficient. Instead of restarting every nth iterations, equation (2.3.44) effectively forces the algorithm to restart back to the steepest descent direction whenever $\gamma_k \leq 0$. In particular, this occurs whenever $\mathbf{g}_{k+1} \approx \mathbf{g}_k$, which prevents the behaviour observed with the Fletcher-Reeves algorithm. These restarts can be seen at 6th, 10th, and 23th iteration steps, where $\cos\theta_k = 1$.

[3]This argument is originally due to Powell [Pow77].

4.3.4 Quasi-Newton BFGS algorithms

The accuracy of Hessian and inverse Hessian approximations can have a considerable effect on the convergence rate of a quasi-Newton algorithm. In order to test this, the error norms

$$\|\mathbf{H}_f(\mathbf{x}_k) - \mathbf{B}_k\|_F, \quad \|\mathbf{H}_f(\mathbf{x}_k)^{-1} - \mathbf{S}_k\|_F, \tag{4.3.4}$$

where $\|\cdot\|_F$ denotes the Frobenius norm (2.3.72) with $\mathbf{W} = \mathbf{I}$, were measured at each iteration step of the `bfgs_hess_mt` and `bfgs_mt` algorithms. In addition, the error norm of Theorem 2.3.79, that is,

$$\Delta_k = \frac{\|[\mathbf{B}_k - \mathbf{H}_f(\mathbf{x}_k)](\mathbf{x}_{k+1} - \mathbf{x}_k)\|}{\|\mathbf{x}_{k+1} - \mathbf{x}_k\|} \tag{4.3.5}$$

was also measured in order to verify superlinear convergence of these algorithms. The iteration was terminated when $\|\mathbf{x}_{k+1} - \mathbf{x}_k\| < 10^{-14}$. The results of these tests on the extended Rosenbrock function are shown in Figure 14. Distance to the minimizer $\mathbf{x}^*$ is also plotted.

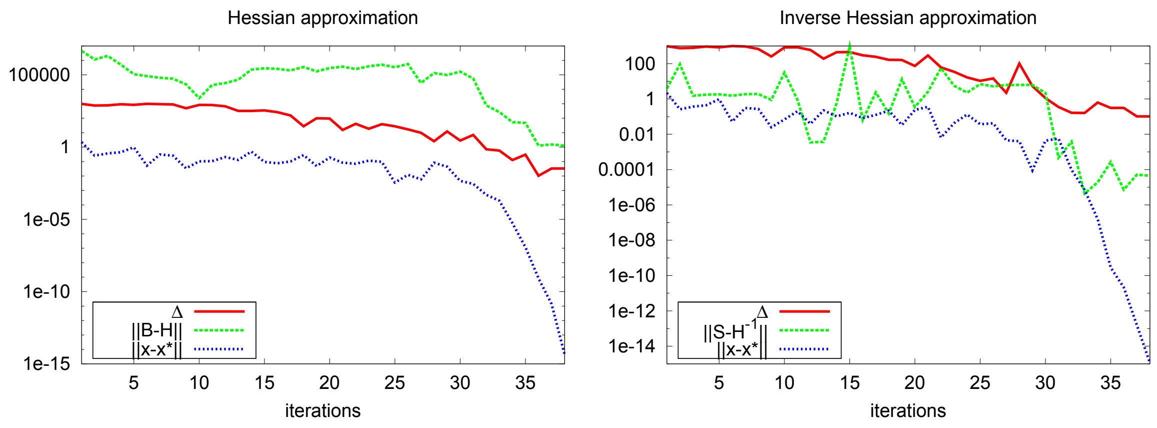


Figure 14: Convergence of BFGS algorithms and their Hessian and inverse Hessian approximations on the extended Rosenbrock function ($n = 4$).

The above results and similar behaviour on several other test problems indicate that superlinear convergence of the BFGS method cannot be established with these implementations due to limited numerical precision. The norm Δ_k stops converging to zero when $\|\mathbf{x}_k - \mathbf{x}^*\|$ becomes small. This can also be observed from the linear shape of $\|\mathbf{x}_k - \mathbf{x}^*\|$ near the minimizer, which is especially the case with the inverse Hessian update. Rather that being superlinear, the observed convergence rate is linear with ratio $\|\mathbf{x}_{k+1} - \mathbf{x}^*\| / \|\mathbf{x}_k - \mathbf{x}^*\| \approx 10^{-2}$. These results however show that the rapid convergence near the minimizer is correlated with decrease in the error norms (4.3.4). That is to say, the quasi-Newton step (2.3.68) approaches (2.3.55).

Consequently, with the choice of initial step length of unity and Remark 2.3.81, this approaches the pure Newton step (2.3.54). [4]

Test results on the extended Powell singular function are shown in Figure 15. Also the condition number $\kappa(\mathbf{H}_f(\mathbf{x}_k)) = \lambda_{max}/\lambda_{min}$, where λ_{max} and λ_{min} are the largest and smallest eigenvalues of $\mathbf{H}_f(\mathbf{x}_k)$, respectively, is plotted. This function has singular Hessian at its minimum. Consequently, the condition number that measures invertibility of the Hessian, approaches infinity as the iterates approach to the minimizer. Thus, the algorithms fail to produce usable Hessian and inverse Hessian approximations in this special case. This is associated with the observed linear convergence rates that are atypically slow for the BFGS algorithm. [5]

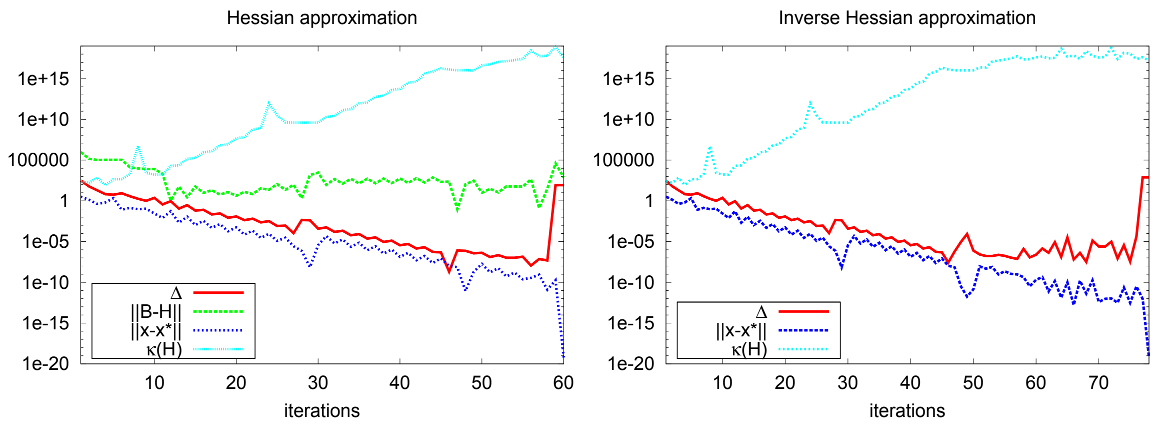


Figure 15: Convergence of BFGS algorithms and their Hessian and inverse Hessian approximations on the extended Powell singular function ($n = 4$).

It should be also noted that on the extended Powell singular function, the Hessian update with Cholesky factorizations has substantially faster convergence near the minimizer ($\|\mathbf{x}_k - \mathbf{x}^*\| < 10^{-10}$), which indicates its better numerical stability. However, the Hessian and inverse Hessian updates were observed to have practically identical convergence rates on most test problems. Because the difference between these two updating methods was observed to be only marginal except for pathological cases, it is recommended to use the computationally more efficient inverse Hessian update if at least double-precision accuracy is available. However, it has been reported that the numerically more stable Hessian update performs substantially better if the available machine precision is less than single precision (8 digits) [BCP04].

[4] Convergence to the exact Hessian is theoretically guaranteed only in the sense of equation (4.3.5).

[5] The error norm $\|\mathbf{H}_f(\mathbf{x}_k)^{-1} - \mathbf{S}_k\|_F$ was not measured with the inverse Hessian approximation because of numerical problems with computing the inverse Hessian.

4.4 Comparison of algorithms

4.4.1 Sensitivity to line search parameters

One of the key properties that define the robustness of a gradient descent algorithm is its dependency on the choice of line search parameters. In order to test this, the GSL++ BFGS and conjugate gradient algorithms were compared on the Beale, extended Rosenbrock and helical valley functions. All algorithms were run with the Moré and Thuente line search with variable accuracy, which is controlled by the parameter η. The effect of parameter μ was not tested because it was not observed to have a significant effect.

In these tests the `bfgs_mt`, `conjgrad_fr_mt` and `conjgrad_pr_mt` algorithms were compared against each other. Each iteration was tested against the stopping criterion $\|\mathbf{x}_k - \mathbf{x}^*\| < 10^{-6}$. The number of used iterations and function and gradient evaluations were also counted. The results are shown in Figure 16. [6]

Observations

From these results, it is evident that the BFGS algorithm `bfgs_mt` is very insensitive to the choice of line search parameters. This is not the case with conjugate gradient algorithms `conjgrad_fr_mt` and `conjgrad_pr_mt`. Even small changes in the value of η cause substantial differences in iteration, function and gradient evaluation counts. This is largely independent of the accuracy of the line search, and it can also occur with small values of η. Such a behaviour can make choosing proper line search parameters for these algorithms very difficult. These results however seem to slightly indicate that choosing small values of η leads to rapid convergence.

It is very interesting to note that `conjgrad_fr_mt` is particularly sensitive to the choice of parameter η. This behaviour is difficult to explain. It is possibly related to the observations made in Section 4.3.3. Some unfortunate choices of the parameter η can trigger this behaviour, which leads to substantially slower convergence.

Function and gradient evaluation counts also show that imposing too strict line search conditions can lead to degraded performance. All algorithms start to exhibit increases in the numbers of function and gradient evaluations in the range $\eta < 0.05$. This is obvious, since the line search algorithm needs more interpolation steps to produce an admissible step length with stricter conditions. Based on these results, the recommended values of η for the BFGS and conjugate gradient algorithms lie in the range $0.1 - 0.3$.

[6]The GSL++ implementation of the Moré and Thuente algorithm uses the same number of function and gradient evaluations each iteration.

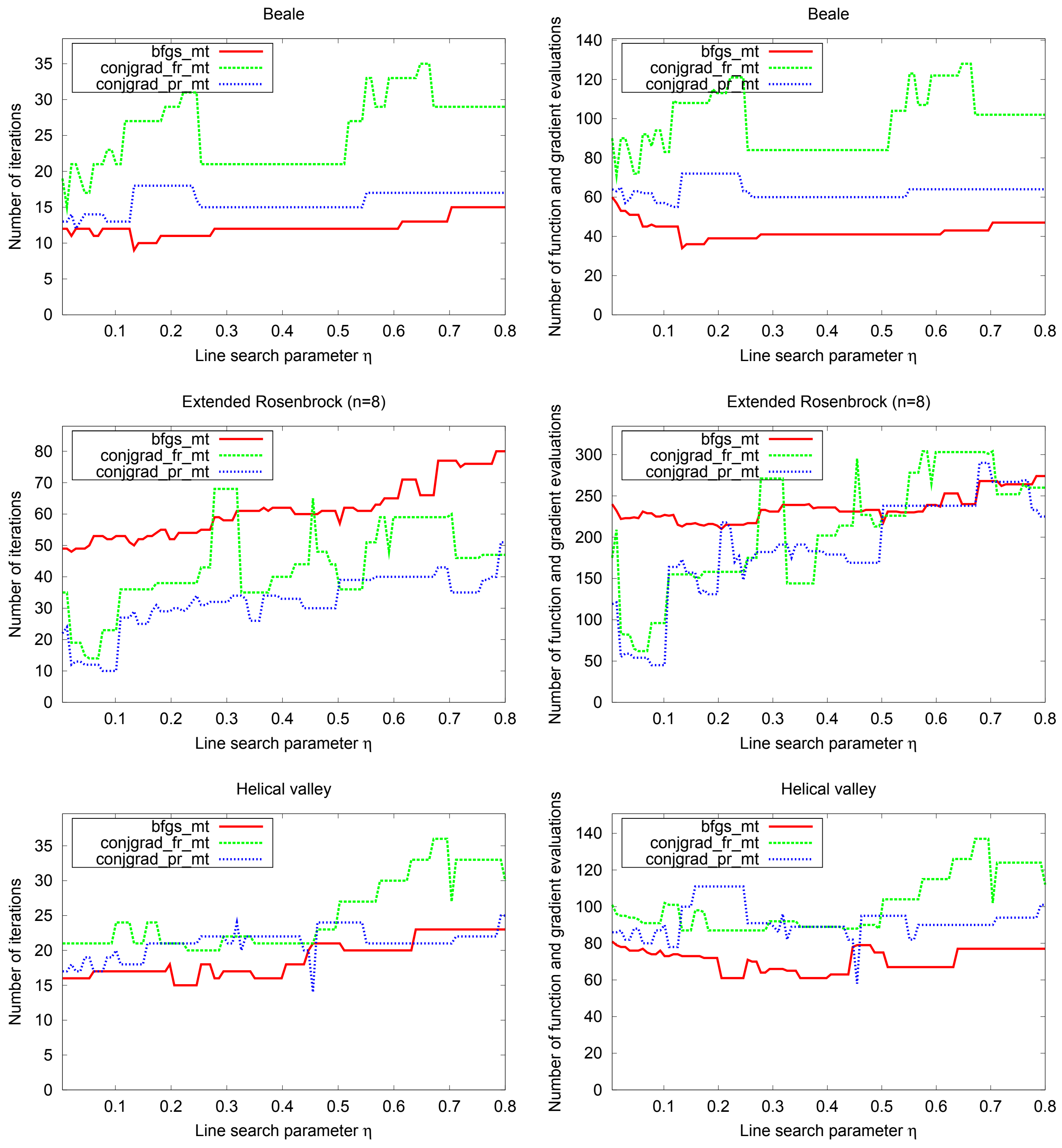


Figure 16: Iteration counts (left) and function and gradient evaluation counts (right) plotted as a function of η.

4.4.2 Convergence rates

All GSL++ and GSL minimization algorithms were tested on all test functions in the [MGH81] problem set with known minimizers. These functions are given in Table 11.

Test function	n	m
Brown badly scaled		
Beale		
helical valley		
Gulf research and development		30
Wood		
extended Rosenbrock	8	
extended Powell singular	4	
variably dimensioned	10	

Table 11: Test functions and their dimensions (if not default) used in the convergence rate tests.

Each iteration was tested against the stopping criterion $\|\mathbf{x}_k - \mathbf{x}^*\| < 10^{-12}$. The results are shown in Figures 17-20. The results on test functions on which several algorithms failed to converge, are not shown. Based on the observed convergence rates, the tested algorithms fall into four different categories:

1. Newton-based algorithms testing the Wolfe conditions
2. conjugate gradient algorithms testing the strong Wolfe conditions
3. gradient descent algorithms with the Brent line search
4. simplex algorithms

Instead of analyzing the convergence rates by verifying the conditions of Definition 2.1.12, we will give only qualitative analysis. Several convergence plots show that theoretically superlinearly or quadratically convergent algorithms fail to exhibit these properties in their strict sense and instead exhibit linear convergence near the minimizer. Due to limited numerical precision, this more rule than exception. Many of these test functions neither satisfy the conditions of these theoretical convergence results. It should also be emphasized that this is not a performance comparison, since numerous other factors contribute to the overall performance of a minimization algorithm.

An important general observation is that convergence rates of gradient descent methods are largely governed by their line search routines. Different algorithms with identical line search routines can exhibit about 50% differences in iteration counts, which in some cases can also be observed with the same algorithm but a different line search routine.

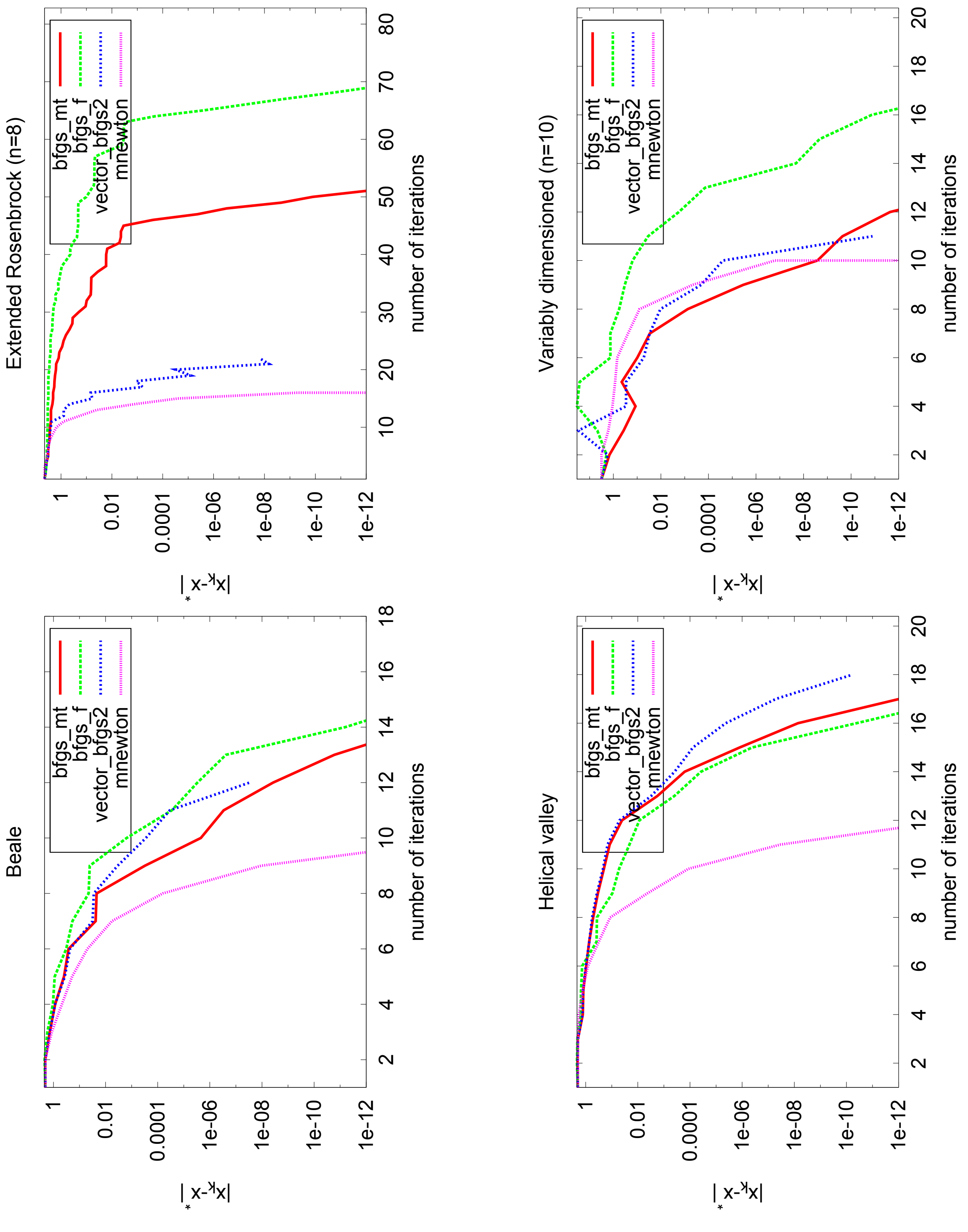


Figure 17: Convergence rates of Newton-based algorithms with line searches satisfying the Wolfe conditions.

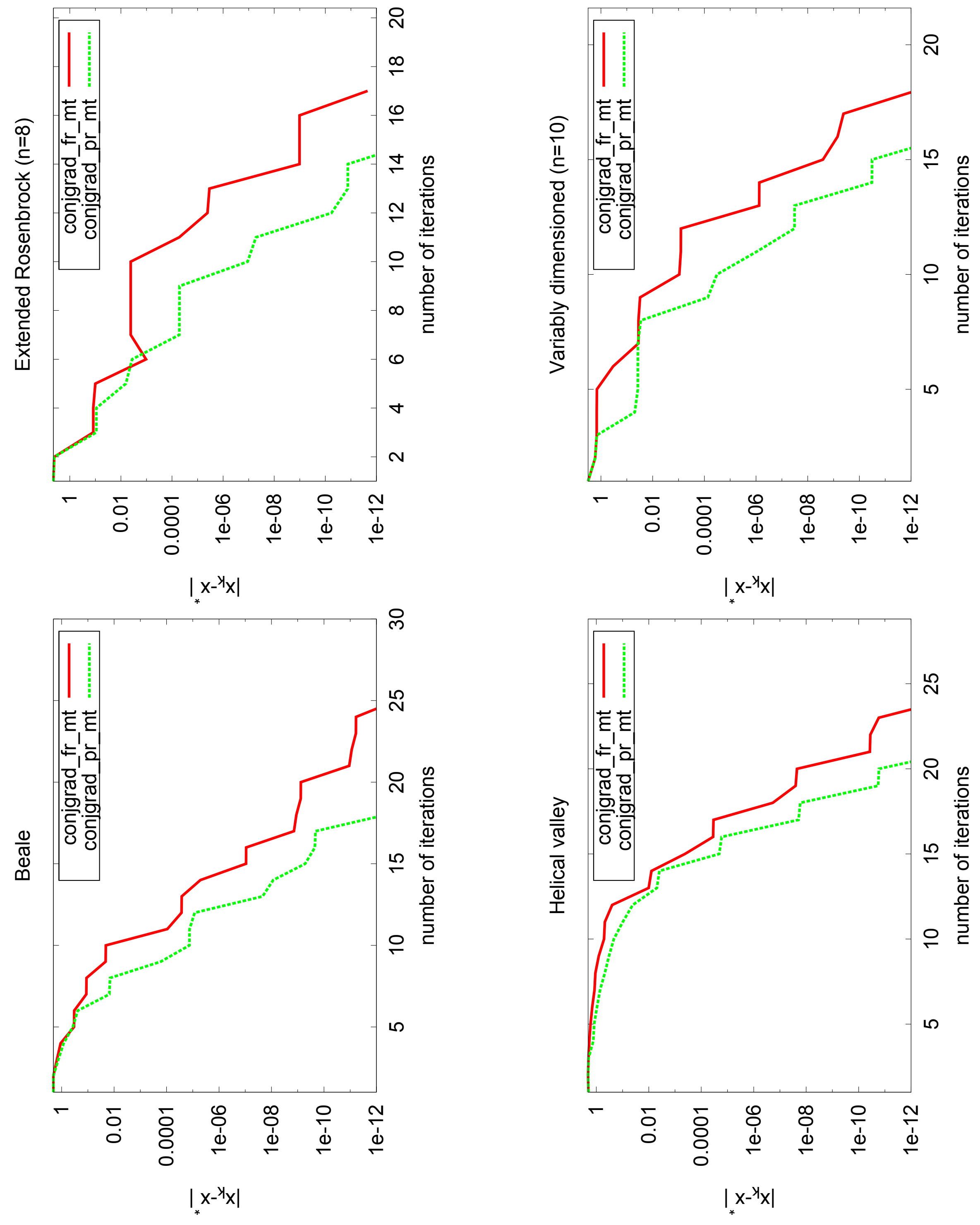


Figure 18: Convergence rates of conjugate gradient algorithms with line searches satisfying the strong Wolfe conditions.

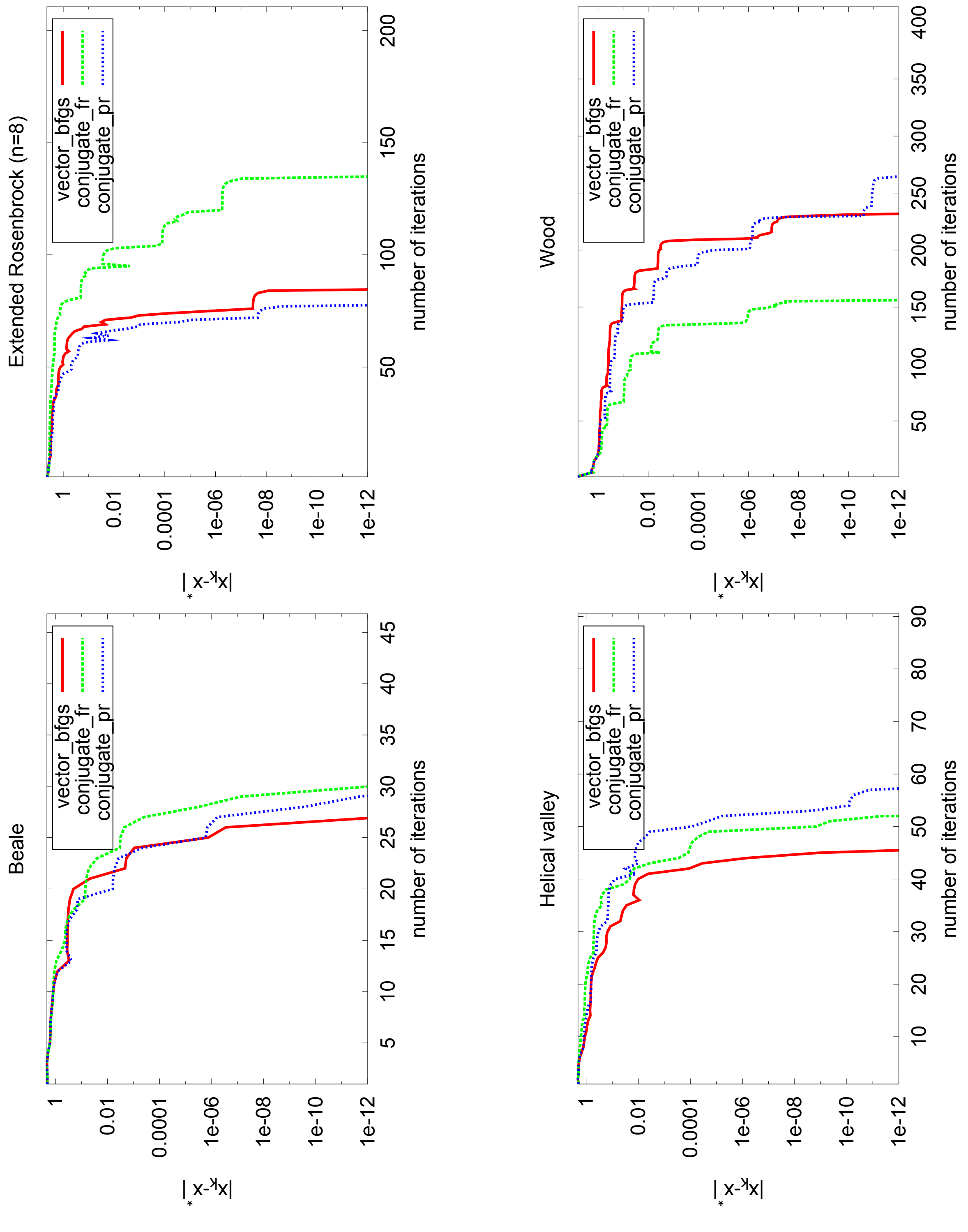


Figure 19: Convergence rates of GSL algorithms with the Brent line search.

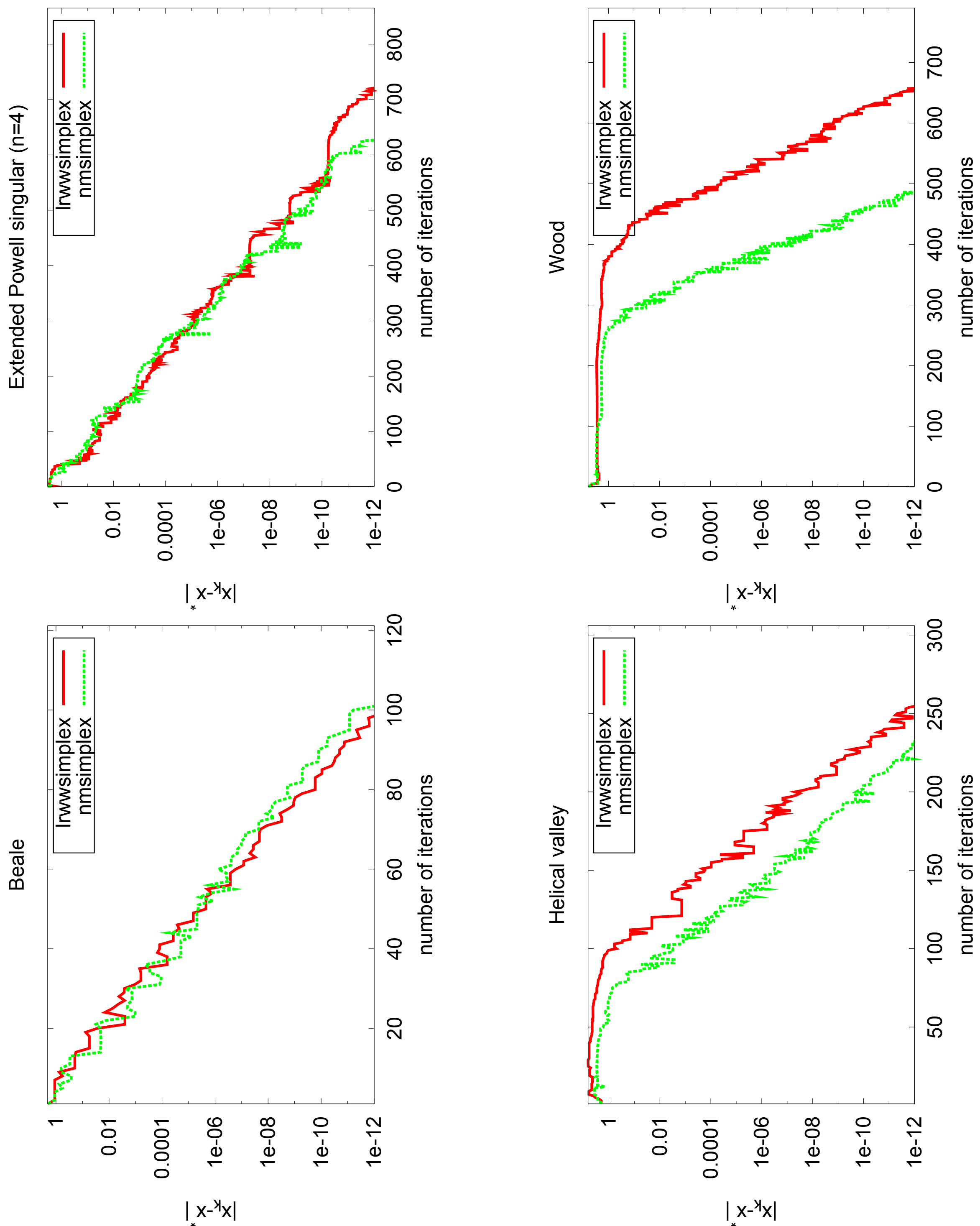


Figure 20: Convergence rates of simplex algorithms.

Observations (convergence class 1)

In general, the Newton-based algorithms exhibit superlinear or rapid linear convergence pattern. The modified Newton algorithm `mnewton` exhibits consistently the fastest convergence rates, which agrees with its theoretically established quadratic convergence rate. This is the case whenever the Hessian remains positive definite, and no modifications to its elements are necessary. The BFGS algorithms converge at a less impressive rate, and their convergence rates show signs of degradation to linear.

It is also interesting to observe that relaxing the strong Wolfe conditions has only limited effect on convergence rates of the BFGS algorithm. The differences between `bfgs_f` and `bfgs_mt` are mostly insignificant. Surpisingly, `vector_bfgs2` consistently stops converging when $\|\mathbf{x}_k - \mathbf{x}^*\|$ enters the range $10^{-12} - 10^{-7}$. These observations indicate serious problems in the line search algorithm of `vector_bfgs2`, since otherwise identical `vector_bfgs` in convergence class 3 exhibits no such behaviour.

Observations (convergence class 2)

Instead of superlinear, the convergence patterns of algorithms of this class seem to be linear, but nevertheless at a very rapid rate. The Polak-Ribière algorithm converges faster on all test functions. One can also see some kind of n-step convergence pattern, as stated in Theorem 2.3.50. This particularly applies in low dimensions to `conjgrad_fr_mt` with periodic restarting, and to some extent to `conjgrad_pr_mt` with less regular restarts.

Observations (convergence class 3)

The initial convergence rates far from the minimizer are very slow with this class of algorithms, which may indicate that condition (3.1.1) is not sufficient to guarantee global convergence as the Wolfe conditions do. However, the final convergence rates near the minimizer are very rapid. Also note that the stair-like pattern observed in several cases is due to periodic restarting implemented in these algorithms.

Observations (convergence class 4)

Only linear convergence rates are observed at best, and the rate remains constant throughout the iteration. Also the number of iterations is substantially higher than in the previous cases. It can also be observed that `lrwwsimplex` exhibits convergence rates very similar to those obtained with `nmsimplex`. Another point of interest is that the simplex algorithms solved some test problems, such as the extended Powell singular function, on which most gradient descent algorithms failed.

4.4.3 Function and gradient evaluation counts

The number of used function and gradient evaluations is a key performance measure of a minimization algorithm. Evaluation of a complicated function may have a significant contribution on its overall performance. The GSL++ and GSL algorithms were tested on the test problems given in Table 11. Only the results on the helical valley function are shown, because the algorithms behaved similarly on several other test problems.

Symbolic derivatives

The results with symbolic function expressions and derivatives are shown in Table 12. The number of iterations (i), function evaluations (f), function evaluations per iteration (f/i), gradient evaluations (g) and gradient evaluations per iteration (g/i) were measured. Each iteration was terminated when the stopping criterion $\|\mathbf{x}_k - \mathbf{x}^*\| < 10^{-8}$, where $\mathbf{x}^*$ is the known minimizer, was satisfied. [7]

Algorithm	i	f	f/i	g	g/i
bfgs_mt	16	76	4.75	76	4.75
bfgs_f	16	39	2.4375	27	1.6875
vector_bfgs2	18	74	4.11111	49	2.72222
mnewton	12	48	4	48	4
conjgrad_fr_mt	21	94	4.47619	94	4.47619
conjgrad_pr_mt	19	88	4.63158	88	4.63158
vector_bfgs	45	87	1.93333	68	1.51111
conjugate_fr	50	88	1.76	70	1.4
conjugate_pr	53	93	1.75472	73	1.37736

Table 12: Function and gradient evaluation counts, symbolic derivatives.

Algorithms with the Moré and Thuente line search use the highest number of function and gradient evaluations. On the other hand, this is to some extent compensated by the rapid convergence rates of these algorithms. These observations imply high computational cost of algorithms with this line search routine.

In terms of function and gradient evaluation counts per iteration, `bfgs_f` is surprisingly efficient. The same applies to `vector_bfgs2`. In particular, the counts of more expensive gradient evaluations are lower than function evaluation counts. These algorithms also use relatively low number of iterations, which indicates high overall efficiency of Fletcher's line search algorithm.

[7] `mnewton` also uses one Hessian evaluation per iteration.

Algorithms with the Brent line search use even lower number of function and gradient evaluations per iteration. However, iteration counts can be substantially higher. The total number of function and gradient evaluations are nevertheless similar to or lower than those obtained with algorithms using the Moré and Thuente line search. In particular, this applies to conjugate gradient algorithms, for which the Brent line search is highly efficient.

Due to its low iteration counts, `mnewton` is very competitive against the other algorithms. Evaluating the symbolic Hessian can however be computationally very expensive, which is even more the case on high-dimensional test problems.

Finite-difference derivatives

These tests were also run on the precompiled test functions with finite-difference derivatives. The central difference formula (A.5.2) and the forward difference Hessian (A.5.4) were used in this case. This time the tolerance of $\|\mathbf{x}_k - \mathbf{x}^*\|$ was set to a less strict value of 10^{-6}. The results are shown in Table 13.

Algorithm	i	f	f/i
bfgs_mt	22	672	30.5455
bfgs_f	27	345	12.7778
vector_bfgs2	73	996	13.6438
mnewton	14	522	37.2857
conjgrad_fr_mt	49	1421	29
conjgrad_pr_mt	55	1568	28.5091
vector_bfgs	109	1027	9.42202
conjugate_fr	143	1326	9.27273
conjugate_pr	123	1152	9.36585
lrwwsimplex	176	312	1.77273
nmsimplex	138	253	1.83333

Table 13: Function and gradient evaluation counts, finite-difference derivatives.

The results are similar to those discussed in the previous section. However, comparison of these results shows that using finite-difference approximations can seriously degrade convergence rates. `mnewton` also uses a very high number of function evaluations per iteration due to additional evaluations required by the computation of the finite-difference Hessian. This is expected to be even more prominent in higher dimensions (c.f. Table 15).

Having substantially lower function evaluation counts per iteration, the simplex algorithms compare very favourably to gradient descent algorithms when finite-difference derivatives are used. They also use a lower total number of function evaluations than any gradient descent algorithm. This suggests that, at least on low-dimensional problems, they can be highly efficient when evaluating the objective function is time-consuming.

4.4.4 Scale-dependency

The behaviour of the GSL++ and GSL algorithms under scaling of variables was experimentally tested. Each algorithm was run on the quadratic function

$$f(\mathbf{x}) = x^2 + y^2, \quad \mathbf{x} = (x, y)$$

in the coordinates $\tilde{\mathbf{x}}$ transformed according to $T(\mathbf{x}) = (x/\alpha, y) = \tilde{\mathbf{x}}$, i.e.

$$\tilde{f}(\tilde{\mathbf{x}}) = f(T^{-1}(\tilde{\mathbf{x}})) = f(\alpha x, y) = \alpha^2 x^2 + y^2.$$

Each iteration was tested against the stopping criterion

$$\|\tilde{\mathbf{x}} - \mathbf{x}^*\| = \|(\tilde{x}, y)\| = \|(\frac{x}{\alpha}, y)\| < 10^{-6}, \tag{4.4.1}$$

where $\mathbf{x}^* = (0, 0)$ is the minimizer of the objective function. The starting point used in these tests was $\tilde{\mathbf{x}}_0 = T(\mathbf{x}_0) = (\frac{3}{\alpha}, 2.1)$. In a similar fashion, the initial simplex for the simplex algorithm was scaled according to

$$\tilde{\mathbf{x}}^0_{i+1} = \tilde{\mathbf{x}}^0_1 + \lambda_i \mathbf{e}_i, \quad i = 1, 2,$$

where $\tilde{\mathbf{x}}^0_1 = T(\mathbf{x}^0_1) = (3/\alpha, 2.1)$, $\lambda_1 = 1/\alpha$ and $\lambda_2 = 1$. The test results are shown in Figure 21. The required number of iterations to produce an iterate satisfying (4.4.1) is plotted as a function of α in the range $[10^{-6}, 1]$.

Observations

As expected, `steepest_descent` is highly intolerant to scaling of variables. The required number of iterations increases rapidly as α decreases. Possibly due to this reason, `conjgrad_fr_mt` exhibits similar behaviour because of periodic restarting back to the steepest descent direction. It is somewhat surprising that `conjgrad_pr_mt` does not suffer from this. Although its scale-invariance is not theoretically guaranteed, only moderate increase in the number of iterations is observed as α decreases. As a Newton-based algorithm, `bfgs_mt` neither seems to be considerably affected by scaling of variables.

The GSL algorithms using the Brent line search with stopping criterion (3.1.1) are not invariant. Unlike the Wolfe conditions, this criterion tests an angle measure, and thus it is not scale-invariant. In particular, the BFGS algorithm loses its tolerance to scaling with this line search (c.f. `bfgs_mt` and `vector_bfgs`). In addition to the results shown here, `lrwwsimplex` and `mnewton` were observed to be practically invariant to scaling, which agrees with their theoretical results.

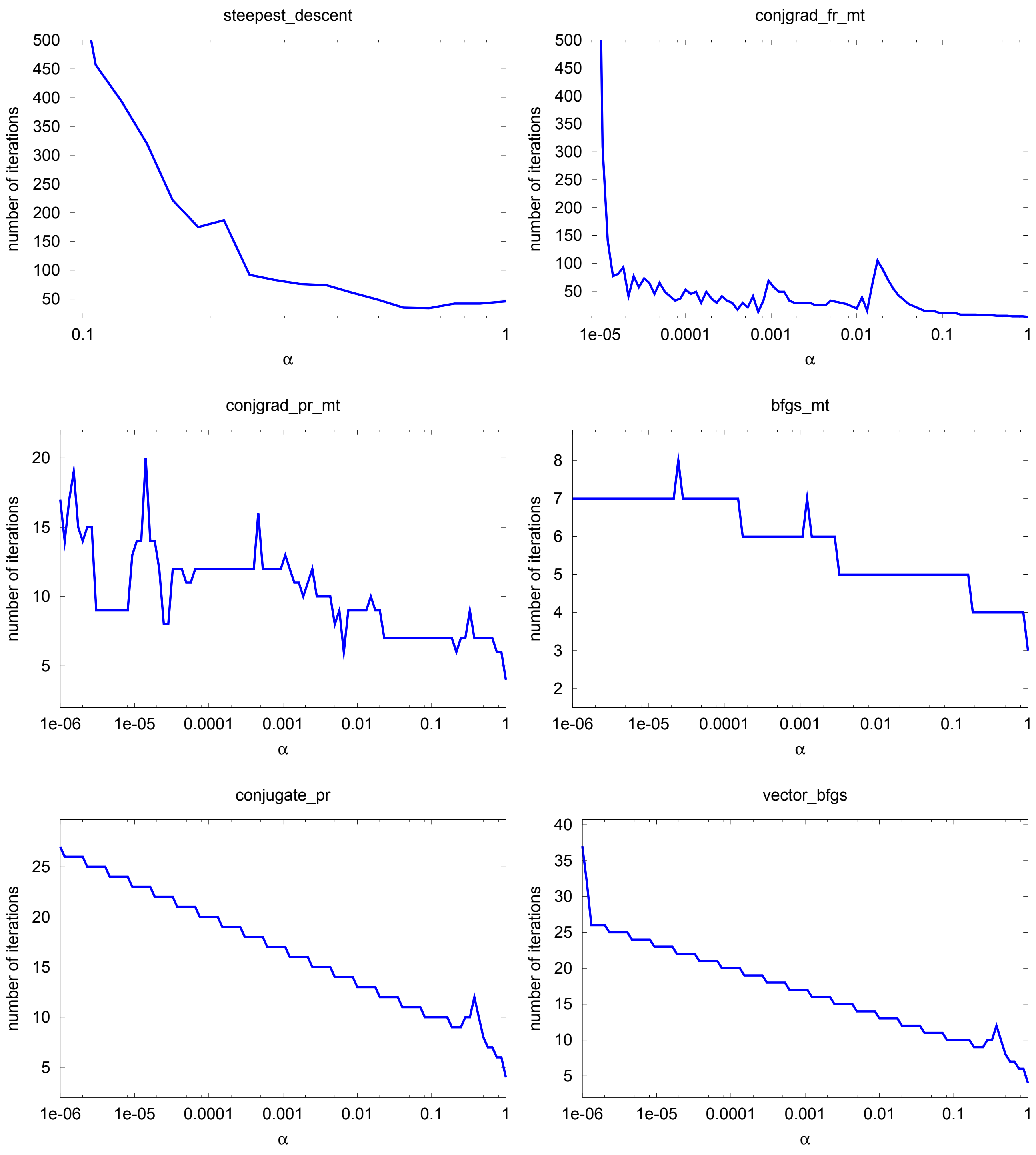


Figure 21: Scale-dependency of minimization algorithms.

4.4.5 Performance profiles

One of the main goals of the testing procedures was to provide an easily interpretable summary of performance differences between the reviewed algorithms. For this purpose, the Dolan and Moré *performance profiles* [DM02] were measured for each GSL++ and GSL minimization algorithm. Following the notation introduced by Dolan and Moré, we denote the set of solvers by $\mathcal{S}$ and the set of test problems by $\mathcal{P}$. The number of solvers is denoted by n_s and the number of test problems is denoted by n_p.

As the absolute computation time is generally hardware-dependent, it is advantageous to use relative test measures for performance comparisons. For this purpose, Dolan and Moré define the *performance ratio* [8]

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} \mid s \in \mathcal{S}\}},$$

where $t_{p,s}$ denotes the computation time required by a solver $s \in \mathcal{S}$ to solve a problem $p \in \mathcal{P}$. This performance ratio describes the relative computation time used by the solver s with respect to the best solver on the problem p.

By using the definition of performance ratio, Dolan and Moré define the probability function

$$\rho_s(\tau) = \frac{1}{n_p} \sum_{p \in \mathcal{P}} \chi_{p,s}(\tau), \tag{4.4.2}$$

where

$$\chi_{p,s}(\tau) = \begin{cases} 0, & r_{p,s} > \tau \\ 1, & r_{p,s} \leq \tau \end{cases}$$

and n_p denotes the number of test problems in $\mathcal{P}$. Expression (4.4.2) can be interpreted as the *cumulative distribution function*. It describes the probability of the solver s being able to solve a fraction of $\rho_s(\tau)$ of all test problems in $\mathcal{P}$ within a relative performance ratio of τ.

The limits of ρ_s are of particular interest. The limit $\lim_{\tau \to \infty} \rho_s(\tau)$ gives the percentage of problems that a solver s is eventually able to solve if computation time is not limited. On the other hand, the value of $\rho_s(1)$ gives the percentace of test problems that a solver s solves faster than any other solver in $\mathcal{S}$.

The measured performance profiles are shown in Figures 22-25. The first set of measurements was carried out with symbolic function evaluation and differentiation. The second set of measurements was carried out by using precompiled test functions written in C with central-difference gradients. Instead of symbolic Hessian, its forward-difference approximations (A.5.3) and (A.5.4) were used with symbolic and finite-difference gradients, respectively. In order to enhance readability, $\rho_s(2^\tau)$ is plotted instead of $\rho_s(\tau)$. This shows more clearly the behaviour of ρ_s near unity.

[8] If solver s is not able to solve problem p, $r_{p,s}$ is set to $r_M \geq r_{p,s}$ for all $p \in \mathcal{P}, s \in \mathcal{S}$.

Test problems

The test problems for measuring performance profiles were obtained from the [MGH81] problem set. The notations

$$\|\mathbf{x}_k - \mathbf{x}^*\|_{REL}, \quad \|f(\mathbf{x}_k) - f(\mathbf{x}^*)\|_{REL}, \quad \|\mathbf{x}_k - \mathbf{x}^*\|, \quad \|f(\mathbf{x}_k) - f(\mathbf{x}^*)\|$$

denote the *relative* and *absolute* errors in the final parameter and function values, respectively. The relative errors are defined as

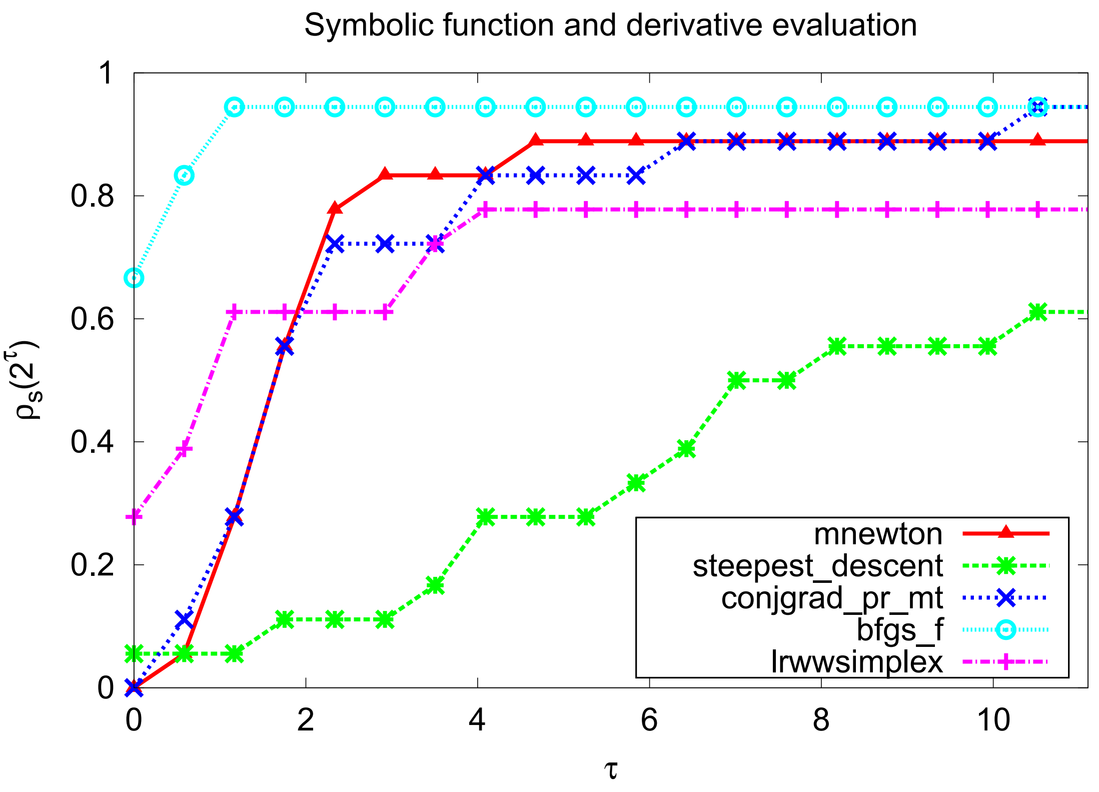
$$\|\mathbf{x}_k - \mathbf{x}^*\|_{REL} = \frac{\|\mathbf{x}_k - \mathbf{x}^*\|}{\|\mathbf{x}^*\|}, \quad \|f(\mathbf{x}_k) - f(\mathbf{x}^*)\|_{REL} = \frac{\|f(\mathbf{x}_k) - f(\mathbf{x}^*)\|}{f(\mathbf{x}^*)},$$

where the stopping criteria are chosen such that $\mathbf{x}^* \neq \mathbf{0}$ and $f(\mathbf{x}^*) \neq 0$. Each test run was also terminated if the stopping criterion was not satisfied after 50000 iterations. The used test functions with their dimensions and stopping criteria are listed in Table 14. [9]

Name	n	m	Stopping criterion
Powell badly scaled			$\|f - f^*\|_{ABS} < 10^{-14}$
Brown badly scaled			$\|\mathbf{x} - \mathbf{x}^*\|_{ABS} < 10^{-6}$
Beale			$\|\mathbf{x} - \mathbf{x}^*\|_{ABS} < 10^{-6}$
helical valley			$\|\mathbf{x} - \mathbf{x}^*\|_{ABS} < 10^{-6}$
Gaussian			$\|f - f^*\|_{REL} < 10^{-4}$
Gulf research and development		5	$\|\mathbf{x} - \mathbf{x}^*\|_{ABS} < 10^{-6}$
Box three-dimensional		5	$\|f - f^*\|_{ABS} < 10^{-6}$
Wood			$\|\mathbf{x} - \mathbf{x}^*\|_{ABS} < 10^{-6}$
Brown and Dennis		20	$\|f - f^*\|_{ABS} < 10^{-1}$
Biggs EXP6		13	$\|f - f^*\|_{REL} < 10^{-4}$
Watson	6		$\|f - f^*\|_{REL} < 10^{-4}$
extended Rosenbrock	10		$\|\mathbf{x} - \mathbf{x}^*\|_{ABS} < 10^{-6}$
extended Powell singular	12		$\|\mathbf{x} - \mathbf{x}^*\|_{ABS} < 10^{-6}$
penalty function I	10		$\|f - f^*\|_{REL} < 10^{-4}$
penalty function II	10		$\|f - f^*\|_{REL} < 10^{-4}$
variably dimensioned	10		$\|\mathbf{x} - \mathbf{x}^*\|_{ABS} < 10^{-6}$
trigonometric	5		$\|f - f^*\|_{ABS} < 10^{-5}$
chebyquad	8	8	$\|f - f^*\|_{REL} < 10^{-5}$

Table 14: Number of test function dimensions (if not default) and stopping criteria used in the performance profile tests.

[9] For notational convenience, we omit the subscripts k and instead use the notations f and f^* for $f(\mathbf{x}_k)$ and $f(\mathbf{x}^*)$, respectively.



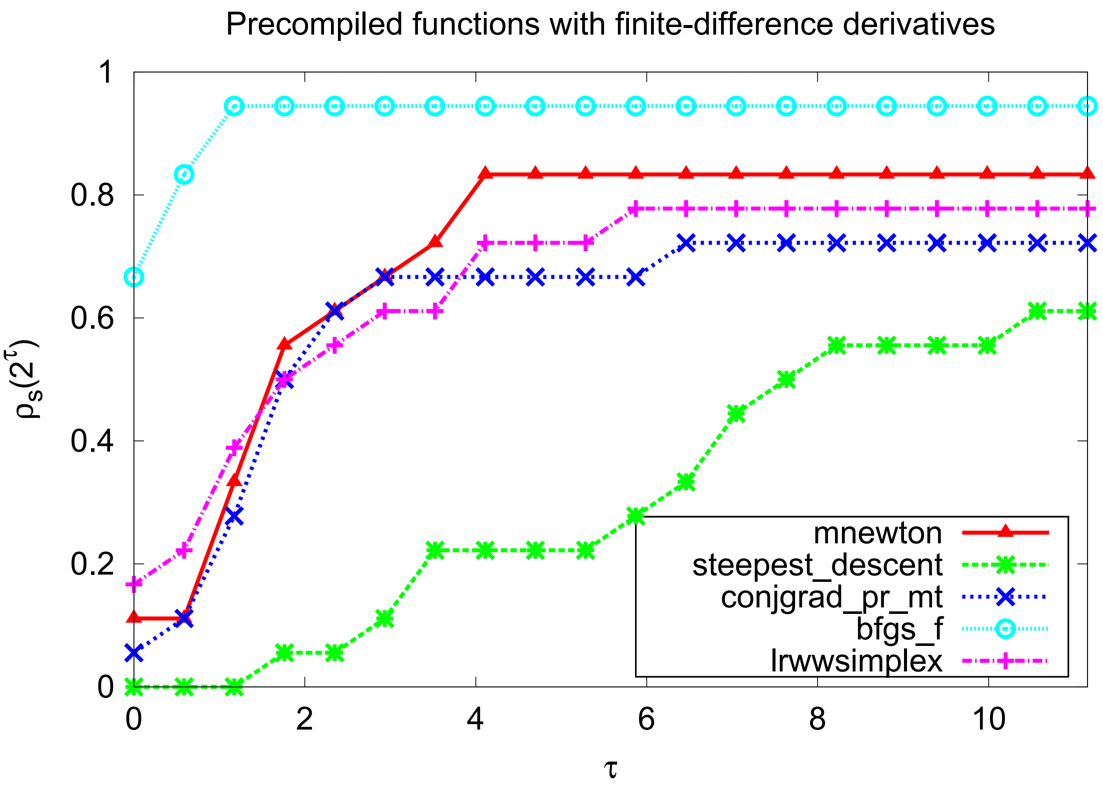
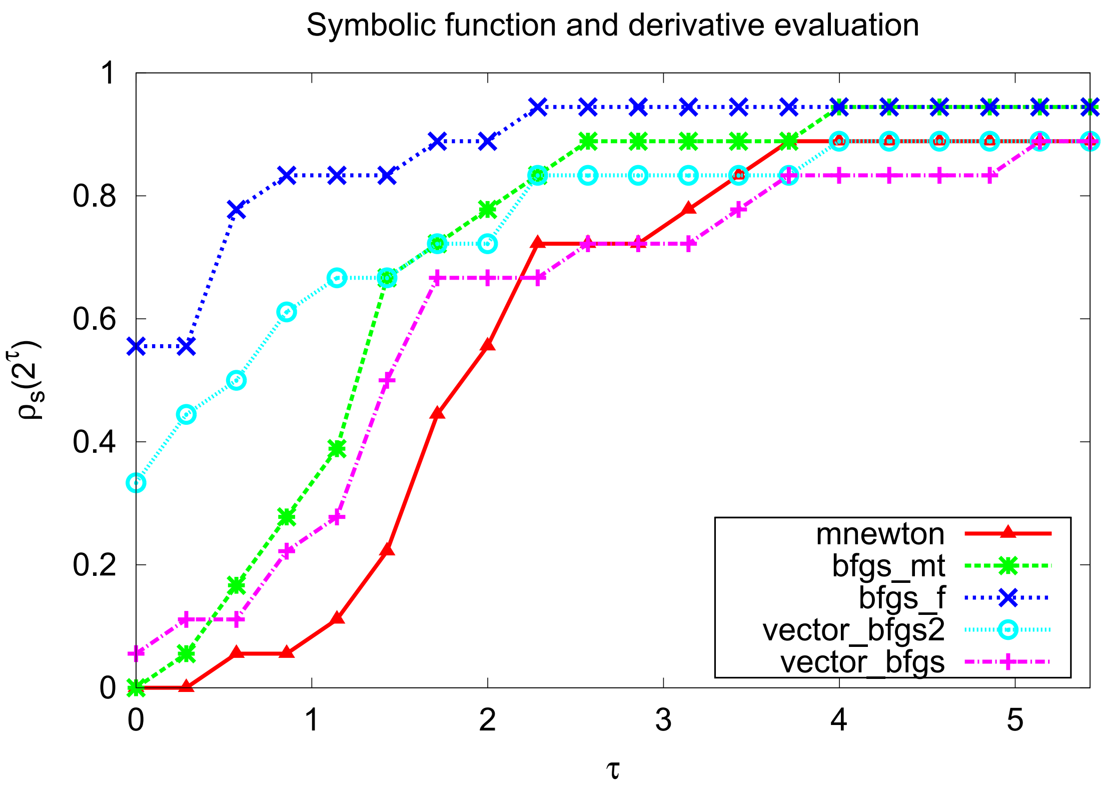


Figure 22: Performance profiles of algorithms of different types.



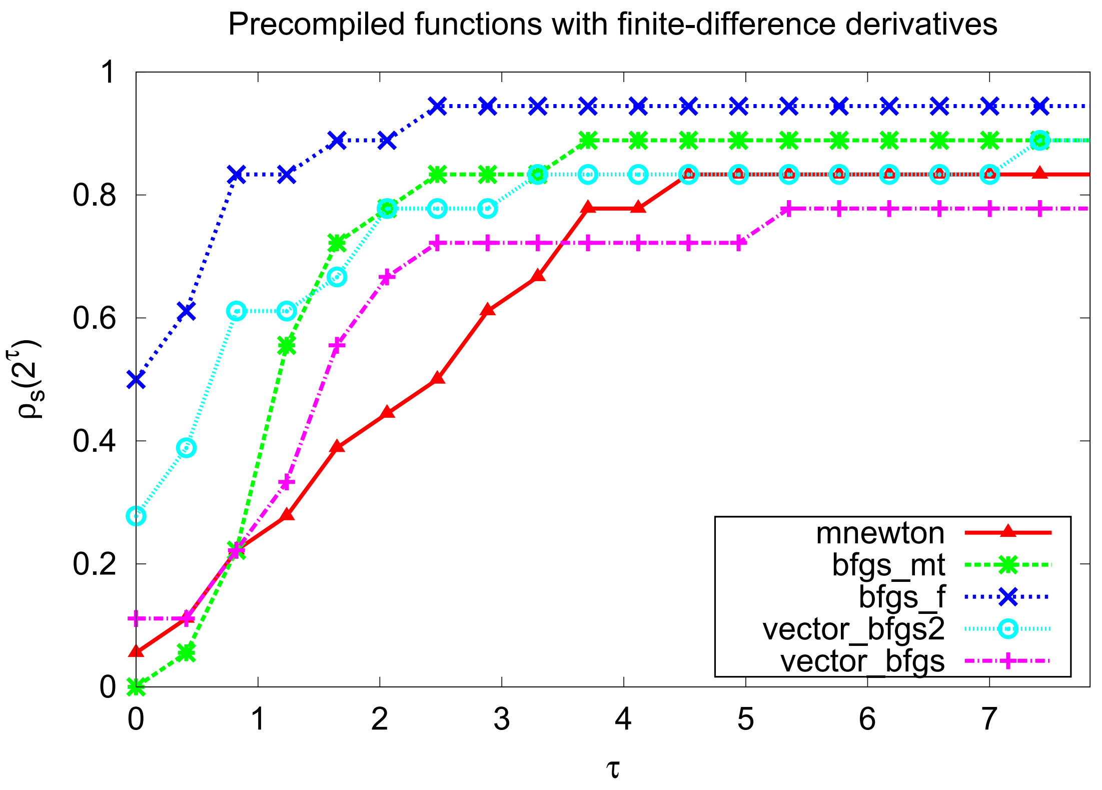
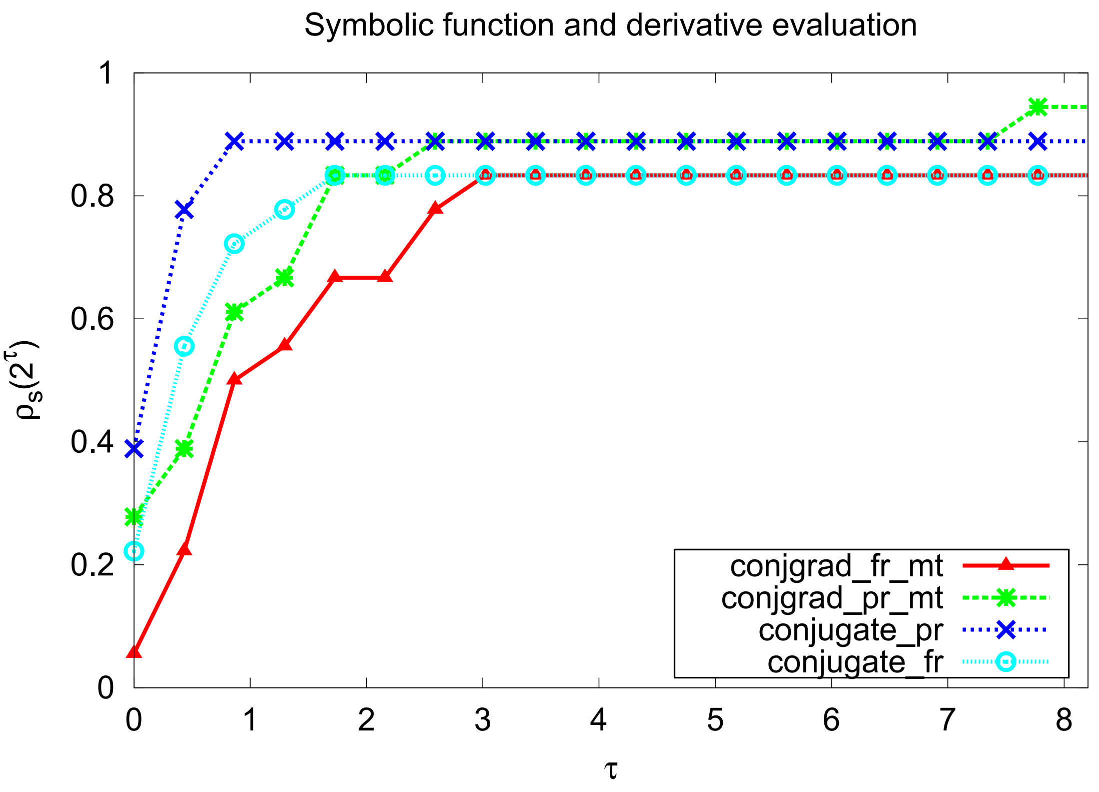


Figure 23: Performance profiles of the GSL++ implementations of Newton-based algorithms and the corresponding GSL implementations.



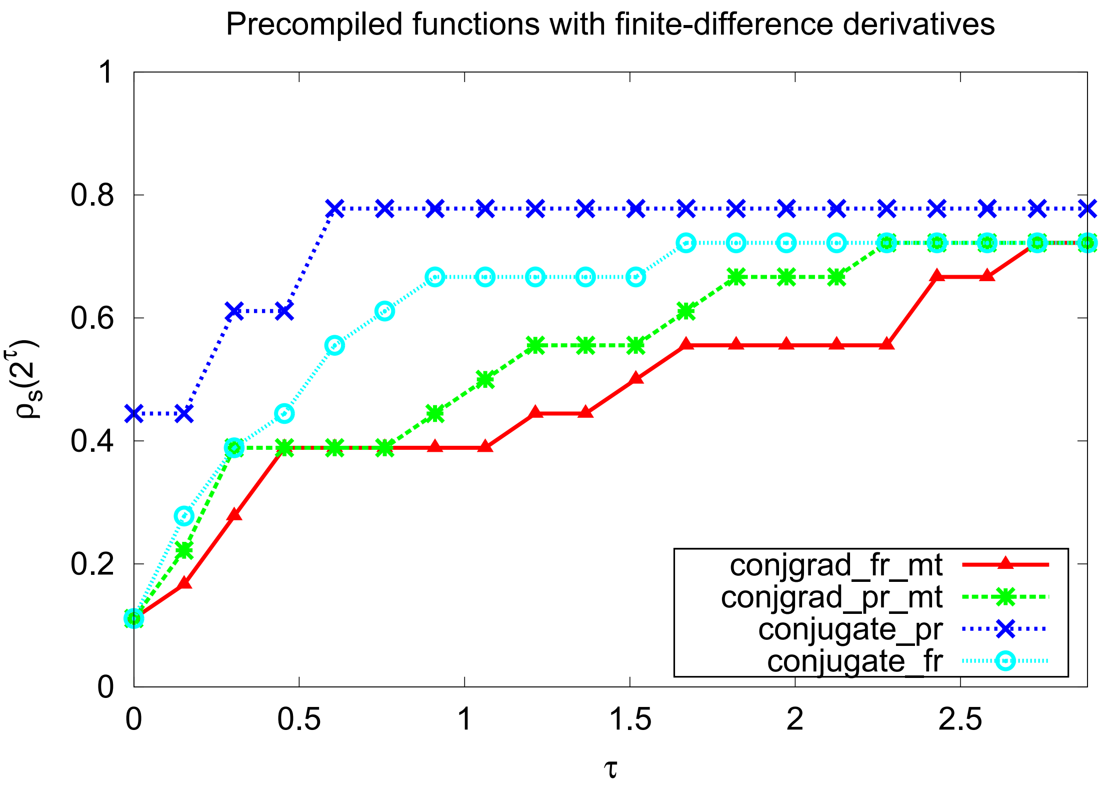
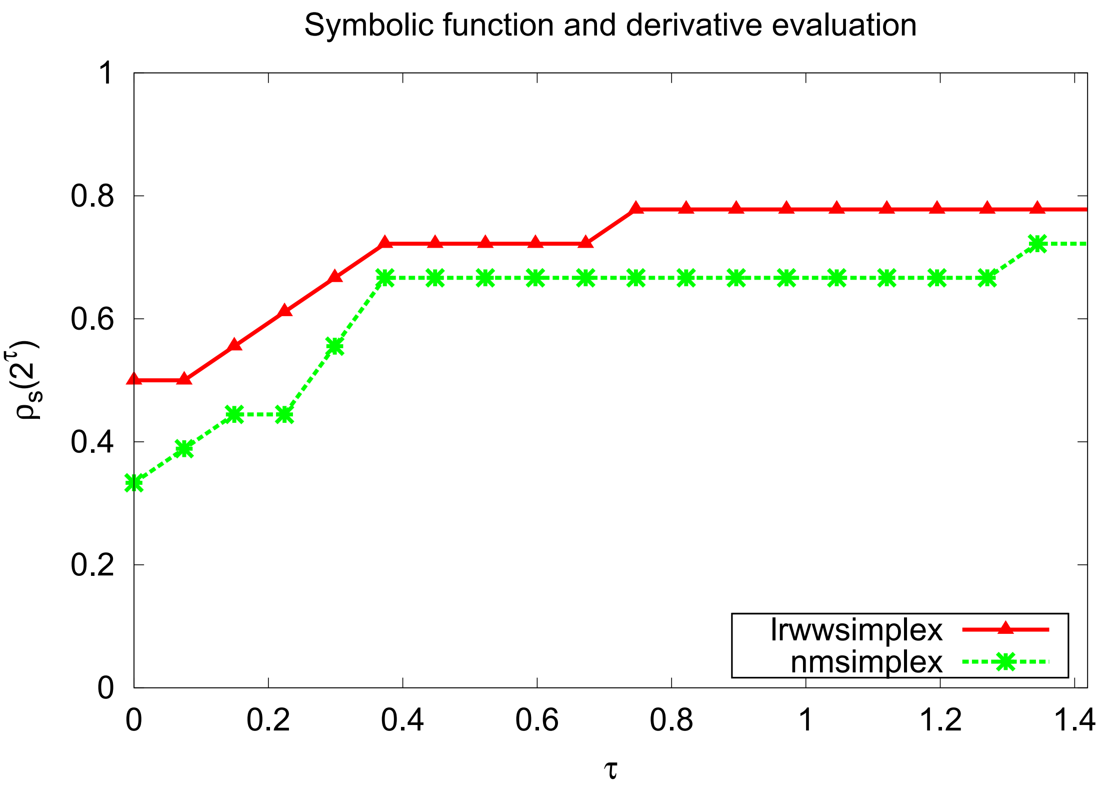


Figure 24: Performance profiles of the GSL++ implementations of conjugate gradient algorithms and the corresponding GSL implementations.



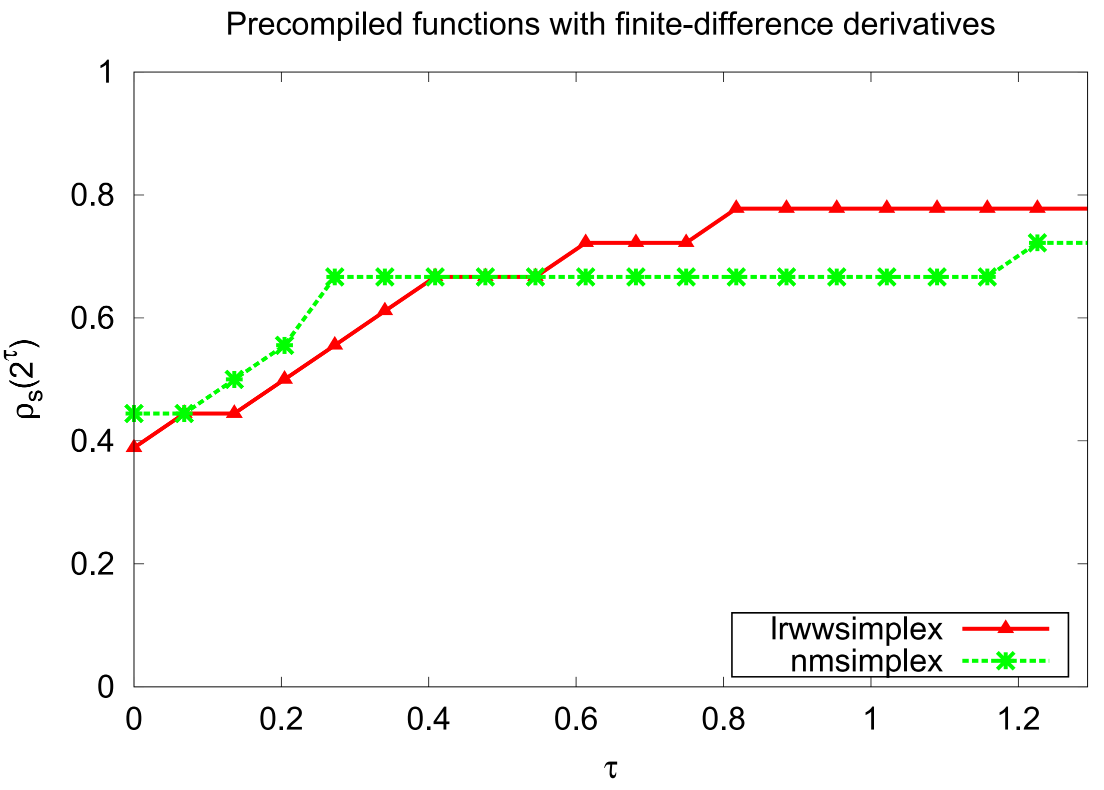


Figure 25: Performance profiles of the GSL++ implementation of the Nelder and Mead simplex algorithm and the corresponding GSL implementation.

Success rates

The BFGS algorithm `bfgs_f` consistently shows superior reliability. It can solve 95% of test problems with symbolic derivatives. This is also the case with finite-difference-derivatives, which implies that Fletcher's line search algorithm with its strict safeguarding rules is very robust. The GSL algorithm `vector_bfgs2` with a modification of Fletcher's line search is slightly less reliable. As observed in Section 4.4.2, its line search algorithm is prone to failures.

The BFGS and the Polak-Ribière algorithms with the Moré and Thuente line search also reach 95% rates of success with symbolic derivatives. Unfortunately, this is not the case with finite-difference derivatives. In this case, they exhibit $5 - 20\%$ losses of reliablity. This indicates that the polynomial interpolation scheme of the Moré and Thuente line search is very sensitive. Its implementation does not impose as strict safeguarding rules as Fletcher's algorithm, which could partly explain this.

Algorithms with the Brent line search satisfying criterion (3.1.1) do not achieve the highest rates of success. This is not surprising, since global convergence with this line search criterion is not guaranteed. Most notably, `vector_bfgs` clearly lags behind all the other Newton-based algorithms with line searches satisfying the Wolfe conditions. The observed 90% rates of success in several cases are nevertheless surprisingly high. In particular, the GSL conjugate gradient algorithms with the Brent line search perform very well achieving similar success rates with the GSL++ implementations.

With symbolic derivatives, the Polak-Ribière algorithms with $88 - 95\%$ rates of success are more robust than the Fletcher-Reeves algorithms with 85% rates. This is somewhat surprising in the light of their convergence theory that guarantees convergence of the Fletcher-Reeves algorithm under more general conditions. This is likely caused by stagnations, and in effect, failures of the Fletcher-Reeves algorithm, as the results of Section 4.3.3 imply.

An interesting observation is that all conjugate gradient algorithms suffer from $10 - 20\%$ losses of reliability when finite-difference approximations are used instead of symbolic derivatives. This could be partly explained by the form of equations (2.3.43) and (2.3.44) which suggests that these algorithms are very sensitive to errors in the denominators of these equations. As it was observed in Section 4.4.4, conjugate gradient algorithms are also sensitive to scaling, which can be an additional source of failures.

The lack of reliability of the Nelder and Mead simplex algorithms is very disappointing. As it was observed in Section 4.3.1, they consistently fail on higher-dimensional problems ($n \geq 10$), for which reason their rates of success remain below 80%. This success rate however becomes much more competitive when finite-difference derivatives are used, and gradient descent algorithms lose their advantage. The GSL++ implementation `lrwwsimplex` is slightly more reliable with success rate of nearly 80% compared to the GSL implementation `nmsimplex` with success rate of little higher than 70%. The steepest descent algorithm `steepest_descent` also yields very low rate of success, because it likely failed on most badly scaled test problems.

The modified Newton algorithm `mnewton` neither achieves the highest success rates. Its $\mathbf{LDL}^T$ decomposition does not seem to lead to convergent iteration in all cases. Although this decomposition is guaranteed to produce positive definite matrices, the parameter μ can become too large. In such a case, the iteration fails because it effectively becomes the steepest descent iteration. Running it with symbolic Hessians could possibly improve its reliability.

Performance comparison, Figure 22

These results show that `bfgs_f` has superior performance. It is the fastest algorithm on 65% of test problems. This is also the case with finite-difference derivatives. The `conjgrad_pr_mt` algorithm uses substantially higher computation times than `bfgs_f` due to its computationally more expensive Moré and Thuente line search routine. The `mnewton` algorithm, which also uses the same line search routine, exhibits very similar performance. Its faster convergence probably explains why it achieves its final success rates with a smaller computational effort.

The simplex algorithm `lrwwsimplex` that uses a relatively small number of function evaluations is the fastest algorithm on 30% of test problems with symbolic evaluation. However, it loses this advantage on precompiled test functions, in which case function evaluations have smaller contribution to performance. It is also not surprising that slowly converging `steepest_descent` algorithm requires very high computation times for achieving its highest success rates.

Performance comparison, Figure 23

BFGS algorithms with Fletcher's line search have the best performance with both symbolic and finite-difference derivatives. Again, `bfgs_f` leads all the others. This time it is the fastest algorithm on about 55% of test problems. It also outperforms `vector_bfgs2` that uses a more complex line search algorithm.

The `bfgs_mt` algorithm has a substantially higher computational cost than `bfgs_f` due to its expensive line search routine. Also the `mnewton` and `vector_bfgs` algorithms are both clearly lagging behind the other algorithms. This further reinforces the earlier observations that the Brent line search might not be a good choice for the BFGS algorithm. Neither is the rapid convergence of the `mnewton` algorithm enough to give it an advantage over the other Newton-based algorithms.

Performance comparison, Figure 24

As observed in Section 4.4.2, the Polak-Ribière algorithms exhibit faster convergence rates than the Fletcher-Reeves algorithms, which also seems to give them a clear performance advantage. They are the fastest algorithms on most test problems.

The Brent line search seems to yield better performance than the Moré and Thuente line search. The GSL++ implementations are seriously lagging behind the GSL implementations in this case. This is not surprising in the light of the results discussed in Section 4.4.3. They showed that the the Brent line search can be significantly more effective than the latter in terms of function and gradient evaluation counts.

Performance comparison, Figure 25

Considering only minor differences in their implementations, the performance difference of the two simplex algorithms is surprisingly large. The reason for this is difficult to explain. A careful inspection of the GSL source code however showed that it unnecessarily computes (3.1.2) even if it is not tested. The expensive square root computations can cause some performance differences.

4.4.6 Asymptotic complexity

An important performance measure of a minimization algorithm is that how its performance scales when the problem dimension increases. The computation times per iteration were measured for several GSL++ and GSL algorithms. Each algorithm was iterated ten times on the extended Rosenbrock function, and the average computation time was measured. In order to minimize the contribution of function evaluations to the measured times, this test was carried out on the precompiled extended Rosenbrock function. Central-difference derivatives (A.5.2) were used. The results are shown in Figure 26.

The measured computation times are in fact dominated by line search routines and not by matrix computations. This argument is based on the observation that as a conjugate gradient algorithm, `conjgrad_pr_mt` should behave according to its complexity of $\mathcal{O}(n)$ compared to the BFGS algorithm `bfgs_mt` with complexity of $O(n^2)$. On the contrary, `bfgs_f` with its simpler line search routine outperforms `conjgrad_pr_mt`. It is also faster than `vector_bfgs2`, which has a more complex line search routine.

The fundamental reason for the high computational complexity of line search routines is that they require a high number of function and gradient evaluations. This concerns both symbolic differentiation and finite-difference approximations. This boils down to the fundamental limitation that evaluating a very high-dimensional objective function is computationally expensive no matter how it is done. It should also be emphasized that due to the highly optimized nature of BLAS, linear algebra operations are carried out very efficiently.

Another observation of these results is that `lrwwsimplex`, which does not use line searches, outperforms all the other algorithms by a wide margin. This observation also agrees with the results of Section 4.4.3, which showed that it uses a substantially lower number of function evaluations than gradient-based methods with finite-difference derivatives.

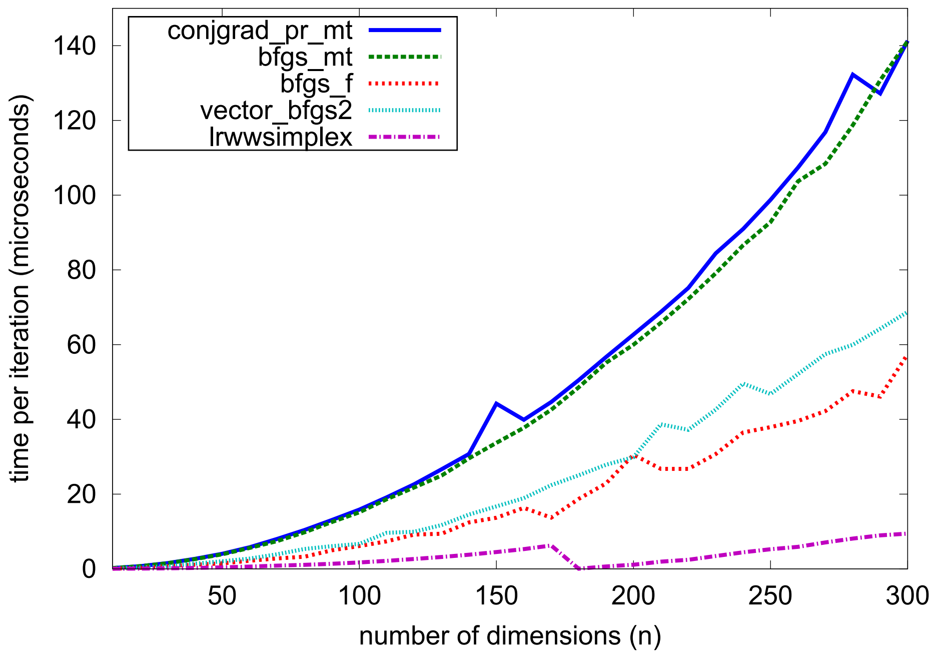


Figure 26: Used computation times per iter iteration as a function of problem dimension, extended Rosenbrock function.

Chapter 5

Conclusions and discussion

In Chapter 2, we concluded that most unconstrained optimization algorithms have well-established convergence theory. Building algorithms on the convergence theory described in Chapter 2 led to robust and efficient implementations that behaved predictably according to their theoretical results. Also some of the individual strengths and possible weaknesses of the reviewed algorithms were identified.

The existing GSL algorithms are already very fine-tuned, and there is only little room for improvement. The test results of the algorithms implemented in GSL++ however showed several small but promising improvements over the existing GSL implementations. Since the algorithms implemented in GSL++ are compliant with the existing GSL interface, they can be be used as replacements for the existing GSL algorithms with only minor modifications.

Evaluating the performance of a minimization algorithm is indeed a difficult task. There is no definite answer for which algorithm has the best overall performance. It was also seen that evaluating the objective function and its derivatives can have a major impact on performance depending on how they are supplied to minimization algorithms. It should also be emphasized that all test results depend on the choice of test problems. Tests on high-dimensional or noisy functions were not carried out in this thesis, which would have led to more conclusive results.

It was shown that The Nelder and Mead simplex algorithm has serious shortcomings. It consistently failed to converge in higher dimensions. Some kind of restarting proceduce could effectively force the iteration to converge if it should stall, see e.g. [Kel99, p. 141]. Provably convergent variants of this algorithm have also been developed [PCB02]. However, all these variants have higher computational complexity. They have been used in very few practical implementations, and thus they were not reviewed in this thesis.

The nonlinear conjugate gradient algorithms were not observed to have any particular advantage over any of the other algorithms. Furthermore, they require accurate line searches that preferably satisfy the strong Wolfe conditions. This makes programming an efficient conjugate gradient implementation very difficult. They seem to be best suited for problems where evaluating the objective function is computationally very cheap. For these reasons, they are most commonly used for solving linear equations.

The modified Newton method showed mixed results. It consistently exhibited the fastest convergence rates. However it suffered from unexplained failures on several test problems. It was also observed to have a very high computational complexity, which questions its usability in practical applications.

The BFGS algorithms were the most reliable and efficient of the reviewed algorithms. They were the only algorithms that were able to solve nearly all test problems in the [MGH81] problem set with consistent performance. In particular, the `bfgs_f` algorithm with Fletcher's line search was superior to all the other algorithms. It was also observed that requiring the strong Wolfe conditions with the BFGS algorithm is unnecessary. The `bfgs_mt` algorithm with its Moré and Thuente line search exhibited very poor performance compared to `bfgs_f`. Moreover, it was not able to solve any larger number of test problems than its faster competitor.

The comparison of line search algorithms yielded perhaps the most interesting results. They seemed to play a major role in the overall performance of gradient descent algorithms. It was also observed that the line search algorithms satisfying the Wolfe or the strong Wolfe conditions yielded the best reliability of the algorithms they were embedded into. No similar success rates were observed with algorithms using the simpler Brent line search. The verdict is that one should use as simple a line search algorithm as possible without sacrificing reliability or global convergence of the multidimensional algorithm.

Possible future extensions of GSL++ include implementing *trust region* methods [Kel99, p. 50-63], which are a viable alternative to line searches. In particular, it has been claimed that quasi-Newton methods with the *SR1 formula* perform remarkably well with this approach [CGT91]. There also exists specific methods for solving least-squares problems such as the trust region-based Levenberg-Marquardt method and the Gauss-Newton method, which were not studied in this thesis. Also the modified simplex algorithm discussed in [PCB02] could be worth testing. A broader extension is to implement a framework for *constrained* optimization, [NW99, p. 314-357]. GSL has no interfaces for implementing trust-region algorithms or algorithms for constrained optimization. However, they can be implemented on top of the existing GSL code basis with relatively little effort.

References

[AB85] M. Al-Baali, *Descent property and global convergence of the Fletcher-Reeves method with inexact line search*, IMA Journal of Numerical Analysis **5** (1985), 121–124.

[Aka59] H. Akaike, *On a successive transformation of probability distribution and its application to the analysis of the optimum gradient method*, Annals of the Institute of Statistical Mathematics **11** (1959), 1–17.

[BCP04] D. Byatt, I.D. Coope, and C.J. Price, *Effect of limited precision on the BFGS quasi-Newton algorithm*, Anziam **45 (E)** (2004), C283–C295.

[Ber99] D.P. Bertsekas, *Nonlinear Programming*, second ed., Athena Scientific, Belmont, 1999.

[BNY87] R.H. Byrd, J. Nocedal, and Y. Yuan, *Global convergence of a class of quasi-Newton methods on convex problems*, SIAM Journal on Numerical Analysis **24** (1987), 1171–1190.

[Bre73] R.P. Brent, *Algorithms for Minimization With Derivatives*, Prentice Hall, Eaglewood Cliffs, 1973.

[Bro70] C.G. Broyden, *The convergence of a Class of Double-rank Minimization Algorithms, parts I and II*, Journal of the Institute of Mathematics and Its Applications **6** (1970), 76–90,222–236.

[Cau48] A. Cauchy, *Méthode générale pour la résolution des systémes d'équations simultanées*, 1848.

[CGT91] A.R. Conn, N.I.M Gould, and P.L. Toint, *Convergence of quasi-Newton matrices generated by the symmetric rank one update*, Mathematical Programming **50** (1991), 177–195.

[Coh72] A. Cohen, *Rate of Convergence of Several Conjugate Gradient Algorithms*, SIAM Journal on Numerical Analysis **9** (1972), no. 2, 248–259.

[Cor01] T.H. Cormen, *Introduction to Algorithms*, MIT Press, 2001.

[Dix72] L.C.W. Dixon, *Variable Metric Algorithms: Necessary and Sufficient Conditions for Identical Behavior of Nonquadratic Functions*, Journal of Optimization Theory and Applications **10** (1972), no. 1, 34–40.

[DM74] J.E. Dennis and J.J. Moré, *A Characterization of Superlinear Convergence and Its Application to Quasi-Newton Methods*, Mathematics of Computation **28** (1974), no. 126, 549–560.

[DM77] ———, *Quasi-Newton Methods, Motivation and Theory*, SIAM Review **19** (1977), 46–89.

[DM02] E.D. Dolan and Jorge J. Moré, *Benchmarking optimization software with performance profiles*, Mathematical Programming **91** (2002), no. 2, 201–213.

[DS83] J.E. Dennis and R.B. Schnabel, *Numerical methods for unconstrained optimization and nonlinear equations*, Prentice-Hall, Englewood Cliffs, 1983.

[Eat02] J.W. Eaton, *GNU Octave Manual*, Network Theory Limited, 2002.

[Fle70] R. Fletcher, *A New Approach to Variable Metric Algorithms*, The Computer Journal **13** (1970), 317–322.

[Fle80] ———, *Practical methods of optimization*, vol. 1: Unconstrained optimization, Wiley, Chichester, 1980.

[FR64] R. Fletcher and C.M. Reeves, *Function minimization by conjugate gradients*, Computer Journal **7** (1964), 149–154.

[GC91] A. Griewank and G.F. Corliss (eds.), *Automatic Differentiation of Algorithms: theory, implementation and application*, SIAM Publications, Philadelphia, 1991.

[GM74] P.E. Gill and W. Murray, *Newton-type methods for unconstrained and linearly constrained optimization*, Mathematical Programming **7** (1974), 311–350.

[GMW81] P.E. Gill, W. Murray, and M.H. Wright, *Practical Optimization*, Academic Press, London, 1981.

[GMW91] ———, *Numerical linear algebra and optimization*, vol. Volume 1, Addison-Wesley, Redwood city, 1991.

[GN92] J. Gilbert and J. Nocedal, *Global Convergence Properties of Conjugate Gradient Methods for Optimization*, SIAM Journal on Optimization **2** (1992), 21–42.

[Gol70] D.A. Goldfarb, *A Family of Variable-Metric Methods Derived by Variational Means*, Mathematics of Computation **24** (1970), no. 109, 23–26.

[Gre70] J. Greenstadt, *Variations on Variable-Metric Methods*, Mathematics of Computation **24** (1970), 1–22.

[GvL89] G.H. Golub and C.F. van Loan, *Matrix Computations*, second ed., Johns Hopkins University Press, Baltimore, 1989.

[HJ85] R.A. Horn and C.R. Johnson, *Matrix Analysis*, Cambridge University Press, Cambridge, 1985.

[HS52] M.R. Hestenes and E. Stiefel, *Methods of Conjugate Gradients for Solving Linear Systems*, Journal of Research of the National Bureau of Standards **49** (1952), 409–436.

[Kel99] C.T. Kelley, *Iterative Methods for Optimization*, SIAM Publications, Philadelphia, 1999.

[KR88] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, second ed., Eaglewood Cliffs, 1988.

[LRWW98] J.C. Lagarias, J.A. Reeds, M.H. Wright, and P.E. Wright, *Convergence properties of the Nelder-Mead simplex method in low dimensions*, SIAM Journal on Optimization **9** (1998), 112–147.

[Lue84] D.G. Luenberger, *Linear and Nonlinear Programming*, second ed., Addison-Wesley, Reading, 1984.

[McK98] K.I.M McKinnon, *Convergence of the Nelder-Mead Simplex Method to a Nonstationary Point*, SIAM Journal on Optimization **9** (1998), no. 1, 148–158.

[Mez94] J.C. Meza, *OPT++: An Object-Oriented Class Library for Nonlinear Optimization*, Tech. Report SAND94-8225, Sandia National Laboratories, Livermore, California, 1994.

[MGH81] J.J. More, B.S. Garbow, and K.E. Hillstrom, *Testing Unconstrained Optimization Software*, ACM Transactions on Mathematical Software **7** (1981), no. 1, 17–41.

[MT94] J.J. Moré and D.J. Thuente, *Line Search Algorithms with Guaranteed Sufficient Decrease*, ACM Transactions on Mathematical Software **20** (1994), 286–307.

[NM65] J.A. Nelder and R. Mead, *A simplex method for function minimization*, The Computer Journal **7** (1965), 308–313.

[NW99] J. Nocedal and S.J. Wright, *Numerical Optimization*, Springer Verlag, 1999.

[PCB02] C.J. Price, D. Coope, and D. Byatt, *A Convergent Variant of the Nelder-Mead Algorithm*, Journal of Optimization Theory and Applications **113** (2002), 5–19.

[Pow76] M.J.D. Powell, *Some global convergence properties of a variable metric algorithm for minimization without exact line searches*, Nonlinear Programming, SIAM-AMS Proceedings (R.W. Cottle and C.E. Lemke, eds.), vol. 9, SIAM Publications, 1976, pp. 53–72.

[Pow77] ______, *Restart Procedures for the Conjugate Gradient Method*, Mathematical Programming **12** (1977), no. 1, 241–254.

[Pow83] ______, *Nonconvex Minimization Calculations and the Conjugate Gradient Method*, Numerical Analysis (1983).

[PR69] E. Polak and G. Ribière, *Note sur la convergence de méthodes directions conjugées*, Revue Française d'Informatique et de Recherche Opérationnelle, 3e Année **16** (1969), 35–43.

[PTVF07] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical recipes: the art of scientific computing*, third ed., Cambridge University Press, New York, 2007.

[Sha70] D.F. Shanno, *Conditioning of Quasi-Newton Methods for Function Minimization*, Mathematics of Computation **24** (1970), 647–657.

[Str00] B. Stroustrup, *The C++ Programming Language*, special ed., Addison-Wesley, 2000.

[Tor89] V.J. Torczon, *Multi-Directional Search: A Direct Search Algorithm for Parallel Machines*, Ph.D. thesis, Rice University, Houston, Texas, 1989.

[Ueb97] C.W. Ueberhuber, *Numerical computation: methods, software and analysis*, vol. Vol. 1, Springer, Berlin, 1997.

[Wol69] P. Wolfe, *Convergence conditions for ascent methods*, SIAM Rev. **11** (1969), 226–235.

[Wol71] ———, *Convergence conditions for ascent methods II: Some corrections*, SIAM Rev. **13** (1971), 185–188.

[Wol03] S. Wolfram, *The Mathematica Book*, fifth ed., Wolfram Media, 2003.

[Wri94] S.J. Wright, *Compact storage of Broyden-class quasi-Newton matrices*, preprint, Argonne National Laboratory, Argonne, IL, 1994.

Appendix A

Supplementary algorithms

A.1 The backtracking algorithm

The most basic algorithm that produces a step length satisfying the sufficient decrease condition (2.3.7) is *backtracking*, [Kel99, p. 39-40]. It reduces the step length α_k from the given initial value α_0 by a factor of $\sigma < 1$ until the sufficient decrease condition (2.3.7) is satisfied. This is mathematically formulated as choosing $\alpha_k = \sigma^m \alpha_0$ such that

$$m = \min_{m \in \mathbb{N}} \{\phi(\sigma^m \alpha_0) \leq \phi(0) + \mu \sigma^m \alpha_0 \phi^{'}(0)\}.$$

This algorithm is computationally very inexpensive. Each iteration only consists of testing the sufficient decrease condition (2.3.7) and reducing the step length α_k, if necessary. However, it has the disadvantage that it is not guaranteed to produce step lengths that satisfy condition (2.3.9) or (2.3.8). Consequently, global convergence of algoritms that require stricter line search conditions is not guaranteed. The statement of this algorithm with a modification adopted from GSL (lines 4-7) is given below. [1]

Algorithm 9: The backtracking algoritm

```
1 $\alpha \leftarrow \alpha_0$
2 while $\phi(\alpha) > \phi(0) + \mu\alpha\phi'(0)$ do
3     $\alpha \leftarrow \sigma\alpha$
4 if $\alpha = \alpha_0$ then
5     $\alpha_0 \leftarrow \rho\alpha_0$
6 else
7     $\alpha_0 \leftarrow \alpha$
```

Lines 4-7 implement a simple but effective strategy for accelerating convergence. If the first iteration step of the loop at lines 2-3 is successful, the algorithm multiplies the initial step length α_0 by a factor of ρ, where $\rho > 1$. On the other hand, if the first iteration is not successful, α_0 is left unmodified. This strategy alongside retaining the initial step length α_0 between subsequent calls of this routine effectively prevents the initial step lengths α_0 from becoming too small.

[1] Instead of the condition at line 2, the GSL implementation tests a simpler condition $\phi(\alpha) > \phi(0)$. It also uses the normed gradient $-\frac{\nabla f(\mathbf{x}_k)}{\|\nabla f(\mathbf{x}_k)\|}$ as the search direction $\mathbf{d}_k$.

A.2 Fletcher's line search algorithm

Fletcher's line search algorithm [Fle80, p. 26-28] is specifically aimed for producing step lengths that satisfy the weaker Wolfe conditions (2.3.7) and (2.3.9). Following the notation introduced in section 2.3.2, we denote the admissible step length intervals by

$$T(\mu, \eta) = \{\alpha \in \mathbb{R} \mid \alpha \in T_s(\mu), \alpha \in T_c(\eta)\},$$

where

$$\begin{aligned} T_s(\mu) &= \{\alpha > 0 \mid \phi(\alpha) \leq \phi(0) + \mu\alpha\phi'(0)\}, \\ T_c(\eta) &= \{\alpha > 0 \mid \phi'(\alpha) \geq \eta\phi'(0)\}. \end{aligned}$$

The algorithm maintains the current step length interval $I_k = [\alpha_l^k, \alpha_u^k]$ and the current trial step length $\alpha_t^k \in I_k$. Each iteration of the algorithm consists of two loops. The outer loop (lines 2-8) aims to produce a trial step $\alpha_t^{k+1} \in T_c(\eta)$ by extrapolation, i.e. choosing an $\alpha_t^{k+1} \in]\alpha_t^k, \alpha_u^k[$. On the other hand, the inner loop (lines 3-4) is repeated until a step length $\alpha_t^{k+1} \in T_s(\mu)$ is found by interpolation, i.e. choosing an $\alpha_t^{k+1} \in]\alpha_l^k, \alpha_t^k[$. If after this loop, $\alpha_t^k \in T_c(\eta)$, and thus $\alpha_t^k \in T(\mu, \eta)$, it is accepted, and the algorithm terminates. If $\alpha_t^k \notin T_c(\eta)$, a new extrapolated trial step is generated. This step length is no longer guaranteed to be in $T_s(\mu)$, and the iteration is restarted with an interpolation loop.

Algorithm 10: Fletcher's line search algorithm.

```
1 Choose $\mu \in ]0, \frac{1}{2}[$, $\eta \in [\mu, 1[$, $\tau \in ]0, \eta[$, $\chi \in ]\tau, \infty[$
2 for $k = 0, 1, \ldots$ do
      /* Step 1.:interpolation                                  */
3     while $\alpha_t^k \notin T_s(\mu)$ do
4         Choose $\alpha_t^{k+1} \in ]\alpha_l^k, \alpha_t^k[$ by using Algorithm 11.
5         $k \leftarrow k + 1$
      /* Step 2.:extrapolation                                  */
6     if $\alpha_t^k \in T_c(\eta)$ then
7         Terminate and accept $\alpha_t^k$.
8     else
9         Choose $\alpha_t^{k+1} \in ]\alpha_t^k, \alpha_u^k[$ by using Algorithm 12.
```

The interpolation step of Fletcher's algorithm is specified in Algorithm 11. It obtains a trial step α_t^{k+1} from equation (2.3.26). The safeguarding rule with the constant τ forces α_t^{k+1} to the interval $]\alpha_l, \alpha_u[$ and also prevents the trial step from being too close its endpoints by setting $\alpha_t^{k+1} \in]\alpha_l^k + \tau\Delta\alpha, \alpha_u^k - \tau\Delta\alpha[$, where $\Delta\alpha = \alpha_u^k - \alpha_l^k$. Each iteration of this step in the inner loop of Algorithm 10 contracts the interval I_k from the right-hand side. This guarantees that condition (2.3.7) is eventually satisfied. [2]

Algorithm 11: Fletcher's line search algorithm, interpolation step.

```
1  $\alpha_t^+ \leftarrow \alpha_l + \frac{(\alpha_t-\alpha_l)^2 f'(\alpha_l)}{2[f(\alpha_l)-f(\alpha_t)+(\alpha_t-\alpha_l)f'(\alpha_l)]}$        /* eq.  (2.3.26) */
2  $\Delta\alpha \leftarrow \alpha_u - \alpha_l$
3  if $\alpha_t^+ < \alpha_l + \tau\Delta\alpha$ then
4      $\alpha_t^+ \leftarrow \alpha_l + \tau\Delta\alpha$
5  else if $\alpha_t^+ > \alpha_u - \tau\Delta\alpha$ then
6      $\alpha_t^+ \leftarrow \alpha_u - \tau\Delta\alpha$
7  $\alpha_l^+ \leftarrow \alpha_l$
8  $\alpha_u^+ \leftarrow \alpha_t$
```

The extrapolation step is specified in Algorithm 12. This step obtains a trial step α_t^{k+1} from equation (2.3.27). As for the interpolation step, the computed trial step is explicitly forced to lie within the interval $]\alpha_t^k, \alpha_u^k[$ such that $\alpha_t^{k+1} \in [\alpha_t^k + \tau\Delta\alpha, \alpha_t^k + \chi\Delta\alpha]$, where $\Delta\alpha = \alpha_t^k - \alpha_l^k$ and $\chi > \tau$. In order to enforce that $\alpha_t^{k+1} < \alpha_u^k$, an additional safeguarding condition $\alpha_t^{k+1} - \alpha_t^k \leq \frac{1}{2}(\alpha_u^k - \alpha_t^k)$ is imposed.

Algorithm 12: Fletcher's line search algorithm, extrapolation step.

```
1  $\alpha_t^+ \leftarrow \alpha_t + \frac{(\alpha_t-\alpha_l)f'(\alpha_t)}{f'(\alpha_l)-f'(\alpha_t)}$        /* eq.  (2.3.27) */
2  $\Delta\alpha \leftarrow \alpha_t - \alpha_l$
3  if $\alpha_t^+ < \alpha_t + \tau\Delta\alpha$ then
4      $\alpha_t^+ \leftarrow \alpha_t + \tau\Delta\alpha$
5  else if $\alpha_t^+ > \alpha_t + \chi\Delta\alpha$ then
6      $\alpha_t^+ \leftarrow \alpha_t + \chi\Delta\alpha$
7  if $\alpha_t^+ - \alpha_t > \frac{1}{2}(\alpha_u - \alpha_t)$ then
8      $\alpha_t^+ \leftarrow \alpha_t + \frac{1}{2}(\alpha_u - \alpha_t)$
9  $\alpha_l^+ \leftarrow \alpha_t$
10 $\alpha_u^+ \leftarrow \alpha_u$
```

Each extrapolation step contracts the interval I_k from the left-hand side, which in conjunction with interpolation steps guarantees convergence of the algorithm to a limit $\alpha^* \in I_0$. This limit is however not guaranteed to satisfy the Wolfe conditions, since Fletcher provides no convergence results for his algorithm. These steps are illustrated in Figures 27 and 28.

[2] As in section 2.3.2, we omit superscripts k and denote indices $k+1$ by $+$.

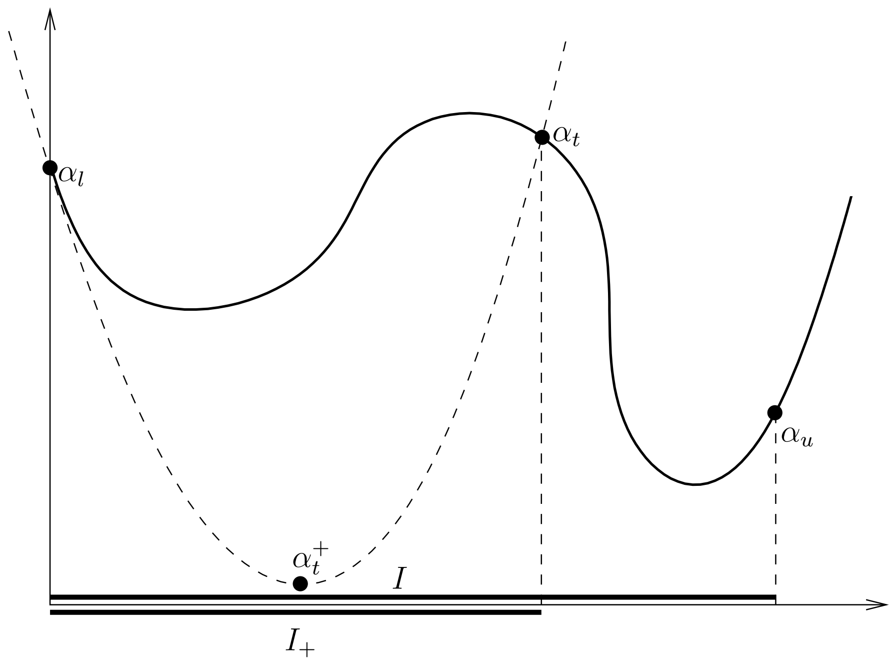


Figure 27: Interpolation step of Fletcher's algorithm.

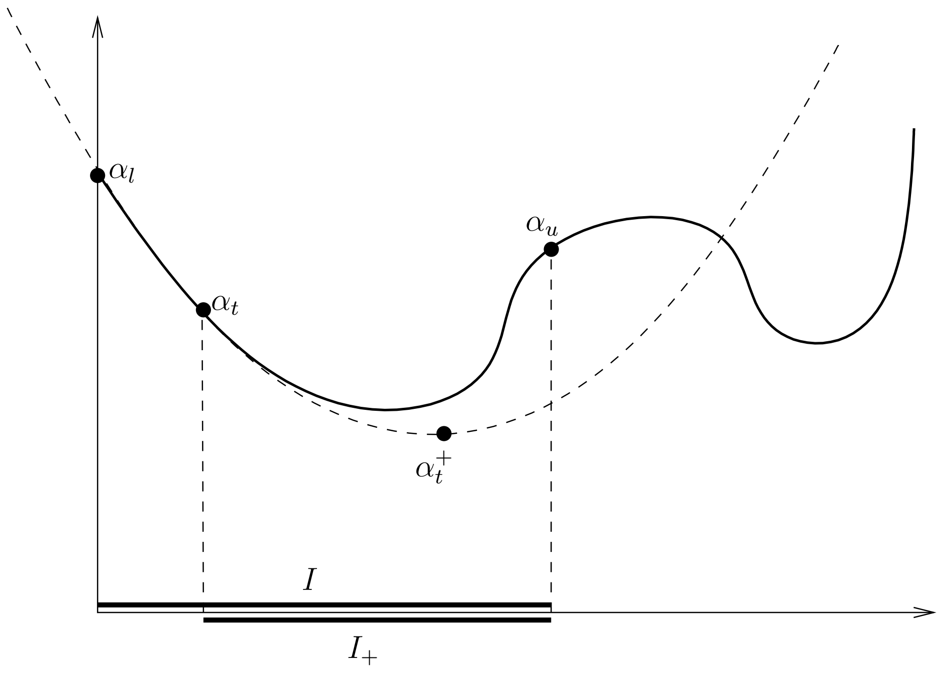


Figure 28: Extrapolation step of Fletcher's algorithm.

A.3 Fletcher's initial step length selection

Fletcher [Fle80, p. 28] suggested applying (2.3.26) for specifying initial line search step lengths. As in section 2.3.2, the initial interval is $I_0 = [0, \infty]$. Equating α_0 and the minimizer of a quadratic polynomial that interpolates $\phi(\alpha_l)$, $\phi'(\alpha_l)$ and $\phi(\alpha_0)$ yields

$$\alpha_0 = \alpha_l + \frac{(\alpha_0 - \alpha_l)^2 \phi'(\alpha_l)}{2[\phi(\alpha_l) - \phi(\alpha_0) + (\alpha_0 - \alpha_l)\phi'(\alpha_l)]}. \tag{A.3.1}$$

The problem with solving α_0 from this formula is that $\phi(\alpha_0)$ depends on α_0 which is to be solved. This problem is circumvented by assuming that

$$\phi(\alpha_l) - \phi(\alpha_0) = \Delta\phi,$$

where

$$\Delta\phi \equiv f(\mathbf{x}_{k-1}) - f(\mathbf{x}_k)$$

is obtained by using $\mathbf{x}_{k-1}$ from the previous iteration of (2.3.1). By this approximation and the choice $\alpha_l = 0$, we can write equation (A.3.1) as

$$\alpha_0 = \frac{\alpha_0^2 \phi'(\alpha_l)}{2[\Delta\phi + \alpha_0 \phi'(0)]}.$$

Rearranging terms in the above equation yields an estimate for the initial step length, that is

$$\alpha_0 = -\frac{2\Delta\phi}{\phi'(0)}.$$

This method for choosing the initial step length is illustrated in Figure 29.

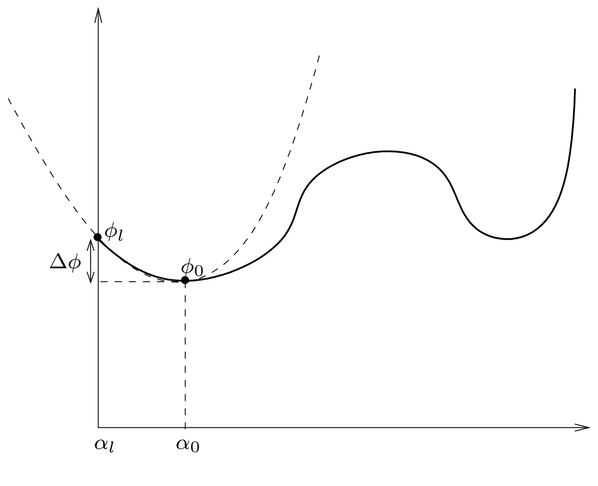


Figure 29: Fletcher's initial step length selection.

A.4 The modified $\mathbf{LDL}^T$ factorization

We begin our analysis with briefly describing the standard $\mathbf{LDL}^T$ factorization. It can be shown that any symmetric and positive definite matrix $\mathbf{A}$ can be written in the form [3]

$$\mathbf{A} = \mathbf{LDL}^T, \tag{A.4.1}$$

where $\mathbf{L}$ is a lower-triangular matrix with unit diagonal elements and $\mathbf{D}$ is a diagonal matrix with positive elements. This also implies that the diagonal elements of $\mathbf{A}$ are positive. We denote the elements of the matrices $\mathbf{A}$, $\mathbf{L}$ and $\mathbf{D}$ as a_{ij}, l_{ij} and d_{ij}, respectively. Equating the matrix elements in (A.4.1) yields

$$d_i = a_{ii} - \sum_{j=1}^{i-1} l_{ij} c_{ij}, \tag{A.4.2}$$

$$c_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{jk} c_{ik}, \quad i = j+1, \ldots, n, \tag{A.4.3}$$

where the shorthand notation $c_{ij} = l_{ij} d_j$ is used.

The modified $\mathbf{LDL}^T$ factorization by Gill and Murray [GM74] produces the $\mathbf{LDL}^T$ factorization of a modified, positive definite matrix $\hat{\mathbf{A}}$ such that

$$\hat{\mathbf{A}} = \mathbf{L}\hat{\mathbf{D}}\mathbf{L}^T = \mathbf{L}(\mathbf{D} + \mathbf{E})\mathbf{L}^T = \mathbf{A} + \mathbf{E},$$

where $\mathbf{E}$ is a non-negative diagonal matrix which is zero when $\mathbf{A}$ is already positive definite. The requirement of positive elements in the matrix $\hat{\mathbf{D}}$ implies positive definiteness of the resulting matrix $\mathbf{A} + \mathbf{E}$, since in this case

$$\mathbf{x}^T(\mathbf{A} + \mathbf{E})\mathbf{x} = \mathbf{x}^T\mathbf{L}\hat{\mathbf{D}}\mathbf{L}^T\mathbf{x} = (\mathbf{L}^T\mathbf{x})^T\hat{\mathbf{D}}\mathbf{L}^T\mathbf{x} > 0.$$

However, requiring positivity of the elements of $\hat{\mathbf{D}}$ is not alone sufficient to constitute a numerically stable algorithm. In particular, if $\mathbf{D}$ has negative elements, the elements $l_{ij}c_{ij}$ may become arbitrarily large. If the unmodified matrix $\mathbf{D}$ has only positive elements, equation (A.4.2) imposes an upper bound to the elements $l_{ij}c_{ij} = l_{ij}^2 d_j$, and thus to the elements of $\mathbf{LD}^{\frac{1}{2}}$ via the diagonal elements of the matrix $\mathbf{A}$. However, this is not the case if negative elements of $\mathbf{D}$ are simply set to a positive number. Thus, Gill and Murray suggest imposing an additional requirement

$$|l_{ij}c_{ij}| = |l_{ij}^2 d_j| \leq \beta^2, \quad \begin{array}{l} i = 1, \ldots, n, \\ j = 1, \ldots, i-1, \end{array} \tag{A.4.4}$$

[3]The Cholesky factorization $\mathbf{A} = \mathbf{LL}^T$ with $\mathbf{L} \equiv \mathbf{LD}^{1/2}$ is a special case of the $\mathbf{LDL}^T$ factorization.

where β is a positive constant. This guarantees a numerically more stable $\mathbf{LDL}^T$ factorization.

In order to satisfy the requirement of positivity of the diagonal elements of $\hat{\mathbf{D}}$ and condition (A.4.4), equation (A.4.2) is modified such that

$$d_i = \max\{|c_{ii}|, \theta_i^2/\beta^2, \delta\}, \tag{A.4.5}$$

where

$$\theta_i = \max\{|c_{ji}| \mid j = i+1, \dots, n\}.$$

The threshold δ is introduced in order to improve numerical stability. Several different values have been suggested in the literature [GM74], [NW99], [GMW81]. To the experience of the author of this thesis, the value ϵ_m or its multiple yields good results.

Modification (A.4.5) guarantees that the required conditions are satisfied. In the first case, $d_i = |c_{ii}|$ and $d_i \geq \theta_i^2/\beta^2$. Hence, by the definition of θ_i, we have

$$|l_{ij}c_j| = |l_{ij}^2 d_j| = \tfrac{|c_{ij}|^2}{d_j} \leq \tfrac{|c_{ij}|^2\beta^2}{\theta_j^2} \leq \beta^2, \quad j = 1, \dots, i-1.$$

In the second case, $d_i = \theta_i^2/\beta^2$. Then

$$\begin{aligned} d_i = \tfrac{\theta_i^2}{\beta^2} &\iff d_i = \tfrac{\max\{|l_{ji}d_i|^2\}}{\beta^2} \\ &\iff \beta^2 = \tfrac{\max\{|l_{ji}d_i|^2\}}{d_i} \qquad j = i+1, \dots, n. \\ &\iff \beta^2 = \max\{|l_{ji}^2 d_i|\}, \end{aligned}$$

The above equality shows that d_j is in this case set to the value for which the greatest value of $|l_{ji}^2 d_i| = |l_{ji}c_i|$, where $j = i+1, \dots, n$, is exactly equal to β^2. Condition (A.4.4) also holds in the third case, if δ is set to a sufficiently small value.

The choice of β is critical for the numerical stability of this factorization. By establishing an approximate upper bound for the norm of $\mathbf{E}$, Gill and Murray suggest using the value

$$\beta^2 = \xi/\sqrt{n^2-1}$$

that minimizes this upper bound [GM74, Thm. 2.2.1, eq. (11)]. The value of ξ is given by

$$\xi = \max\{|a_{ij}| \mid i \neq j,\ i,j = 1, \dots, n\}.$$

A lower bound for β also needs to be imposed in order to avoid unnecessary modifications if $\mathbf{A}$ is already positive definite. Equation (A.4.2) can be rewritten as

$$a_{ii} = \textstyle\sum_{j=1}^{i} l_{ij}^2 d_j, \quad l_{ii} = 1,$$

which implies for a positive definite matrix $\mathbf{A}$ that $l_{ij}^2 d_{ij} = l_{ij}c_{ij} \leq a_{ii}$ for all $j = 1, \ldots, i$, since a_{ii} is positive. Consequently,

$$l_{ij}c_{ij} \leq a_{ii} \leq \max\{|a_{ii}| \mid i = 1, \ldots, n\}.$$

Thus, Gill and Murray give the lower bound

$$\gamma = \max\{|a_{ii}| \mid i = 1, \ldots, n\}$$

which guarantees that already positive definite matrix is not modified. The final value of β is given by

$$\beta^2 = \max\{\gamma, \frac{\xi}{\sqrt{n^2-1}}, \epsilon_m\},$$

where the lower bound of machine epsilon ϵ_m is for improving numerical stability.

A.5 Finite-difference approximations

The components of the forward difference gradient approximation are given by

$$[\nabla f(x)]_i \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}, \tag{A.5.1}$$

and the components of the central difference approximation are given by

$$[\nabla f(x)]_i \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h}, \tag{A.5.2}$$

where h is the user-supplied step size. In general, accuracy of gradient approximations is critical. The line search methods discussed in this thesis and stopping criteria that use gradient information are particularly sensitive to inaccurate derivatives. Thus, using at least central difference gradients is strongly recommended. [4]

Since evaluating the Hessian matrix has computational complexity of $\mathcal{O}(n^2)$, using forward difference formulas is recommended for performance reasons. Two different methods have been suggested in the literature [DS83, p. 103-104]. The first method applies the forward difference formula to symbolically computed gradient. The resulting matrix is given by

$$[\mathbf{H}_f(\mathbf{x})]_{ij} \approx \tfrac{\partial f}{\partial x_i}(\mathbf{x} + h\mathbf{e}_j) - \tfrac{\partial f}{\partial x_i}(\mathbf{x}), \quad i, j = 1, \ldots, n. \tag{A.5.3}$$

[4]GSL implements more accurate forward and central difference approximations that use function values evaluated at four points. These approximations also use adaptive step size selection.

This operation does not in general produce a symmetric matrix, and therefore the resulting matrix is symmetrized by setting

$$\mathbf{H}_f(\mathbf{x}) \approx \frac{\tilde{\mathbf{H}}_f(\mathbf{x}) + \tilde{\mathbf{H}}_f(\mathbf{x})^T}{2},$$

where $\tilde{\mathbf{H}}_f(\mathbf{x})$ is obtained by applying (A.5.3). The second method applies the forward difference formula twice to obtain the second order derivatives $\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x})$. This is done by approximating

$$[\mathbf{H}_f(\mathbf{x})]_{ij} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i + h\mathbf{e}_j) - f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} + h\mathbf{e}_j) + f(\mathbf{x})}{h^2}. \quad \text{(A.5.4)}$$

Computational complexities of finite-difference methods are dominated by function and derivative evaluation counts. They are listed in Table 15. The optimal step lengths and the errors with these values are listed in Table 16. The reader is referred to [NW99, p. 167-169, 173-174] for their derivations.

Method	Function evaluations/ gradient	Gradient eval./ Hessian (symbolic gradient)	Function eval./ Hessian (f.-d. gradient)
forward difference	$n+1$	$n+1$	$\frac{1}{2}n(n+1)+n+1$
central difference	$2n$	not implemented	not implemented

Table 15: Number of function and derivative evaluations per gradient and Hessian, finite-difference derivatives.

Method	Optimal step size	Gradient error	Hessian error
forward difference	$\sqrt{\epsilon_m}$	$\sqrt{\epsilon_m}$	$\sqrt[4]{\epsilon_m}$
central difference	$\sqrt[3]{\epsilon_m}$	$\sqrt[3]{\epsilon_m^2}$	$\sqrt[9]{\epsilon_m^4}$

Table 16: Optimal step lengths for finite-difference approximations and their corresponding error estimates (ϵ_m denotes the machine epsilon).