A New Interpolation Approach for Linearly Constrained Convex Optimization

Thesis by

M. Sc. Francisco Javier Franco Espinoza

Submitted in Partial Fulfillment of the Requirements for the degree of Masters of Science

King Abdullah University of Science and Technology
Mathematical and Computer Science and Engineering Division
Applied Mathematics and Computational Science

Thuwal, Makkah Province, Kingdom of Saudi Arabia
August, 2012
EXAMINATION COMMITTEE APPROVALS FORM

The thesis of Francisco Javier Franco Espinoza is approved by the examination committee.

Committee Chairperson: Dr. Alyn Rockwood.
Committee Member: Dr. Xiangliang Zhang.
Committee Member: Dr. George Turkiyyah.
ABSTRACT

A New Interpolation Approach for Linearly Constrained Convex Optimization

Francisco Javier Franco Espinoza

In this thesis we propose a new class of Linearly Constrained Convex Optimization methods based on the use of a generalization of Shepard’s interpolation formula. We prove the properties of the surface such as the interpolation property at the boundary of the feasible region and the convergence of the gradient to the null space of the constraints at the boundary. We explore several descent techniques such as steepest descent, two quasi-Newton methods and the Newton’s method. Moreover, we implement in the Matlab language several versions of the method, particularly for the case of Quadratic Programming with bounded variables. Finally, we carry out performance tests against Matlab Optimization Toolbox methods for convex optimization and implementations of the standard log-barrier and active-set methods. We conclude that the steepest descent technique seems to be the best choice so far for our method and that it is competitive with other standard methods both in performance and empirical growth order.
I would like to express my gratitude to my advisor Dr. Alyn Rockwood for giving me the opportunity to work with him. His continual support and trust encouraged me to keep working hard in difficult parts of the project. It is truly an honor to have been able to work together with such an accomplished researcher and a great person as well.

I owe sincere thankfulness to Dr. George Turkiyyah, not only for his invaluable advise throughout the project, but because it was him that introduced me to the exciting field of optimization. I am grateful as well to Dr. Xiangliang Zhang for her encouragement throughout my studies in KAUST, where I had the honor of attending two courses taught by her.

I would also like to thank my fellow student Han Liu, for her important early work in this project, Mr. Dick Sowar, who also worked on the project and provided very valuable input, and Eliana Zarzar who showed initiative in joining the project and provided very valuable work.

Finally, I am thankful to my siblings Rafael and Paulina for their support and inspiration, and I would like to dedicate this work to my parents, who have always taught me the value of education. It is because of them that I am here and it is because of them that I do my best.
# TABLE OF CONTENTS

**Acknowledgments**  
5

**List of Abbreviations**  
9

**List of Symbols**  
10

**List of Illustrations**  
11

**List of Tables**  
14

## I Introduction  
15

- **I.1 Convex Optimization**  
15
- **I.2 Motivation and Overview**  
17

## II Theory of Convex Optimization  
18

- **II.1 Linear Programming**  
18
- **II.2 Quadratic Programming**  
20
- **II.3 Duality**  
21
- **II.4 Methods of Inequality Constrained Convex Optimization**  
23
  - **II.4.1 Interior-Point Methods**  
24
  - **II.4.2 Active Set Methods**  
27
  - **II.4.3 Sequential Quadratic Programming**  
29
III An Interpolation Function for Linearly Constrained Convex Programs 31

III.1 Shepard’s Formula ............................................. 31

III.1.1 Properties of Shepard’s Formula ............................. 32

III.2 Generalization of Shepard’s Formula for Linear Constraints in a Convex Optimization Context 34

III.2.1 Interpolation Property of the Surface ...................... 37

III.2.2 Gradient of the Interpolation Surface ...................... 39

III.3 Potential Use of the Interpolation Surface in an Optimization Context 46

IV Optimization Algorithm 51

IV.1 Iteration Paradigms: Line Search and Trust Region .......... 52

IV.1.1 Line Search .................................................. 52

IV.1.2 Trust Region ................................................ 54

IV.2 Descent Methods .............................................. 56

IV.2.1 Steepest Descent ............................................ 56

IV.2.2 Quasi-Newton Methods ..................................... 56

IV.2.3 Newton’s Method ............................................ 64

V Implementation and Tests 69

V.1 Phase I Algorithm .............................................. 70

V.2 Interpolation Surface Function .................................. 71

V.3 Implementation of the Optimization Algorithms .............. 72

V.3.1 Algorithm 2: Steepest Descent ............................... 73

V.3.2 Algorithm 3: BFGS Method .................................. 75

V.3.3 Algorithm 4: Damped BFGS Method ....................... 77

V.3.4 Algorithm 5: Newton’s Method .............................. 78

V.4 Solving a Sequence of Problems in a Subdimensional Space . 82
List of Abbreviations

BFGS  Broyden-Fletcher-Goldfarb-Shanno
KKT   Karush-Kuhn-Tucker
LP    Linear Program
QCQP  Quadratically Constrained Quadratic Program
QP    Quadratic Program
SOCP  Second Order Cone Programming
SQP   Sequential Quadratic Programming
List of Symbols

\( \mathbb{R} \) Set of real numbers.

\( \mathbb{R}^k \) Set of real vectors of size \( k \).

\( \mathbb{R}^{k \times l} \) Set of real matrices of size \( k \times l \).

\( \mathbb{S}^k \) Set of symmetric matrices of size \( k \times k \).

\( \mathbb{S}^+_k \) Set of positive semidefinite symmetric matrices of size \( k \times k \).

\( \mathbb{S}^{++}_k \) Set of positive definite symmetric matrices of size \( k \times k \).

\( \mathcal{C}^k \) Set of functions whose derivatives up to the \( k \)th degree exist and are continuous.

\( \mathcal{J}(f(x)) \) Jacobian matrix of the function \( f(x) \).

\( n \) Throughout the text, used to denote the number of dimensions of the optimization problem.

\( \mathcal{L}(a, \lambda, \nu) \) Lagrangian function of the primal variable \( x \) and the dual variables \( \lambda \) and \( \nu \).

\( m \) Throughout the text, used to denote the number of constraints of the optimization problem.

\( \lambda \) Throughout the text, used to denote the vector of dual variables or Lagrange multipliers corresponding to inequality constraints.
Throughout the text, used to denote the dual variables or Lagrange multipliers corresponding to equality constraints.

$\mathcal{A}(x)$ Active set of $x$. Represents the set of active constraints at the point $x$.

$W_k$ Working set of at the $k$th iteration. In active-set methods, represents the guess of the optimal active set at the $k$th iteration.
List of Illustrations

<table>
<thead>
<tr>
<th>Illustration</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>III.1</td>
<td>Influence of $\mu$ in the properties of the interpolation function.</td>
<td>34</td>
</tr>
<tr>
<td>III.2</td>
<td>An interpolation point $x$ is projected to the constraint planes. The projection points will then serve as data points for the interpolation.</td>
<td>36</td>
</tr>
<tr>
<td>III.3</td>
<td>Example of a quadratic program. The red lines represent the projection of the boundary on the objective function. The interpolation function will go through these lines.</td>
<td>46</td>
</tr>
<tr>
<td>III.4</td>
<td>The gradient field of the interpolation function leads to the optimum.</td>
<td>47</td>
</tr>
<tr>
<td>III.5</td>
<td>Both images depict the interpolation surface for the example problem. The surface passes exactly through the objective function at the boundary.</td>
<td>48</td>
</tr>
<tr>
<td>III.6</td>
<td>1D toy problem exemplifying the stationary point effect.</td>
<td>49</td>
</tr>
<tr>
<td>III.7</td>
<td>In the box-constrained case, only boundary points are used for the interpolation function.</td>
<td>50</td>
</tr>
<tr>
<td>IV.1</td>
<td>As the iterates approach the boundary, the trust region radius becomes smaller, thus hurting the convergence speed.</td>
<td>55</td>
</tr>
<tr>
<td>IV.2</td>
<td>The interpolation function is not convex but quasilinear. It has a region with positive curvature but also a region with negative curvature.</td>
<td>60</td>
</tr>
<tr>
<td>V.1</td>
<td>Example solutions of 2-D problems with steepest descent algorithm.</td>
<td>74</td>
</tr>
</tbody>
</table>
V.2 Convergence plot of the 2-D example problems solved with steepest descent. 75

V.3 Example solutions of 2-D problems with the BFGS algorithm. 76

V.4 Convergence plot of the 2-D example problems solved with the BFGS algorithm. 77

V.5 Example solutions of 2-D problems with the Damped BFGS method. 77

V.6 Convergence plot of the 2-D example problems solved with the Damped BFGS method. 78

V.7 Example solutions of 2-D problems with Newton’s method. 79

V.8 Convergence plot of the 2-D example problems solved with Newton’s method. 79

V.9 Close up of Figure V.7b. 80

V.10 Convergence plot of a 30-D example problems solved with the steepest descent method. 81

V.11 Convergence plot of the 30D example problems solved with Newton’s method. 82

V.12 Average running times of all five interpolation surface algorithms. The simple steepest descent seems to be the most effective choice. 89

V.13 Average number of iterations performed by the interpolation surface algorithms. Quasi-Newton and Newton’s methods do not improve the number of iterations performed by the steepest descent implementations. 90

V.14 Average running times of several popular methods implementations compared with the steepest descent interpolation method. Our interpolation method with steepest descent seems to be at least competitive with some of the methods. 91

V.15 Zoom in on Figure V.14. 92
## List of Tables

<table>
<thead>
<tr>
<th>V.1</th>
<th>Number of failed cases per algorithm out of the first 920 problems in the set (Dimensions 2-D to 600-D). For the mean precision calculation, the failed cases have been left out.</th>
<th>87</th>
</tr>
</thead>
<tbody>
<tr>
<td>V.2</td>
<td>Average empirical complexity of the tested methods measured at dimensions 400-D to 1000-D, except for the subdimensional methods where it was measured only at the 400-D and 600-D due to computing time restrictions (*). The last column shows the growth order averaged over these two problem sizes.</td>
<td>94</td>
</tr>
</tbody>
</table>
Chapter I

Introduction

I.1 Convex Optimization

Optimization is becoming ubiquitous in different areas of science, engineering, economics, management, finance and statistics for example. Engineers optimize designs in terms of material usage, shape measurements or energy efficiency; managers use it to create schedules for the employees and machines and to design optimal transport routes; investors use it to create high-return and low risk portfolios; scientists use it to obtain parameters from models and analyze phenomena.

Although optimization has been studied for many years, it has gained a great deal of attention during the last century first of all due to the invention of the computer, which enabled the solution of problems that were not possible to solve before, and then, due to the revolution that followed the introduction of the Simplex method for linear programs (LPs), and finally the introduction of efficient interior-point algorithms in the 1980s.

George B. Dantzig [1] [2] first proposed the Simplex method in 1947 which led to a new era in linear programming. Interior-point methods were in use by the 1960s [3] [4], but their use did not became widespread until Narendra Karmarkar introduced
his polynomial time algorithm for linear programming in the year of 1984 [5]. Years later, efficient interior-point methods were developed for non linear problems as well.

A natural division of the optimization field can be seen in linear and non linear solution methods, however, this traditional view has been superseded in the last decades by the convex and non convex optimization division since convex optimization problems have more properties in common such as the guarantee that any local optimum is also a global one. Also, convex optimization is much more mature than general non linear optimization since there are still no methods to solve general problems of the latter category efficiently and reliably. Despite the maturity of the convex optimization field, it is still a very active area of research.

A convex problem has the form:

\[
\begin{align*}
\text{minimize} & \quad f_0(x) \\
\text{subject to} & \quad f_i(x) \leq 0 \quad i = 1, \ldots, m \\
& \quad h_i(x) = 0 \quad i = 1, \ldots, p
\end{align*}
\]  

where \( x \in \mathbb{R}^n \) is called the optimization variable, \( f_0 : \mathbb{R}^n \to \mathbb{R} \) is the objective function, \( f_i : \mathbb{R}^n \to \mathbb{R} \) are the inequality constraint functions and \( h_i : \mathbb{R}^n \to \mathbb{R} \) are the equality constraint functions. The objective and the inequality constraint must be convex functions while the equality constraints must be linear functions.

At the same time, a convex function is one such that:

\[
f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y)
\]  

for \( 0 \leq \theta \leq 1 \) and where the domain of \( f \) is a convex set. The negative of a convex function is called a concave function.

Convex optimization problems are important because they arise in areas such as
portfolio optimization [6], automatic control design [7] [8] [9], signal processing and communication [10] [11], electronic circuit design [12] [13], machine learning and data mining [14] [15], to name a few.

I.2 Motivation and Overview

As stated before, convex optimization is still a very active field of research and although it is not straightforward to create a new algorithm that significantly improves what has been done in the last century, proposing a novel approach can be beneficial to the field. It encourages discussion. For example, interior-point methods had been known for a long time before Karmakar proposed his polynomial time algorithm, which revolutionized the field of Linear Programming. Moreover, people later realized that the same paradigm could be applied to other types of convex programs, enabling the solution of new types of problems.

In this thesis, we propose a novel method for convex optimization problems with linear inequality constraints. It is based on the substitution of the original objective function by an interpolation surface whose gradient field leads us directly to the optimum. We first develop the algorithm for the case where the only constraints are bounds on the variables.

In Chapter 2, gives a discussion of the most popular convex optimization methods. In Chapter 3 we introduce the theory behind the interpolation surface based on a generalization of the Shepard’s method. This is the key part of our optimization method. Chapter 4 includes the description of the descent methods used. Chapter 5 discusses the numerical results and Chapter 6 presents the conclusions and future work directions. Given that the project relates to very different topics, we decided to distribute the literature references related to each topic within Chapters 2, 3 and 4.
Chapter II

Theory of Convex Optimization

The importance that convex optimization has gained in the last decades is in great part due to the fact that convex problems are so common in practice. Many problems that are apparently non convex can be reformulated as convex optimization problems. Furthermore, using convex relaxations of non convex problems is a common way of obtaining approximate solutions or bounds on the optima and are at the core of many global optimization techniques.

II.1 Linear Programming

Inside convex optimization, Linear Programming is not only the most popular, but the most studied and mature field. The standard form of the Linear Programming problem is the following:

$$\begin{align*}
\text{minimize} & \quad c^\top x \\
\text{subject to} & \quad Ax = b \\
& \quad x \geq 0
\end{align*}$$

(II.1)
The inequality form is:

\[
\text{minimize} \quad c^\top x \\
\text{subject to} \quad Ax \leq b
\]  

(II.2)

where \( c \in \mathbb{R}^n \) is the cost vector, \( A \in \mathbb{R}^{m \times n} \) is the matrix whose rows are the normals to the inequality constraint planes and \( b \in \mathbb{R}^m \) is the vector whose elements are the bounds to the linear constraints. Any LP can be transformed to any of these formulations.

Applications of Linear Programming range from resource allocation, to network optimization, classification problems in machine learning and linear surface fitting, to name a few examples. The introduction of the well-known Simplex method by Dantzig [2] after the Second World War coupled with the development of computers generated a lot of attention in Linear Programming and its applications. The Simplex algorithm finds a vertex on the boundary of the feasible polytope as an initial point and then searches for an adjacent vertices where the objective value is lower and repeats until no vertex with a lower value can be found.

It has been proven that the complexity of the Simplex method can have exponential growth on certain families of problems as noted by Klee and Minty [16] and Goldfarb [17], among others. However, in practical problems, the method is much more efficient than its worst case complexity and this is the reason why it is still used today.

Nonetheless, there was a quest to find a polynomial time algorithm for Linear Programming and in 1979, Leonid Khachiyan [18] proved that the ellipsoid algorithm could solve linear programs in polynomial-time. In spite of this desirable theoretical property, the ellipsoid algorithm underperformed against the Simplex method in
practice.

The next big revolution in Linear Programming came in 1984, when Narendra Karmarkar [5] announced the discovery of a polynomial time interior-point algorithm that was supposed to outperform the Simplex method. Indeed, the algorithm was very efficient in practice and soon enough similar algorithms were devised for other types of convex problems. Interior-point methods were actually not new by then. The log barrier method was in use since the 1960s and it was later demonstrated that there was a connection between the methods [19].

II.2 Quadratic Programming

Another important area of convex optimization is *quadratic programming*. A quadratic program has the following form:

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2} x^\top P x + q^\top x + r \\
\text{subject to} & \quad Ax \leq b \\
& \quad C x = d
\end{align*}
\]

where \( P \in \mathbb{S}_+^n \), \( q \in \mathbb{R}^n \), \( r \in \mathbb{R} \), \( A \in \mathbb{R}^{m \times n} \), \( b \in \mathbb{R}^m \), \( C \in \mathbb{R}^{p \times n} \), and \( d \in \mathbb{R}^p \). This is a convex quadratic function subject to linear inequalities and linear equalities. The importance of QP lies not only in its applications, which include constrained least-squares and the Markowitz portfolio optimization [6], but also because some methods for solving other families of convex problems involve the solution of a series of quadratic programs as in the case of Sequential Quadratic Programming (SQP).

Other instances of convex optimization include quadratically constrained quadratic programs (QCQP), second order cone programming (SOCP) and the convex form of
geometric programming [20].

II.3 Duality

The Lagrangian of the problem (I.1) is defined as:

\[ \mathcal{L}(x, \lambda, \nu) = f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{i=1}^{p} \nu_i h_i(x) \]  

(II.4)

where \( \lambda_i \) are the Lagrange multipliers associated with the \( i \)th inequality constraint and \( \nu_i \) are the Lagrange multipliers associated with the \( i \)th equality constraints. The vectors \( \lambda \) and \( \nu \) are also called the dual variables. In turn \( x \) is called the primal variable. This definition is also valid when the objective function and the constraints are not convex.

The Lagrange dual function is defined as the infimum of the Lagrangian function (II.4):

\[ g(\lambda, \nu) = \inf_{x \in \mathcal{D}} \mathcal{L}(x, \lambda, \nu) \]  

(II.5)

where \( \mathcal{D} \) is the domain of (I.1). This function is always convex, even when the original problem is not. Since it is the point-wise infimum of the Lagrangian function, it provides a lower bound on the optimum value of the original problem:

\[ g(\lambda, \nu) \leq p^* \]  

(II.6)

for \( \lambda \geq 0 \), and where \( f_0(x^*) = p^* \) and \( x^* \) is the optimal point of problem (I.1). We
define the pair \((\lambda^*, \nu^*)\) as the *dual optimal*, and \(d^*\) as the optimal value of the dual problem:

\[
\text{maximize } g(\lambda, \nu) \quad \text{(II.7)}
\]

subject to \(\lambda \geq 0\)

Then we have that:

\[d^* \leq p^* \quad \text{(II.8)}\]

This property is called *weak duality*. If the equality holds we say that there is *strong duality*.

This is an important property because convex problems present strong duality in most of the cases. Particularly when the convex problem complies with *Slater’s condition*, which states that there must be an \(x\) such that:

\[f_i(x) < 0, \quad i = 1, ..., m \quad \text{(II.9)}\]

which means that the inequality constraints are *strictly feasible*. Otherwise, if inequality constraints exist for which \(x\) is feasible but not strictly feasible, i.e. \(f_j(x) = 0\), then they must be linear.

We are now ready to state the necessary conditions for optimality, the famous Karush-Kuhn-Tucker (KKT) conditions, which for convex problems are also sufficient.
\[ \nabla f_0(x^*) + \sum_{i=1}^{m} \lambda_i^* \nabla f_i(x^*) + \sum_{i=1}^{p} \nu_i^* \nabla h_i(x^*) = 0 \] (II.10)
\[ \lambda_i^* f_i(x^*) = 0, \quad i = 1, \ldots, m \] (II.11)
\[ \lambda_i^* \geq 0, \quad i = 1, \ldots, m \] (II.12)
\[ f_i(x^*) \leq 0, \quad i = 1, \ldots, m \] (II.13)
\[ h_i(x^*) = 0, \quad i = 1, \ldots, p \] (II.14)

Condition (II.10) says that the gradient of the Lagrangian function must be zero, this is because \( x^* \) is also a minimizer of \( \mathcal{L}(x, \lambda^*, \nu^*) \). The second condition (II.11) is called the complementarity condition, it basically says that either \( \lambda_i^* = 0 \) or \( f_i(x^*) = 0 \). Condition (II.12) comes from the fact that \( g(\lambda, \nu) \) is only bounded when \( \lambda \geq 0 \). Finally, the last couple of constraints are the primal constraints for the original problem.

The importance of the KKT conditions comes from the fact that they are the base for many algorithms for convex optimization.

### II.4 Methods of Inequality Constrained Convex Optimization

In this section we give a brief overview of the ideas behind the most popular convex optimization algorithms. These descriptions aim to provide the flavor of the methods and not the description of state of the art algorithms, which is beyond the scope of this thesis. However, we provide references to publications with more insightful descriptions.
II.4.1 Interior-Point Methods

Logarithmic Barrier Method

We restate the optimization problem making the linear equalities explicit:

\[
\begin{align*}
\text{minimize} & \quad f_0(x) \\
\text{subject to} & \quad f_i(x) \leq 0 \quad i = 1, \ldots, m \\
& \quad Ax = b
\end{align*}
\]

where \( A \in \mathbb{R}^{p \times n} \) and \( b \in \mathbb{R}^p \). The basic idea behind this method is to substitute the original problem by one with no inequality constraints, i.e. by creating a "barrier function" with the inequality constraint functions and adding it to the objective. For the logarithmic barrier method, such a function has the following form:

\[
\phi(x) = -\sum_{i=1}^m \log(-f_i(x))
\]  

The optimization problem now becomes:

\[
\begin{align*}
\text{minimize} & \quad tf_0(x) + \phi(x) \\
\text{subject to} & \quad Ax = b
\end{align*}
\]

for \( t > 0 \), where \( t \) is a factor that decreases the influence of \( \phi(x) \) away from the boundary as its value increases. This method can be seen as a perturbation on the KKT conditions (II.10) to (II.14). If we write the KKT conditions for problem (II.17), we have
\[ \nabla f_0(x) + \sum_{i=1}^{m} \lambda_i \nabla f_i(x) + A^\top \nu = 0 \] (II.18)

\[ \lambda_i f_i(x) = -1/t, \quad i = 1, \ldots, m \]

\[ \lambda_i \geq 0, \quad i = 1, \ldots, m \]

\[ f_i(x) \leq 0, \quad i = 1, \ldots, m \]

\[ Ax = b, \quad i = 1, \ldots, p \]

where it is clear that the complementarity slackness condition is perturbed. We also note that as \( t \to \infty \), the KKT conditions become equivalent to the original ones so the optimal point \( x^* \) will be the same. In the logarithmic barrier method, typically one solves the set of equations (II.18) by the Newton method by first eliminating the \( \lambda_i \) variable with the second equation and then substituting the first two terms of the Lagrangian by the first degree Taylor approximation. The resulting system of equations can be solved for a suitable value of \( t \). However, in practice we solve a sequence of problems (II.17) for a series of increasing values of \( t \) until reaching the desired tolerance.

**Primal-Dual Interior-Point Methods**

This type of interior-point methods are very similar to the logarithmic barrier method, they are also based on the solution of the modified KKT conditions (II.18) but there are important differences:

- The resulting search directions are different
- The primal-dual interior-point methods include the dual search directions
- The logarithmic barrier method solves a sequence of subproblems, thus it implies one outer and one inner loop. In the primal-dual interior-point methods there
is only one main loop.

We begin by rearranging the modified KKT conditions (II.18):

$$r_t(x, \lambda, \nu) = \begin{bmatrix} \nabla f_0(x) + J(f(x))^\top \lambda + A^\top \nu \\ -\text{diag}(\lambda) f(x) - (1/t) e \\ A x - b \end{bmatrix} = 0 \quad \text{(II.19)}$$

where $J(f(x))$ is the Jacobian matrix of the vector of inequality constraints $f(x)$, $\text{diag}(\lambda)$ is the matrix whose diagonal elements are the Lagrange multipliers $\lambda_i$ and $e$ is a vector of in $\mathbb{R}^m$ whose elements are all 1. The primal-dual interior-point method obtains the primal-dual search directions $(\Delta x, \Delta \lambda, \Delta \nu)$ by solving the linear system:

$$J(r_t(x, \lambda, \nu)) \begin{bmatrix} \Delta x \\ \Delta \lambda \\ \Delta \nu \end{bmatrix} = -r_t(x, \lambda, \nu) \quad \text{(II.20)}$$

In each iteration we determine the value of $t$, compute the primal-dual search direction, perform a line search that makes sure that $f_i(x) < 0$ and $\lambda_i > 0$ for all $i$. We stop after the desired tolerance is met.

According to Boyd [20], primal-dual interior-point methods are often times more efficient than logarithmic barrier methods for many types of convex problems. The main reason why primal-dual methods are more efficient is that they often show superlinear convergence. This is hard to obtain in barrier methods since the Hessian of the Newton system becomes increasingly ill-conditioned near the solution which causes problems in methods such as Quasi-Newton or Conjugate Gradient. The Newton method is not sensitive to ill-conditioning but it also suffers difficulties since the quadratic approximation is not precise in this case.
Frisch [3] originally proposed the logarithmic barrier method in 1955, but Fiacco and McCormick [4] later developed them in the 1960s. During this period it was mainly used for non linear convex programs, although their use slowly decreased due to the fact that the Hessian of the logarithmic function becomes ill-conditioned in the neighbourhood of the solution [21]. Nonetheless, after Karmarkar introduced his interior-point algorithm for linear programs, there was interest in applying the logarithmic barrier method to this area after a connection was made between the two algorithms by Gill et al. [22]. Hertog et al. [23] present an implementation of the classical logarithmic barrier method. In [24], Wright studies the convergence of the method. Polyak [25] proposed a modification on the classical method that improves the conditioning of the Hessian Matrix. Breitfeld [26] compares implementations of several logarithmic barrier methods. A couple of surveys on interior-point methods are given by Gould, Orban and Toint [27] and Forsgren, Gil and Wright [28].

II.4.2 Active Set Methods

The active-set $\mathcal{A}(x)$ of an optimization problem is the set of indices $i$ that correspond to the equality constraints $h_i(x) = 0$ and the inequality constraints for which equality holds $f_i(x) = 0$.

Active-set methods are a category of optimization methods that try to guess what the active-set at the solution is. The details of this guessing procedure varies a lot from algorithm to algorithm, but they all share the same essence. For example, a typical active-set algorithm for a quadratic program as described in the book by Nocedal and Wright [29] starts by computing a feasible starting point $x_0$. We define the working set of the $k$th iteration $\mathcal{W}_k$ to be the current guess of $\mathcal{A}(x^*)$ and we choose $\mathcal{W}_0$ to be $\mathcal{A}(x_0)$ or any subset of it. Next we find the primal step $p_k$, where:
that is, we find the step that minimizes $f_0(x_k + p)$ subject to the constraints of the active-set $A(x_k)$. There are two possibilities:

- If $p_k = 0$ we compute the Lagrange multipliers and terminate if $\lambda_i \geq 0 \; \forall i \in W_k$. Otherwise, we drop from the working set one of the constraints whose corresponding Lagrange multiplier is negative and continue.

- If $p_k \neq 0$ and $x_k + p_k$ is feasible, we set this point to be $x_{k+1}$ and continue with the next iteration. Otherwise, we find the maximum $a_k$ such that $x_k + a_k p_k$ remains feasible, we set this point to be $x_{k+1}$, we add the new active constraint to the working set and continue.

Active-set methods are normally not efficient in large-scale problems, where interior-point methods are preferred. For example, an active-set method like the one described has the disadvantage that it adds at most one constraint to the working set per iteration. Thus if the working set is empty at the initial point, and there are $n$ active constraints at the solution, we would need at least $n$ iterations to terminate the algorithm.

There are other active-set methods that are able to add more than one constraint at a time such as the gradient projection method which works on QPs where the only inequality constraints are bounds on the variables \[29\]. In \[30\] Gill et al. describe an active-set method for general linear constraints on which the Matlab’s Optimization Toolbox active-set algorithm for constrained optimization is based. Murtagh and
Saunders [31] developed an active-set type algorithm for solving problems with linear constraints and non linear objective.

### II.4.3 Sequential Quadratic Programming

Sequential Quadratic Programming (SQP) refers to a kind of optimization methods for non linear problems where one solves a sequence of quadratic problems that approximate the original one. SQP methods normally take an active-set approach and can be of two kinds:

- The quadratic sub problems are inequality-constrained and we compute a step and estimate the active-set \( A(x^*) \) at the same time in each iteration.

- We first estimate the active-set \( A(x^*) \) and then solve an equality-constrained quadratic sub problem where equality is forced on the constraints that belong to the active-set estimate.

In order to derive the QP that we will solve at each iteration we simply form the Newton system necessary to solve the KKT.

In the equality-constrained case we have the case:

\[
\text{minimize } f_0(x) \\
\text{subject to } h(x) = 0
\]  

(II.22)

where \( h(x) = 0 \) is the vector whose elements are the equality constraint functions. It can be shown that solving the KKT conditions of this problem by the Newton’s method is equivalent to solving the quadratic problem:
\begin{align*}
&\text{minimize } f_0(x_k) + \nabla f_0(x_k)^T p + \frac{1}{2} p^T \nabla^2_{xx} \mathcal{L}(x_k, \nu_k) p \text{ w.r.t. } p \quad (\text{II.23}) \\
&\text{subject to } \mathcal{J}(h(x_k))p + h(x_k) = 0
\end{align*}

which can be interpreted as minimizing a quadratic approximation to the Lagrangian subject to the linearisation of constraints $h(x) = 0$. In the same way, for the inequality constrained problem (II.24) we would linearize the inequality constraints to obtain the QP sub problem (II.25).

\begin{align*}
&\text{minimize } f_0(x) \quad (\text{II.24}) \\
&\text{subject to } f(x) \leq 0 \\
&\text{subject to } h(x) = 0
\end{align*}

\[ p_k = \arg \min_p f_0(x_k) + \nabla f_0(x_k)^T p + \frac{1}{2} p^T \nabla^2_{xx} \mathcal{L}(x_k, \nu_k) p \quad (\text{II.25}) \]

subject to \quad \mathcal{J}(h(x_k))p + h(x_k) = 0 \\
\quad \mathcal{J}(f(x_k))p + f(x_k) \leq 0

where we obtain $p_k$, the primal step such that $x_{k+1} = x_k + p_k$. If we use an active-set approach to solve the sub problem, we also obtain an estimated active-set $\mathcal{A}(x_k)$. Nocedal and Wright \cite{nocedal1999} provide a more detailed treatment of active-set Sequential Quadratic Programming methods. Gould et al. \cite{gould2000} and Boggs and Tolle \cite{boggs2000} provide surveys of SQP methods.
Chapter III

An Interpolation Function for Linearly Constrained Convex Programs

In their 2005 paper, Rockwood and Gao [34], proposed a method for creating multi-sided surface patches based on a generalization of Shepard’s interpolation formula. Rockwood took the inspiration from this earlier work to propose a convex optimization method based on the substitution of the original objective function with an interpolation function. We explore this idea in this thesis.

III.1 Shepard’s Formula

Shepard introduced his formula in 1968 [35] as a method for scattered data interpolation. Its simple and logical nature has gained it popularity throughout the years. Basically, it consists of a weighted average of the data points $f_i$, where the weights $w_i(x)$ depend on a distance measure $d_i(x)$ which normally is a negative power of the euclidean distance from a point $x$ to the data points $f_i$: 
\[ F_s(x) = \sum_{i=1}^{M} w_i(x) f_i \quad \text{(III.1)} \]
\[ w_i(x) = \frac{d_i(x)^{-\mu}}{\sum_{j=1}^{M} d_j(x)^{-\mu}} \quad \text{(III.2)} \]

### III.1.1 Properties of Shepard’s Formula

**Interpolation Property**

In Shepard’s formula, the weight corresponding to a certain point becomes larger as the distance to the point becomes smaller until the value is interpolated exactly when the distance is zero, which means that if \( d_i(x) = 0 \), then \( F_s(x) = f_i \), the proof is straightforward if we take the limit of (III.2) as \( d_i(x) \to 0 \). This is what we refer to as the interpolation property.

**Boundedness of the Surface**

Surfaces produced by Shepard’s formula are bounded by the minimum and maximum values of the data points:

\[ \min_i(f_i) \leq F_s(x) \leq \max_i(f_i) \quad \forall x \in \mathcal{D} \quad \text{(III.3)} \]

This property comes from the fact that \( F_s(x) \) is a convex combination of the data points, i.e. \( 0 \leq w_i(x) \leq 1 \) for all \( i \) and \( \sum_{i=1}^{M} w_i(x) = 1 \), where \( M \) denotes the total number of data points.
Continuity

Another important property is the continuity of Shepard’s formula, which depends on the distance power $\mu$:

- $F_s \in C^\infty$ if $\mu$ is a positive even integer.
- $F_s \in C^{\mu-1}$ if $\mu$ is a positive odd integer.
- $F_s \in C^{\lfloor \mu \rfloor}$ if $\mu$ is positive but not an integer.

where $\lfloor \mu \rfloor$ denotes the floor function. This results are proven by Gordon and Wixom in [36], where they also explore the effects of $\mu$ on the interpolation surface. These properties are further explored by Barnhill, Dube and Little in [37].

Basically, we have that:

- When $0 < \mu < 1$, then $F_s \in C^0$ and the function presents cusps at the data points (see Figure III.1a).
- When $\mu = 1$, then $F_s \in C^0$ and the function presents corners at the data points (see Figure III.1b).
- When $\mu > 1$, then $F_s$ belongs at least to $C^{\lfloor \mu \rfloor}$ and the function presents flat spots at the data points (see Figures III.1c and III.1d).
(a) At $\mu = \frac{1}{2}$, the function shows cusps at the data points. 

(b) At $\mu = 1$, the function shows corners at the data points. 

(c) At $\mu = 2$, the function shows flat spots at the data points. 

(d) $\mu = 10$. The function becomes flatter as $\mu$ increases.

Figure III.1: Influence of $\mu$ in the properties of the interpolation function.

These properties will be fundamental for the purposes of our optimization algorithm. In the rest of the document, we assume that $\mu$ is an integer greater than or equal to 2.

### III.2 Generalization of Shepard’s Formula for Linear Constraints in a Convex Optimization Context

As mentioned at the beginning of the chapter, Rockwood and Gao proposed a generalization of Shepard’s formula to create multi-sided patches from given curves in space. This method was aimed mainly at attribute-based 3-D and terrain modelling, 3D model compression, among other computer graphics related applications. However, it was also noticed that such a generalization of Shepard’s formula could
have potential in the optimization field as we will see next.

We begin by stating the problem that will be of our concern, an optimization problem with linear inequality constraints:

\[
\begin{align*}
\text{minimize} & \quad f_0(x) \\
\text{subject to} & \quad Ax \geq b
\end{align*}
\]

where \( f_0 : \mathbb{R}^n \to \mathbb{R} \) is a convex objective function, \( A \in \mathbb{R}^{m \times n} \) and \( b \in \mathbb{R}^m \). Throughout this text, we assume that the row vectors \( a_i^\top \) of \( A \), which are the normals of the inequality constraint planes, are normalized, i.e. \( \|a_i^\top\|_2 = 1 \).

We will generalize Shepard’s formula by interpolating the values obtained by evaluating the objective function \( f_0(x) \) at the constraints \( Ax \geq b \) so that the equality holds, i.e. we will interpolate the subspaces given by \( \{ x, f_0(x) \} \) such that \( Ax = b \).

We define the function \( d_i(x) : \mathbb{R}^n \to \mathbb{R} \) as the shortest (thus perpendicular) distance from a point \( x \) to the \( i \)th constraint:

\[
d_i(x) = a_i^\top x - b
\]

which by the standard that we assume, is a positive quantity within the feasible region of the problem. Next, we define the function \( x_i : \mathbb{R}^n \to \mathbb{R}^n \) to be a function that projects \( x \) to the \( i \)th constraint along its normal direction:

\[
x_i(x) = x - d_i(x)a_i
\]

The negative sign comes from the inequality form \( Ax \geq b \) that we are using.
Next, we define $F_S : \mathbb{R}^n \to \mathbb{R}$ to be the generalization of Shepard’s formula for problem (III.4):

$$F_S(x) = \sum_{i=1}^{m} w_i(x) f_0(x_i(x)) \quad (III.7)$$

where the weight function $w_i(x) : \mathbb{R}^n \to \mathbb{R}$ is:

$$w_i(x) = \frac{\frac{1}{d_i(x)^\mu}}{\sum_{j=1}^{m} \frac{1}{d_j(x)^\mu}} \quad (III.8)$$

Figure III.2 depicts an interpolation point $x$ and its perpendicular projections to the constraints.

![Diagram](image)

**Figure III.2:** An interpolation point $x$ is projected to the constraint planes. The projection points will then serve as data points for the interpolation.

We now prove the interpolation property of the surface when the distance to one
or more constraints decreases to zero. Furthermore, we analyse what happens to the
gradient of the interpolation function as we approach the boundary. This will be the
key to the optimization algorithm.

III.2.1 Interpolation Property of the Surface

Theorem 1. The generalized Shepard’s formula for linear inequality constraints (III.7)
using weights of the form (III.8) interpolates exactly the values of the objective func-
tion at the boundary of the feasible region, i.e. \( F_S(x) = f_0(x) \) if there is at least one
constraint \( i \) such that \( a_i^\top x = b_i \).

Proof. We begin by proving that this is true for one constraint only.

We substitute the weight expression (III.8) in (III.7):

\[
F_S(x) = \sum_{i=1}^m \frac{1}{d_i(x)^\mu} f_0(x_i(x)) \left( \frac{1}{\sum_{h=1}^m \frac{1}{d_h(x)^\mu}} \right)
\]

Let \( x \) approach a point \( \xi \) on the \( j \)th constraint so that \( d_j(x) \to 0 \). Without loss of
generality, we assume that the direction of approach to \( \xi \) is perpendicular to the
constraint. We can express (III.9) as:

\[
F_S(x) = \frac{1}{d_j(x)^\mu} f_0(x_j(x)) + \sum_{i \neq j} \frac{1}{d_i(x)^\mu} f_0(x_i(x)) \left( \frac{1}{d_j(x)^\mu} \right) + \sum_{h \neq j} \frac{1}{d_h(x)^\mu} + \sum_{h \neq j} \frac{1}{d_h(x)^\mu}
\]

We can now multiply this expression by \( \frac{d_j(x)^\mu}{d_j(x)^\mu} \):

\[
F_S(x) = \frac{d_j(x)^\mu}{d_j(x)^\mu} \frac{1}{d_j(x)^\mu} f_0(x_j(x)) + \sum_{i \neq j} \frac{d_j(x)^\mu}{d_i(x)^\mu} \frac{1}{d_j(x)^\mu} f_0(x_i(x)) \left( \frac{d_j(x)^\mu}{d_j(x)^\mu} \right) + \sum_{h \neq j} \frac{d_j(x)^\mu}{d_h(x)^\mu} + \sum_{h \neq j} \frac{d_j(x)^\mu}{d_h(x)^\mu}
\]
Now, by L’Hôpital’s rule we have that \( \lim_{d_j(x) \to 0} \frac{d_j(x)^n}{d_j(x)^n} = 1 \) and that \( \lim_{d_j(x) \to 0} \frac{d_j(x)^n}{d_i(x)^n} = 0 \) for any \( i \neq j \). Furthermore, \( \lim_{d_j(x) \to 0} x_j(x) = \xi \) Therefore:

\[
\lim_{d_j(x) \to 0} F_S(x) = f_0(\xi)
\]  

(III.12)

So this proves the interpolation property when the distance to one constraint becomes zero.

Now, we prove the interpolation property for the case of the intersection of two or more constraints.

Let \( x \) approach a point \( \xi \) on the the intersection of some set of constraints \( S \), so that \( d_i(x) \to 0 \) for all \( i \in S \), and let the approximation be in some fixed but arbitrary direction \( p \). We can express the those distances \( d_i(x) \) in terms of a common distance function \( \rho(x) \) as:

\[
d_i(x) = \rho(x)(a_i^\top p) \quad \forall i \in S
\]  

(III.13)

where \( \rho(x) = \xi - x \). So as \( x \to \xi \) we have that \( \rho(x) \to 0 \).

Rearranging the interpolation formula and plugging in (III.13):

\[
F_S(x) = \sum_{i \in S} \frac{1}{(\rho(x)(a_i^\top p))^n} \frac{f_0(x_i(x))}{d_i(x)^n} + \sum_{h \notin S} \frac{1}{d_h(x)^n} f_0(x_i(x)) + \sum_{i \notin S} \frac{1}{d_i(x)^n} f_0(x_i(x)) + \sum_{h \notin S} \frac{1}{d_h(x)^n} f_0(x_h(x))
\]  

(III.14)

Multiplying numerator and denominator by \( \rho(x)^n \):
\[ F_S(x) = \sum_{i \in S} \frac{\rho(x)^\mu}{(\rho(x)(a_i^j p))^\mu} f_0(x_i(x)) + \sum_{i \not\in S} \frac{\rho(x)^\mu}{(\rho(x)(a_i^j p))^\mu} + \sum_{i \not\in S} \frac{\rho(x)^\mu}{(\rho(x)(a_i^j p))^\mu} \sum_{h \in S} \rho(x)^\mu \sum_{h \not\in S} \frac{\rho(x)^\mu}{(\rho(x)(a_i^j p))^\mu} + \sum_{h \not\in S} \frac{\rho(x)^\mu}{(\rho(x)(a_i^j p))^\mu} \]

(III.15)

Taking the limit as \( \rho(x) \to 0 \) the right term vanishes and we have:

\[
\lim_{\rho(x) \to 0} F_S(x) = \sum_{i \in S} \frac{1}{(a_i^j p)^\mu} f_0(\xi) = f_0(\xi) \left( \sum_{i \in S} \frac{1}{(a_i^j p)^\mu} \right) = f_0(\xi) \quad (III.16)
\]

So the interpolation property is proved.

\[\square\]

### III.2.2 Gradient of the Interpolation Surface

The gradient of the generalized Shepard’s formula (III.7) is:

\[
\nabla F_S(x) = \sum_{i=1}^{m} \nabla w_i(x) f_0(x_i(x)) + \sum_{i=1}^{m} w_i(x) J(x_i(x))^\top \nabla f_0(x_i(x))
\]

(III.17)

where \( J(x_i(x)) \) is the Jacobian matrix of the projection function \( x_i(x) \).

**Theorem 2.** The gradient of the generalized Shepard’s formula for linear inequality constraints (III.7) using weights of the form (III.8) converges to a convex combination of the projections of the gradient of the original objective function to the null-spaces of inequality constraints for which equality holds. In other words, \( \nabla F_S(x) = \sum_\omega \omega_i [I - a_i a_i^\top] \nabla f_0(x) \), where \( 0 \leq \omega_i \leq 1 \) and \( \sum_\omega \omega_i = 1 \) for some con-
straints such that $a_i^T x = b_i$.

**Proof.** To make the analysis of the gradient function simpler, we split it in two terms so that:

\[
\nabla F_S(x) = \nabla_1 F_S(x) + \nabla_2 F_S(x) \tag{III.18}
\]

\[
\nabla_1 F_S(x) = \sum_{i=1}^{m} \nabla w_i(x) f_0(x_i(x)) \tag{III.19}
\]

\[
\nabla_2 F_S(x) = \sum_{i=1}^{m} w_i(x) J(x_i(x))^T \nabla f_0(x_i(x)) \tag{III.20}
\]

We begin with the analysis of $\nabla_1 F_S(x)$. We will use the same notation as in the last section and again let $x$ approach a point $\xi$ on the boundary of the feasible region along the direction $p$. Let $S$ be the set of constraints for which $d_i(x) \to 0$.

\[
\nabla_1 F_S(x) = \sum_{i=1}^{m} \left[ \frac{(\sum_{h=1}^{m} \frac{1}{d_h(x)^{p}})(-\mu) \frac{1}{d_i(x)^{p+1}} \nabla d_i(x)}{(\sum_{h=1}^{m} \frac{1}{d_h(x)^{p}})^2} \right. \\
\left. - \frac{1}{d_i(x)^{p}} \frac{(\sum_{h=1}^{m} \frac{1}{d_h(x)^{p+1}} \nabla d_h(x)}{(\sum_{h=1}^{m} \frac{1}{d_h(x)^{p}})^2} \right] f_0(x_i(x)) \tag{III.21}
\]

In order to analyze the expression when $x$ approaches the boundary, we take the same approach as in the last section and split the summations of expression (III.21) in the summations that involve the indices of the distances $i \in S$ (which are decreasing towards zero) and the indices of the rest of the distances $i \notin S$. 
\[ \nabla_1 F_S(x) = \sum_{i \in S} \left[ \frac{\left( \sum_{h \in S} \frac{1}{d_h(x)^\mu} + \sum_{h \notin S} \frac{1}{d_h(x)^\mu} \right)(-\mu)\left( \frac{1}{d_i(x)^{\mu+1}} \right) \nabla d_i(x)}{\left( \sum_{h \in S} \frac{1}{d_h(x)^\mu} + \sum_{h \notin S} \frac{1}{d_h(x)^\mu} \right)^2} \right] f_0(x_i(x)) \]

Next, we make use of the substitution \( d_i(x) = \rho(x)(a_i^T p) \), for all \( i \in S \)

\[ \nabla_1 F_S(x) = \sum_{i \in S} \left[ \frac{\left( \sum_{h \in S} \frac{1}{(\rho(x)(a_i^T p))^\mu} + \sum_{h \notin S} \frac{1}{d_h(x)^\mu} \right)(-\mu)\left( \frac{1}{(\rho(x)(a_i^T p))^{\mu+1}} \right) \nabla d_i(x)}{\left( \sum_{h \in S} \frac{1}{(\rho(x)(a_i^T p))^\mu} + \sum_{h \notin S} \frac{1}{d_h(x)^\mu} \right)^2} \right] f_0(x_i(x)) \]

(III.23)

Now, we multiply this expression by \( \frac{\rho(x)^{2\mu}}{(\rho(x))^{2\mu}} \) since \( 2\mu \) is the highest power of \( \rho(x) \) in the denominator:
\[ \nabla_1 F_S(x) = \sum_{i \in S} \left[ \left( \frac{1}{\rho(x)(p(x)_{i,j})} \right) \frac{\sum_{h \in S} \frac{1}{d_h(x)}}{\rho(x)(a_{i,j})} + \sum_{h \notin S} \frac{1}{d_h(x)} \right] (-\mu) \left( \frac{\rho(x)^2}{\rho(x)(a_{i,j})} \right) \nabla d_i(x) \]

\[ - \frac{1}{\rho(x)(a_{i,j})} \left( \frac{\sum_{h \in S} \frac{1}{d_h(x)}}{\rho(x)(a_{i,j})} + \sum_{h \notin S} \frac{1}{d_h(x)} \right) \nabla d_i(x) \]

\[ + \sum_{i \notin S} \left[ \left( \frac{1}{\rho(x)(p(x)_{i,j})} \right) \frac{\sum_{h \in S} \frac{1}{d_h(x)}}{\rho(x)(a_{i,j})} + \sum_{h \notin S} \frac{1}{d_h(x)} \right] (-\mu) \left( \frac{\rho(x)^2}{\rho(x)(a_{i,j})} \right) \nabla d_i(x) \]

\[ - \frac{1}{d_i(x)^2} \left( \frac{\sum_{h \in S} \frac{1}{d_h(x)}}{\rho(x)(a_{i,j})} + \sum_{h \notin S} \frac{1}{d_h(x)} \right) \nabla d_i(x) \]

\[ f_0(x_i(x)) \]

(III.24)

Now we take the limit of \( \nabla_1 F_S(x) \) as \( \rho(x) \to 0 \). It would be too cumbersome if we expanded the sum in the last expression further so we first notice that the denominator would include terms of the form \( \frac{\rho(x)^2}{\rho(x)(a_{i,j})} \), and it will not vanish. Next we notice that the second part of the sum related to the indices \( i \notin S \) vanishes since all of the terms including ones of the form \( \frac{\rho(x)^2}{\rho(x)(a_{i,j})} \) vanish by L’Hôpital’s rule when \( \mu \geq 2 \).

From the first part of the sum, related to the indices \( i \in S \), we eliminate the terms that readily vanish and we also remember that projections \( x_i(x) \) for all \( i \in S \) approach the same point \( \xi \), so we have \( f_0(x_i(x)) \to f_0(\xi) \) for all \( i \in S \), and we can take that factor out of the sum. Thus we have:
\[
\lim_{\rho(x) \to 0} \nabla_1 F_S(x) = f_0(\xi) \lim_{\rho(x) \to 0} \sum_{i \in S} \left[ \left( \sum_{h \in S} \frac{1}{(\rho(x)(a_h^T p))^\mu} \right) \left( -\mu \frac{\rho(x)^{2\mu}}{(\rho(x)(a_h^T p))^{\mu+1}} \right) \nabla d_i(x) \right] \\
- \frac{1}{(\rho(x)(a_i^T p))^\mu} \left( \sum_{h \in S} \left( -\mu \frac{\nabla d_h(x) \rho(x)^{2\mu}}{(\rho(x)(a_h^T p))^{\mu+1}} \right) \right) \\
- \frac{1}{(\rho(x)(a_i^T p))^\mu} \left( \sum_{h \in S} \left( -\mu \frac{\nabla d_h(x) \rho(x)^{2\mu}}{(\rho(x)(a_h^T p))^{\mu+1}} \right) \right)
\]

(III.25)

\[
\lim_{\rho(x) \to 0} \nabla_1 F_S(x) = \frac{f_0(\xi)(-\mu)}{\left( \sum_{h \in S} \frac{1}{(a_h^T p)^\mu} \right)^2} \lim_{\rho(x) \to 0} \sum_{i \in S} \left[ \sum_{h \in S} \frac{\rho(x)^{2\mu} \nabla d_i(x)}{\rho(x)^{2\mu+1}(a_h^T p)^\mu(a_i^T p)^{\mu+1}} \right] \\
- \sum_{h \in S} \frac{\rho(x)^{2\mu} \nabla d_h(x)}{\rho(x)^{2\mu+1}(a_i^T p)^\mu(a_h^T p)^{\mu+1}}
\]

(III.26)

It is now clear to see that all the terms inside the braces will cancel out since the sums of the indices \( i \) and \( h \) are done over the same set \( S \). We conclude that:

\[
\lim_{\rho(x) \to 0} \nabla_1 F_S(x) = 0 \quad \text{ (III.27)}
\]

For the analysis of \( \nabla_2 F_S(x) \) we start by restating the equation [III.20]:

\[
\nabla_2 F_S(x) = \sum_{i=1}^m w_i(x) J(x_i(x))^\top \nabla f_0(x_i(x))
\]

where the transpose of the Jacobian matrix is given by:
\[ \mathcal{J}(x_i(x))^\top = \mathcal{J} \left[ x - (a_i^\top x - b)a_i \right] = I - a_ia_i^\top \]  (III.28)

As we recall, \( a_i^\top \) is the normal of the \( i \)th inequality constraint. We also notice that the Jacobian (III.28) turns out to have the form of a projector matrix to the null space of \( a_i \) since we have assumed that the all vectors \( a_i \) are normalized.

\( \nabla_2 F_S(x) \) is basically a convex combination of the factors \( \mathcal{J}(x_i(x))^\top \nabla f_0(x_i(x)) \) with the same weights as in formula (III.2). Therefore, when we take the limit as \( x \) approaches the boundary, the same conclusions apply.

For the case when \( x \) approaches a point \( \xi \) on the boundary where only one constraint \( j \) is active, we have that:

\[
\lim_{x \to \xi} \nabla_2 F_S(x) = \lim_{x \to \xi} \mathcal{J}(x_j(x))^\top \nabla f_0(x_j(x)) = [I - a_ja_j^\top] \nabla f_0(\xi) \]  (III.29)

Furthermore, when \( x \) approaches a point \( \xi \) lying on the intersection of two or more constraint planes \( j \in S \) such that the distances \( d_j(x) \to 0 \) the result is a convex combination of vector projections:

\[
\lim_{x \to \xi} \nabla_2 F_S(x) = \sum_{j \in S} \frac{1}{(a_j p)^\mu} \left( \frac{1}{(a_h p)^\mu} \right) \left( [I - a_ja_j^\top] \nabla f_0(\xi) \right) \]  (III.30)

Moreover, since \( \lim_{x \to \xi} \nabla_1 F_S(x) = 0 \), the conclusion for the full gradient of the interpolation formula is:
\[
\lim_{x \to \xi} \nabla F_S(x) = \sum_{j \in S} \frac{1}{(a_j \cdot p)^\mu} \left( \frac{1}{(a_h \cdot p)^\mu} \frac{1}{a_j} \nabla f_0(\xi) \right) \quad (III.31)
\]

It is clear that the weights of the combination are bounded between zero and one if both vectors \( a_j \) and \( p \) are normalized. Moreover, they add up to one since each term is divided by the whole sum.

It is true that in formula (III.31), the weights depend on the direction of approach \( p = \xi - x \), so we can still be near the intersection and the gradient will vary between the projections on each plane depending on the relative position of \( x \) with respect to the intersection, i.e. if we are much closer to a plane, then the weight of that projection \( \frac{1}{(a_j \cdot p)^\mu} \) will be greater. This means that there is a singularity in the gradient at intersections of the planes. To solve this issue, in the actual implementation, we define a small threshold away from the constraints after we assume the equality holds and we choose the weights \( \frac{1}{(a_j \cdot p)^\mu} \) to be equal to each other as if we were always approaching in a central direction.

This is a very important property for our algorithm since it means that the cross-derivative at the boundary of the feasible region is zero. Even if we just followed the negative of the gradient in a steepest descent manner, starting from anywhere inside the feasible region we will never step out if we are careful enough in the line search, that is, if we make sure that the iterates remain feasible with the maximum step length allowed.
III.3 Potential Use of the Interpolation Surface in an Optimization Context

To illustrate the importance of the properties shown in the last section and how they can be of utility in an optimization context, we take an example of a quadratic program (QP). Figure III.3 shows an instance of a QP, a convex paraboloid as the objective function subject to linear inequality constraints.

Figure III.3: Example of a quadratic program. The red lines represent the projection of the boundary on the objective function. The interpolation function will go through these lines.

Figure III.4 shows clearly the potential of the interpolation function for an optimization algorithm: the gradient field of the surface readily leads to the optimum from any point within the feasible region.

Figure III.5 depicts the interpolation surface for the example problem intersecting exactly the original objective function at the boundary of the feasible region.

In the last section we proved two key theorems of the interpolation function: first, that the generalized Shepard’s formula interpolates the original objective function at
Figure III.4: The gradient field of the interpolation function leads to the optimum. The feasible region of the problem is the inside of the pentagon.

the boundary of the feasible region. This property is important since we know that if at least one constraint is active in the original problem, then the optimum lies on the boundary of the feasible region. Most of the times this is the case, since the constraints are redundant if the optimum lies inside the feasible region i.e. we could just solve the unconstrained problem.

The second property is that the cross-derivatives of the interpolation function at the boundary of the feasible region are zero. As we said before, this is a key property since the gradient field readily leads us to the optimum from any point inside the feasible region and it converges to the projection of the original gradient on the null-space of the constraints.

With this in mind, we could use a typical descent method for unconstrained minimization such as steepest descent, quasi-Newton or Newton method in a line search or trust-region framework to find the solution to the optimization problem. We explore
such descent methods in the next chapter.

There is only one more desirable property: that the interpolation surface does not have a point on the inside of the feasible region whose value is less than the minimum of the original problem. In other words, the interpolation function should not have any stationary points inside the feasible region which could cause the descent methods to converge to those points instead of the actual solution to the original problem. Unfortunately, for the general case, we cannot be sure of that since as seen in Figure III.2 sometimes the projections on the constraints used by the interpolation formula are outside the feasible region, those points can have a lower value than the optimum thus making it possible that there is a point in the interpolation surface, inside the feasible region, that has a smaller value than the minimum of the problem.

Figure III.6 shows the stationary point effect in a 1D problem with three data points created with a linear function. We interpolate the points via Shepard’s formula with $\mu = 2$. We take the feasible region to be the orange coloured segment, while the infeasible region is represented by the blue segments. The real optimum of this
Shepard’s interpolation of the 3 data points.

Figure III.6: 1D toy problem exemplifying the stationary point effect.

(a) Shepard’s interpolation of the 3 data points. (b) Close up image of the middle point and the stationary point effect.

For this example we used a linear function to create the data points. The possibility of having stationary points near the solution of the problem should be reduced as we increase the convexity of the function since we have a positive curvature. Nevertheless, this increases the possibility of having a stationary point near the maximum values of the boundary. Moreover, increasing the value of $\mu$ could help decrease the influence of data points at greater distance and help diminish this effect, but it would also make the computation of the surface less accurate.

A special case of convex problems where we can make sure that there are no stationary points is the case when we have only bounds on the variables. The logic behind this is that in this scenario, we have $n$ parallel pairs of planes which are perpendicular among each other. Thus projections outside of the feasible region are avoided (see Figure III.7).

We concentrate our efforts in developing optimization methods based on the gen-
Figure III.7: In the box-constrained case, only boundary points are used for the interpolation function.

Generalization of the Shepard’s interpolation formula for these type of problems.
Chapter IV

Optimization Algorithm

In this Chapter we will study the descent strategies for the interpolation surface described in the last chapter. We describe the paradigms of line search and trust region and show that the line search method seems to be more suitable for our case. Next we describe the steepest descent method, quasi-Newton and Newton methods that were used in the optimization algorithms. The implementation of these methods was done for the Quadratic Programming case with bounds on the variables:

$$\min \ f_0(x) = \frac{1}{2} x^\top P x + q^\top x + r$$

subject to $lb \geq x \geq ub$

where $P \in S^n_+$, $q \in \mathbb{R}^n$, $r \in \mathbb{R}$, $lb \in \mathbb{R}^n$, and $ub \in \mathbb{R}^n$. This is one of the simplest but most important cases of Convex Optimization. Our intention is to demonstrate the technique but it can be expanded to other cases as well.
IV.1 Iteration Paradigms: Line Search and Trust Region

Optimization algorithms generate a sequence of points \( x_k \) starting from an initial point \( x_0 \) until some stopping criteria is met, hopefully achieving convergence to the solution \( x^* \). There are mainly two paradigms of carrying out the iterations. The line search strategy consists of finding a direction of descent \( p_k \) and then attempting to find a parameter \( \alpha_k \) such that \( f_0(x_k + \alpha_k p_k) \) is minimized. In the trust region strategy, we create an approximate model of the objective function that is valid within a defined "trust region" and then find the step \( s_k \) that minimizes the model subject to the validity region constraints.

IV.1.1 Line Search

In the line search paradigm, given an iterate \( x_k \), we try to find \( x_k + \alpha_k p_k \) where \( p_k \) is the descent direction and \( \alpha_k \) is the step length. Choices for \( p_k \) include, among others, \(-\nabla f_0(x_k)\), the steepest descent direction; \(-\nabla^2 f_0(x_k)^{-1}\nabla f_0(x_k)\), the Newton direction; or \(-B_k^{-1}\nabla f_0(x_k)\) in the quasi-Newton approach, where \( B_k \) is an approximation of the Hessian matrix.

The step length \( \alpha_k \) must be chosen carefully to guarantee sufficient descent. In an optimal scenario, we would like to find \( \alpha_k \) such that \( f_0(x_k + \alpha_k p_k) \), for \( \alpha_k > 0 \), is exactly minimized.

The Wolfe Conditions

Imposing the condition that \( f_0(x_k + \alpha_k p_k) < f_0(x_k) \) will not guarantee convergence to the solution since the decrease can become too small for convergence. The Armijo condition (IV.2) forces sufficient decrease on the objective function:
\[ f_0(x_k + \alpha_k p_k) \leq f_0(x_k) + c_1 \alpha_k \nabla f_0(x_k)^\top p_k, \quad c_1 \in (0, 1) \quad (IV.2) \]

This condition makes sure that the decrease is at least some constant factor of the step length times the directional derivative \( \nabla f_0(x_k)^\top p_k \), but is not enough to ensure that we make sufficient progress. Therefore, we must enforce a condition that ensures that steps are long enough.

\[ \nabla f(x_k + \alpha_k p_k)^\top > c_2 \nabla f_0(x_k)^\top p_k \quad (IV.3) \]

Condition \( (IV.3) \) is called the curvature condition and together with \( (IV.2) \) they form the so called \textit{Wolfe conditions}.

**Backtracking Line Search**

One common way to avoid enforcing condition \( (IV.3) \) explicitly is to apply the backtracking strategy. This method consists of starting with an initial value \( \alpha^0 \) and checking condition \( (IV.2) \). If the condition is not met, we then set \( \alpha^{i+1} = \rho \alpha^i \), for \( \rho \in (0, 1) \) and repeat the procedure until the condition is met. This is a sound technique since it ensures that if the initial value is not accepted, we are only decreasing the parameter by a reasonable factor.

In some algorithms for constrained optimization, including ours, we must first make sure that \( \alpha_k \) is not too large to cause the algorithm to step out of the feasible region. Moreover, the initial value \( \alpha^0 \) can be set to a suitable value depending on the scaling of the problem or could be set dynamically at each iteration to ensure a proper size.
Algorithm 1 Backtracking line search for constrained problems

Require: $\alpha^0 > 0, \rho \in (0, 1), c_1 \in (0, 1)$

$\alpha_k \leftarrow \alpha^0$

repeat

$a_k \leftarrow \rho \alpha_k$

until $A(x_k + \alpha_k p_k) \geq b$ \quad $\triangleright$ Ensure feasibility

while $f_0(x_k + \alpha_k p_k) \geq f_0(x_k) + c_1 \alpha_k \nabla f_0(x_k)^T$ do

$a_k \leftarrow \rho \alpha_k$

end while

IV.1.2 Trust Region

In the trust region strategy, we create an approximate model of the objective function in the neighbourhood of the current iterate $x_k$. The model is usually quadratic and comes from the second order Taylor series expansion of $f_0(x_k + s_k)$ where $s_k$ is the step that we take to find the next iterate. Since the model stops being accurate far away from the current point, we limit the length of $s_k$ to some form of "trust region" which is generally a sphere such that $\|s_k\|_2 \leq r$ for some radius $r$, but can also take other shapes. We then minimize the approximate model with this constraint.

The model then, has the form:

\[
\begin{align*}
\text{minimize} & \quad f_0(x_k) + \nabla f_0(x_k)s_k + \frac{1}{2}s_k^T B_k s_k \\
\text{subject to} & \quad \|s_k\|_2 \leq r
\end{align*}
\]  

(IV.4)

where $B_k$ is either the Hessian matrix of $f_0(x_k)$ or an approximation of it. In a trust region type algorithm we solve a sequence of problems of the form (IV.4) and adapt the size of the region at each iteration. We evaluate the model for the quality of the last step, if it was not good enough, we decrease the size of the region. If the model was good, but the minimum of the sub problem fell inside the region, we leave the size unchanged. If the model was good, but the minimum was on the boundary, we
increase the size of the region.

Solving problem (IV.4) can be time consuming and it is not normally solved exactly in practice. Methods to approximate the solution are the dogleg method [38], two subspace minimization [39] or iterative solutions [40].

(a) Solution of a 2D QP problem with a(b) Close up image of solution neighborhood.

Figure IV.1: As the iterates approach the boundary, the trust region radius becomes smaller, thus hurting the convergence speed.

In a constrained optimization context, in some algorithms we must be careful not to allow the step to take the iterate outside of the feasible region. This is the case of our algorithm, therefore, a limitation must be enforced so that the size of the trust region does not allow the possibility of stepping to the infeasible region. This can hurt the convergence speed of the algorithm if we are too close to the boundary since the steps taken will be unnecessarily short or even cause premature termination of the algorithm due to tolerance limits. Figure IV.1 shows this phenomenon in a 2-D QP. The method used to descend on the interpolation surface was a naïve trust region quasi-Newton algorithm with exact minimization of the step subproblem. For this reason, we think the line search paradigm is a better choice for our algorithm.
IV.2 Descent Methods

In this section we describe the descent methods chosen for our optimization algorithm: steepest descent, BFGS, damped BFGS and Newton’s method. Since these methods will be used on the interpolation function, throughout this section we will use $F_S(x)$ instead of $f_0(x)$ to denote the objective function.

IV.2.1 Steepest Descent

Steepest descent is probably the most straightforward descent algorithm since it only requires the computation of the gradient. Basically, at each step we choose the descent direction $p_k = -\nabla F_S(x_k)$ and perform a line search such as backtracking described in algorithm 1. The advantage of steepest descent is that it is very simple and inexpensive to compute. The main disadvantage is that for some problems the convergence to the solution can be extremely slow, requiring a very large number of iterations.

Algorithm 2 Steepest descent method for the interpolation surface

Require: $\alpha_0 > 0, x_0 : Ax_0 \geq b, xTol > 0, gTol \geq 0, fTol > 0, maxIter \in \mathbb{N}$

$k \leftarrow 0$

$F_S(x_{-1}) \leftarrow \infty$

while $k \leq maxIter$ and $\|\nabla F_S(x_k)\|_2 > gTol$ and $F_S(x_{k-1}) - F_S(x_k) > fTol$ do

$p_k \leftarrow -\nabla F_S(x_k)$

$\alpha_k \leftarrow \text{BacktrackingLineSearch}(x, p, \alpha_0)$

if $\|\alpha_k p_k\|_2 < xTol$ then

break

end if

$x_{k+1} \leftarrow x_k + \alpha_k p_k$

$k \leftarrow k + 1$

end while

IV.2.2 Quasi-Newton Methods

Quasi-Newton methods refer to a class of algorithms that just like Newton’s method, model an objective function $f(x_k + p)$ with a quadratic approximation of the form:
\[ f(x_k) + \nabla f(x_k)^\top p + \frac{1}{2} p^\top B_k f(x_k)p, \]  

where \( B_k \) is a symmetric positive definite matrix. Whereas in Newton’s method, \( B_k \) is the Hessian of the objective function \( \nabla^2 f(x_k) \), in a quasi-Newton method, \( B_k \) is an approximation of the Hessian built with the information from the changes of curvature gathered at each iteration.

There are several ways of performing the update to the matrix \( B_k \). The most popular approach is the BFGS update formula, named after Broyden, Fletcher, Goldfarb and Shannon, who discovered the formula. The BFGS formula is a rank-2 update to the matrix \( B \) and it is a special case of the Broyden class family. The DFP formula is closely related to BFGS and is also a special case of the same family of methods. The SR1 formula is also popular since it is only a rank-1 update on the matrix \( B \). The damped BFGS method is a modification on the original BFGS formula which enforces positive-definiteness on the matrix \( B_k \) even if the function has a non-positive curvature.

The advantages of the quasi-Newton methods are that they avoid the computation of second derivatives, the methods can sometimes be transformed to update the Cholesky factors or even the inverse of the matrix \( B_k \) thus reducing the complexity in the computation of the step direction, and that they often exhibit super linear convergence on unconstrained problems.

Quasi-Newton methods were first introduced by Davidon in the 1950’s, although his method was not published until much later. Theory of quasi-Newton methods is covered the books by Nocedal and Wright and Dennis and Schnabel. The publication by Denis and Moré presents an ample discussion of such methods and Nocedal discusses their self-correcting properties in.
BFGS

Given the quadratic approximation $IV.5$ the direction of minimization is:

$$p_k = -B_k^{-1} \nabla f(x_k)$$  \hspace{1cm} (IV.6)

So the next iterate will be $x_{k+1} = x_k + a_k p_k$ where $a_k$ is obtained so that $f(x_{k+1})$ satisfies conditions $IV.2$ and $IV.3$. We make the following definitions:

$$s_k = x_{k+1} - x_k = a_k p_k$$  \hspace{1cm} (IV.7)
$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$$ \hspace{1cm} (IV.8)
$$H_k = B_k^{-1}$$ \hspace{1cm} (IV.9)

The BFGS formula comes from the solving the following optimization problem:

$$H_{k+1} = \text{argmin} \quad \|H - H_k\|_F$$ \hspace{1cm} (IV.10)
subject to $H = H^\top$
\hspace{1cm}$H y_k = s_k$

The goal of this formulation is to obtain a new matrix $H_{k+1}$ that is as close as possible to $H_k$ in the Frobenius norm sense. The first condition enforces symmetry and the second condition comes from forcing the approximation $f(x_{k+1})$ to interpolate the gradient at the last iterate $\nabla f(x_k)$. Solving problem (IV.10) yields the following update formula for the inverse of $B_k$: 
\[ H_{k+1} = (I - \frac{s_k y_k^\top}{y_k^\top s_k}) H_k (I - \frac{y_k s_k^\top}{y_k^\top s_k}) + \frac{s_k s_k^\top}{y_k^\top s_k} \] (IV.11)

By applying the Sherman-Morrison-Woodbury formula, one can obtain the expression for the update of \( B_k \):

\[ B_{k+1} = B_k - \frac{B_k s_k s_k^\top B_k}{s_k^\top B_k s_k} + \frac{y_k y_k^\top}{y_k^\top s_k} \] (IV.12)

To ensure that \( p_k \) is a descent direction, it is necessary for \( H_k \) (or \( B_k \)) to be positive definite. One important note on the BFGS formula is that it only produces a positive definite matrix \( H_{k+1} \) if \( H_k \) is positive definite and \( s_k^\top y_k > 0 \). This condition means that the objective function must have positive curvature.

Our interpolation surface for convex problems is not convex itself, it is a quasiconvex, and in fact, a quasilinear function having positive curvature near the minimum and negative curvature far away from it. Therefore, a modification to the BFGS algorithm must be made so that the updates on the matrix \( H_k \) are only carried out when the positive curvature condition holds and otherwise perform a steepest descent step. The parameter \( \text{minCurve} \) in \( 3 \) defines the minimum curvature allowed to make the update as measured by the product \( s_k^\top y_k \).

Algorithm \( 3 \) is a modified version of the standard BFGS method which is more suitable for the interpolation surface. Besides the minimum curvature condition, we have included other modifications such as taking a steepest descent step every few iterations defined by the parameter \( \text{nSD} \). This will help the convergence of the algorithm when the Hessian approximation is not accurate enough or ill-conditioned. Another important modification involves resetting the \( H_k \) when the resulting direction \( p_k \) is not a descent direction since this is a clue that the approximation to the Hessian
Figure IV.2: The interpolation function is not convex but quasilinear. It has a region with positive curvature but also a region with negative curvature.

Moreover, we include the same termination conditions as in the steepest descent algorithm with the variation that the function value termination condition can only be imposed after a Steepest Descent step. The rationale is that, in our experience, there are times when $x_k$ is close to the boundary and the quasi-Newton direction yields a direction that points towards the boundary, making the next step very short and possibly a negligible decrease in the function value. By checking this condition only after a steepest descent step, we avoid the risk of terminating the algorithm too early.
**Algorithm 3** Modified BFGS method for the interpolation surface

**Require:** $H_0 \in S^{n+}_+, \alpha_0 > 0, x_0 : Ax_0 \geq b, \text{minCurve} > 0, xTol > 0, gTol \geq 0, fTol > 0, maxIter \in \mathbb{N}, nSD \in \mathbb{N}$

$x_1 \leftarrow \text{SteepestDescentStep}(x_0, \nabla F_S(x_0))$

$k \leftarrow 1$

$H_1 \leftarrow H_0$

while $k \leq maxIter$ and $\|\nabla F_S(x_k)\|_2 > gTol$ and $(\text{mod}(k - 1, nSD)! = 0$ or $F_S(x_{k-1}) - F_S(x_k) > fTol)$ do

if $\text{mod}(k, nSD)! = 0$ then

$p_k \leftarrow -H_k \nabla F_S(x_k)$

if $p_k^\top g_k \geq 0$ then

$H_k = H_0$

$p_k \leftarrow -\nabla F_S(x_k)$

$p_k \leftarrow p_k / \|p_k\|_2$

end if

else

$p_k \leftarrow -\nabla F_S(x_k)$

$p_k \leftarrow p_k / \|p_k\|_2$

end if

$\alpha_k \leftarrow \text{BacktrackingLineSearch}(x_k, p_k, \alpha_0)$

$s_k \leftarrow \alpha_k p_k$

if $\|s_k\|_2 < xTol$ then

break

end if

$x_{k+1} \leftarrow x_k + s_k$

$y_k = \nabla F_S(x_{k+1}) - \nabla F_S(x_k)$

if $s_k^\top y_k > \text{minCurve}$ then

$H_{k+1} = (I - \frac{s_k y_k}{y_k^\top s_k})H_k (I - \frac{y_k s_k^\top}{y_k^\top s_k}) + \frac{s_k s_k^\top}{y_k^\top s_k}$

end if

$k \leftarrow k + 1$

end while

**Damped BFGS**

The damped BFGS formula is a modification to the original BFGS formula for updating the Hessian approximation $B_k$ while enforcing positive definiteness when the positive curvature condition is not met, i.e. when $s_k^\top y_k \leq 0$. According to Nocedal and Wright [29], performing this update is preferred to just skipping the update as in algorithm 3. We first define:
\[ \theta_k = \begin{cases} 
1 & \text{for } s_k^\top y_k \geq 0.2s_k^\top B_k s_k \\
\frac{0.8s_k^\top B_k s_k}{s_k^\top B_k s_k - s_k^\top y_k} & \text{for } s_k^\top y_k < 0.2s_k^\top B_k s_k \end{cases} \quad (IV.13) \]

\[ r_k = \theta_k y_k + (1 - \theta_k) B_k s_k \quad (IV.14) \]

The update formula is:

\[ B_{k+1} = B_k - \frac{B_k s_k s_k^\top B_k}{s_k^\top B_k s_k} + \frac{r_k r_k^\top}{r_k^\top s_k} \quad (IV.15) \]

As we can see, when \( \theta_k = 1 \), formula \( (IV.15) \) is equivalent to the normal BFGS update, however, when the curvature measure \( s_k^\top y_k \) is too small or negative, the modification to the update makes sure that \( B_{k+1} \) remains positive definite. Due to the fact that this is a rank-four update. It would be extremely cumbersome to develop an update formula for \( H_k \) without involving any matrix inverse. However, it is possible to apply the Sherman-Morrison-Woodbury formula and write an update formula which involves only calculation of a \( 4 \times 4 \) matrix inverse. At this stage, we are more concerned about the precision of the method, moreover, for a low number of dimensions this will not have a significant effect.

Algorithm \[ \] describes our implementation of the damped BFGS quasi-Newton method. As with the other algorithms, this pseudocode is a somewhat simplified version of the actual implementation. We have included the actual code in the appendix for reference.
Algorithm 4 Damped BFGS method for the interpolation surface

Require: $B_0 \in S^{n+}_+, \alpha_0 > 0, x_0 : A x_0 \geq b, \minCurve > 0, xTol > 0, gTol \geq 0, fTol > 0, \maxIter \in \mathbb{N}, nSD \in \mathbb{N}$

$x_1 \leftarrow \text{SteepestDescentStep}(x_0, \nabla F_S(x_0))$

$k \leftarrow 1$

$B_1 \leftarrow B_0$

while $k \leq \maxIter$ and $\|\nabla F_S(x_k)\|_2 > gTol$ and ($\text{mod}(k-1, nSD)! = 0$ or $F_S(x_{k-1}) - F_S(x_k) > fTol$) do

if $\text{mod}(k, nSD)! = 0$ then

$p_k \leftarrow -B_k^{-1}\nabla F_S(x_k)$

if $p_k^T g_k >= 0$ then

$B_k = B_0$

$p_k \leftarrow -\nabla F_S(x_k)$

$p_k \leftarrow p_k/\|p_k\|_2$

end if

else

$p_k \leftarrow -\nabla F_S(x_k)$

$p_k \leftarrow p_k/\|p_k\|_2$

end if

$s_k \leftarrow \alpha_k p_k$

$\alpha_k \leftarrow \text{BacktrackingLineSearch}(x_k, p_k, \alpha_0)$

if $\|s_k\|_2 < xTol$ then

break

end if

$x_{k+1} \leftarrow x_k + s_k$

$y_k = \nabla F_S(x_{k+1}) - \nabla F_S(x_k)$

if $s_k^T y_k \geq 0.2s_k^T B_k s_k$ then

$r_k = y_k$

else

$\theta_k = \frac{s_k^T B_k s_k}{s_k^T B_k s_k - s_k^T y_k}$

$r_k = \theta_k y_k + (1 - \theta_k) B_k s_k$

end if

$B_{k+1} = \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + r_k r_k^T / r_k^T s_k$

$k \leftarrow k + 1$

end while

The same modifications on algorithm 3 for improving convergence are implemented in algorithm 4 as well.
IV.2.3 Newton’s Method

In the Newton’s method for optimization, we model the objective function as a quadratic one by making use of the second order Taylor approximation:

\[
\hat{f}(x_k + p) = f(x_k) + \nabla f(x_k)^\top p + \frac{1}{2} p^\top \nabla^2 f(x_k)p
\]  

(IV.16)

The step that exactly minimizes this approximation is called the *Newton Step* and is obtained by making the gradient of the expression (IV.16) equal to zero:

\[
p_{\text{newton}}^* = -\nabla^2 f(x_k)^{-1} \nabla f(x_k)
\]  

(IV.17)

After finding the newton direction, we can apply a line search procedure such as backtracking to find a proper step length. Near the optimum, Newton’s method becomes increasingly accurate so it is recommended to start the backtracking with an initial parameter value \(\alpha^0 = 1\) since, normally, it will be chosen as the step length in the neighbourhood of the solution. Naturally, in our case, we must first make sure that the step length does not cause the iterate to step out of the feasible region so the same procedure described in algorithm 1 is carried out.

The advantages of the Newton’s method are that it normally requires less iterations to converge than the steepest descent and the quasi-Newton methods and can exhibit quadratic convergence in the neighbourhood of the solution. The main disadvantage is the need for calculating the Hessian matrix of the objective function which can be quite expensive to do in certain cases and the need to solve the system of equations to find the *Newton step*. In this thesis we explore the use of the Newton’s method in the context of the interpolation function.
We now derive the expression for the Hessian matrix of the interpolation formula for the case of Quadratic Programming. We recall the expression for the gradient:

\[ \nabla F_S(x) = \sum_{i=1}^{m} \nabla w_i(x) f_0(x_i(x)) + \sum_{i=1}^{m} w_i(x) \mathcal{J}(x_i(x)) \top \nabla f_0(x_i(x)) \]

The Hessian is then given by:

\[
\nabla^2 F_S(x) = \sum_{i=1}^{m} \nabla^2 w_i(x) f_0(x_i(x)) + \sum_{i=1}^{m} \mathcal{J}(x_i(x)) \top \nabla f_0(x_i(x)) \nabla w_i(x) \top \\
+ \sum_{i=1}^{m} \nabla w_i(x) \nabla f_0(x_i(x)) \top \mathcal{J}(x_i(x)) + \sum_{i=1}^{m} w_i(x) \nabla^2 f_0(x_i(x)) \mathcal{J}(x_i(x)) \]

(IV.18)

In our case, since we have only linear constraints, the Jacobian matrix of the projection points \( \mathcal{J}(x_i(x)) \) is actually a constant matrix, so no second derivatives of this matrix are included in expression (IV.18). We define:

\[
\nabla^2 F_{S1}(x) = \sum_{i=1}^{m} \nabla^2 w_i(x) f_0(x_i(x)) \]  

(IV.19)

\[
\nabla^2 F_{S2}(x) = \sum_{i=1}^{m} \mathcal{J}(x_i(x)) \top \nabla f_0(x_i(x)) \nabla w_i(x) \top \]  

(IV.20)

\[
\nabla^2 F_{S3}(x) = \sum_{i=1}^{m} \nabla w_i(x) \nabla f_0(x_i(x)) \top \mathcal{J}(x_i(x)) \]  

(IV.21)

\[
\nabla^2 F_{S4}(x) = \sum_{i=1}^{m} w_i(x) \nabla^2 f_0(x_i(x)) \mathcal{J}(x_i(x)) \]  

(IV.22)

Expanding the expressions:
\[ \nabla^2 F_{S_1}(x) = \sum_{i=1}^{m} \left[ \sum_{j=1}^{m} (-\mu) \frac{1}{d_j(x)^{\mu+1}} a_i \right] \frac{1}{d_i(x)^{\mu+1}} a_i^T \left( \sum_{j=1}^{m} \frac{1}{d_j(x)^{\mu+1}} a_i a_i^T \right) f_0(x_i(x)) \right] \]

\[ - \sum_{i=1}^{m} \left[ (-\mu) \frac{1}{d_i(x)^{\mu+1}} a_i \left( \sum_{j=1}^{m} \frac{1}{d_j(x)^{\mu+1}} a_i a_i^T \right) \frac{1}{d_i(x)^{\mu+1}} \right] f_0(x_i(x)) \]

\[ - \sum_{i=1}^{m} \left[ (-\mu) \frac{1}{d_i(x)^{\mu+1}} a_i \left( \sum_{j=1}^{m} \frac{1}{d_j(x)^{\mu+1}} a_i a_i^T \right) \frac{1}{d_i(x)^{\mu+1}} \right] f_0(x_i(x)) \]

For the case with general linear constraints, the Hessian computation can be very expensive. However, for the case with only bounds on the variables, several
simplifications can be done on the equations. For example, we know that the product 
\[(I - a_i a_i^\top)[P x_i(x) + q]\] is a projector of the gradient of the quadratic equation at \(x_i(x)\) to the null space of the vector \(a_i\), in the general case we need to perform that matrix vector multiplication. In the bounded case we know that the projection only involves zeroing out the \(i\)th component of the gradient. These sort of simplifications allows us to vectorize the code to write an efficient Matlab implementation which can be found in the Appendix.

Algorithm 5 shows the pseudo code corresponding to a slightly simplified version of the actual implementation. An important thing to notice is that we perform a few steepest descent steps at the beginning. This is done to make sure that we begin with the newton iterations in the region with positive curvature which is closer to the minimum. If we begin with the newton iterations in the region of negative curvature, the algorithm will have an ascent direction instead.
Algorithm 5  Newton’s method for the interpolation surface

Require: \( \alpha_0 = 1, x_0 : Ax_0 \geq b, xTol > 0, gTol \geq 0, fTol > 0, maxIter \in \mathbb{N}, nSD \in \mathbb{N}, \) initialSD \( \in \mathbb{N} \)

for \( k = 1 \rightarrow \) initialSD do
    \( x_k \leftarrow \text{SteepestDescentStep}(x_0, \nabla F_S(x_0)) \)
    \( k \leftarrow k + 1 \)
end for

while \( k \leq \) maxIter and \( \| \nabla F_S(x_k) \|_2 > gTol \) and \( F_S(x_{k-1}) - F_S(x_k) > fTol \) do
    if \( \text{mod}(k,nSD) \neq 0 \) then
        \( p_k \leftarrow -\nabla^2 F_S(x_k)^{-1} \nabla F_S(x_k) \)
    end if
    if \( p_k^\top g_k \geq 0 \) or \( \text{mod}(k,nSD) = 0 \) then
        \( p_k \leftarrow -\nabla F_S(x_k) \)
        \( p_k \leftarrow p_k / \| p_k \|_2 \)
    end if
    \( \alpha_k \leftarrow \text{BacktrackingLineSearch}(x_k, p_k, \alpha_0) \)
    \( s_k \leftarrow \alpha_k p_k \)
    if \( \| s_k \|_2 < xTol \) then
        break
    end if
    \( x_{k+1} \leftarrow x_k + s_k \)
    \( k \leftarrow k + 1 \)
end while
Chapter V

Implementation and Tests

In this chapter we present the implementation details of the algorithms described in the last section. We begin by briefly describing the strategy used for the Phase I of the optimization algorithm where we find a suitable initial point for the problems. We then present some example problems solved with the Steepest Descent and Newton’s method described in the last chapter and introduce a further modification to the algorithms as a result of the convergence tests findings. Finally, we present the results of the computing time tests of our algorithm against implementations of popular methods.

The problem set contains randomly generated bound-constrained QPs of dimensions dimensions 2-10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 800, 900 and 1000. There are 40 instances for each dimension adding up to 1080 examples. The problems were generated so that they always have a closed strictly feasible region and so that the solution lies on the boundary, i.e. at least one constraint is active. In addition, the problems are well scaled and well conditioned so that the feasible region is not much tighter on one dimension than in another and so that the paraboloid’s Hessian condition number is not too high. Investigating the properties of the method in an ill-conditioned context is left for future research.
V.1 Phase I Algorithm

The Phase I refers to the stage where we compute a feasible initial point $x_0$ in order to start the optimization procedure, which is called Phase II. Although there are optimization algorithms that can start with an infeasible initial point, most of them need to start with feasibility. Our family of optimization algorithms, being a sort of interior-point methods, need a strictly feasible initial point, i.e. $x_0 : Ax_0 > b$. In fact, a more or less central initial point will be ideal.

A suitable method to obtain a strictly feasible initial point is solving an optimization problem that maximizes the distance to the constraint functions. The formulation of the problem for a set of linear inequality constraints is:

$$\begin{align*}
\text{maximize} & \quad s \\
\text{subject to} & \quad a_i^\top x_0 - b_i \geq s \quad \text{for} \quad i = 1, \ldots, m \quad (V.1)
\end{align*}$$

where $s \in \mathbb{R}$. In this formulation we are actually maximizing the greatest distance to any constraint plane, thus making sure that if the set of constraints are strictly feasible, then the optimal point $x_0^*$ is as far away from the boundary as it can be. It goes without saying that precision is not critical to solve this problem once it has been proven that it is strictly feasible.

There are three possible outcomes for the formulation $V.1$

- $s^* > 0$: The problem has a strictly feasible solution $Ax_0^* > b$
- $s^* = 0$: The problem is feasible but not strictly feasible., i.e. here is at least one constraint $i$ such that $a_i^\top = b$.
- $s^* < 0$: The problem is infeasible.
This formulation was implemented to find the initial points for the problem sets. The same initial points were used for all optimization algorithms tested.

V.2 Interpolation Surface Function

The interpolation surface function takes as input the vector \( x \in \mathbb{R}^n \) and returns the value of the interpolation function at that point \( F_S(x) \), the gradient \( \nabla F_S(x) \), the Hessian matrix \( \nabla^2 F_S(x) \), and the index of the bounds that have been reached by the algorithm.

One important factor to take into account is that the weight functions of \( F_S(x) \) involve calculations with some positive power \( \mu \) of the inverse of the distances to the constraints. It is obvious that there will be loss of precision as \( d_i(x) \to 0 \), and eventually underflow will occur causing the algorithm to break down if we don’t take safeguards. Therefore we will set a reasonable bound on how close the iterates can approach the boundary in order to avoid this case. We will call this threshold \textit{boundaryTol}.

At the beginning of the calculations, the code checks for the smallest distance to the constraint planes, and once one of the variables has surpassed the threshold, the algorithm automatically returns the exact value at the point \( x \), i.e. \( F_S(x) = f_0(x) \). It then projects the gradient \( \nabla f_0(x) \) to the null space of the variables that reached the boundary, this is straightforward to do in the case where there are only bounds on the variables. In the case for general linear constraints, we would use the formula (III.31) that we derived for the case when \( x \) lies on the boundary instead of (III.17). In addition, since the Hessian of the interpolation formula becomes singular at the boundary, the identity matrix is returned as the Hessian matrix once \( x \) is inside the threshold to avoid breakdown of the method. The choice of the identity matrix when calculating the Newton direction yields a steepest descent direction.
The value of the threshold has a direct influence on the precision and convergence of the algorithm. First of all because it determines how close we can get to the boundary, where the solution is, therefore it cannot be too large as to hurt the desired precision. On the other hand, it can not be too small as to cause severe inaccuracy in the calculation of the gradient and Hessian due to the terms involving large powers of the distances $d_i(x)$.

After performing some tests with different values in part of the problem set, the value of the threshold parameter for all interpolation methods was chosen to be $8 \times 10^{-10}$. It goes without saying that the stopping condition parameter $xTol$ must be smaller than the boundary threshold parameter. The algorithm still makes large improvements on the function value after reaching the boundary as we will see next.

V.3 Implementation of the Optimization Algorithms

Algorithms 2 to 5 were implemented in the Matlab computer language in order to carry out computing time and convergence tests. Efficiency considerations were used when writing the codes such as vectorizing all operations so that there are no for loops in the code. In addition, we make use of the Matlab’s sparse matrices capabilities in order to take advantage of the sparsity of the constraint matrices. Nevertheless, no code parallelization was used besides what Matlab’s built-in commands might include. This is an area where our algorithms can still benefit from. In our optimization algorithms, the calculation of the interpolation surface’s value, gradient and Hessian matrix consume most of the computing resources. The calculation of these quantities is easily parallelizable since they involve combinations of independent values. For example, to obtain the interpolation function value we can calculate the projection values and weights in a parallel way and then combine them together.
In this section we solve some example problems with the four methods and propose a further modification for all algorithms based on the observations made.

V.3.1 Algorithm 2: Steepest Descent

There were no major modifications to algorithm 2. The gradient field of the interpolation surface leads readily to the solution from any strictly feasible initial point so the steepest descent method should work well if we make sure we don’t step out of the feasible region and we make sure that we do not terminate the algorithm before the solution is reached. Since we do not make use of the KKT conditions we have no way of making sure that we have arrived to the solution, however, in our experience the heuristics that we use as stopping criteria work well with the parameter values used.

The parameter values used for this algorithm were:

The parameters used were:

- \( \alpha_0 \leftarrow 1 \)
- \( xTol \leftarrow 10^{-12} \)
- \( fTol \leftarrow \epsilon_m \)
- \( gTol \leftarrow \epsilon_m \)
- \( maxIter \leftarrow 100n \)
- \( boundaryTol \leftarrow 8e - 10 \)

where \( \epsilon_m \) is the machine epsilon. The choice of \( \alpha_0 \), which is the initial step length for the backtracking procedure is somewhat arbitrary, it can be fixed or adjusted at every iteration but care must be taken so that it is not too small as to prevent the algorithm to take large enough steps. Because of the scaling of our problem set, we fixed this value to 1.
The $xTol$ parameter stops the algorithm when it is not taking steps of sufficient length, however, we must be careful when setting it since the sequence of steps that the algorithm takes are not monotonically decreasing as we will see later. Moreover, the $xTol$ value should be smaller than the boundary threshold $boundaryTol$.

The $gTol$ or gradient tolerance parameter stops the algorithm when the gradient is close enough to zero, since this indicates that we are near the solution or that we are in a vertex. The $fTol$ parameter stops the algorithm when there is no decrease in the function value. This is set to machine epsilon since we allow some steps to produce a small decrease in the function value but we don’t allow it to be zero or too close to zero since it can get cycled or require a very large number of iterations to terminate.

Figures V.1 show a couple of 2-D problems solved with the steepest descent algorithm. The image on the left shows a problem where the solution lies on a vertex of the feasible region. The image on the right shows an instance where the solution lies on one of the planes. In high dimensional cases, solutions normally lie on the intersection of a subset of planes but normally not on an $n$-Dimensional vertex or one only plane, so it can be thought of a combination of these two cases.

(a) Minimum is found on a vertex  (b) Minimum is found on a plane

Figure V.1: Example solutions of 2-D problems with steepest descent algorithm.
It is interesting to note in the case where the minimum lies on a plane how the iterates approach the boundary rather quickly before being close to the solution. This phenomenon causes some difficulties to the method. Figure V.2 shows the difference between the current objective value $F_S(x_k)$ and the optimum value $p^*$ as the algorithm progresses. When the solution lies on the intersection of two planes (Subfigure V.2a), which is a vertex, the decrease is clearly linear. When the solution lies on the plane (Subfigure V.2b), at the beginning the decrease is linear as well but with a slower rate of decrease until it is close to the neighborhood of the solution where the rate increases.

V.3.2 Algorithm [3]: BFGS Method

For the BFGS method, the same parameter values as the steepest descent method were used. The extra parameters for the BFGS method were set to:

- $nSD \leftarrow 5$

- $minCurve \leftarrow 1e - 8$
When the iteration counter is a multiple of the \( nSD \) parameter, the algorithm performs a steepest descent iteration. This helps improve convergence, for example, when the Hessian is not accurate or ill-conditioned. The \( \text{minCurve} \) parameter defines the lower limit of the curvature measure \( s_k^T y_k \), where if it is lower than the parameter, the update of the \( H_k \) matrix is not carried out since the formula would not guarantee the positive definiteness of the matrix.

Figure V.3 shows the same 2-D problems solved with the BFGS method. The image on the left shows a problem where the solution lies on a vertex of the feasible region.

(a) Minimum is found on a vertex  (b) Minimum is found on a plane

Figure V.3: Example solutions of 2-D problems with the BFGS algorithm.

Figure V.4b shows the convergence plot for the example when the minimum is on a single plane. It can be seen that the convergence is better than the same case with the steepest descent method in the first iterations. However, the rate of convergence starts to decrease as we approach the neighborhood of the solution. The Hessian approximation becomes ill-conditioned when close to the boundary. Quasi-Newton methods have difficulties when the Hessian approximation is ill-conditioned.

Figure V.4a shows the convergence plot for the example when the minimum is on a vertex. The convergence seems to show no improvement when compared to the steepest descent since it seems to be more or less linear for this case.
(a) Minimum is found on a vertex
(b) Minimum is found on a plane

Figure V.4: Convergence plot of the 2-D example problems solved with the BFGS algorithm

V.3.3 Algorithm 4: Damped BFGS Method

For the damped BFGS method, the same parameter values as the steepest descent and BFGS methods were used with the exception of the minCurve parameter since the update formula from the damped BFGS method already ensures that the Hessian stays positive definite.

Figure V.5 shows the same 2-D problems solved with the Damped BFGS method. It can be seen that the behavior of the method is similar to the normal BFGS method.

(a) Minimum is found on a vertex
(b) Minimum is found on a plane

Figure V.5: Example solutions of 2-D problems with the Damped BFGS method.
Figure V.6a shows the convergence plot for the example when the minimum is on a plane. As in the standard BFGS method, it can be seen that the convergence is better than with the steepest descent method in the first iterations but the convergence rate decreases as the matrix becomes ill-conditioned.

The convergence for the case when the minimum is on a vertex (V.6b) seems to show no improvement when compared to the steepest descent since it seems to be more or less linear for this case.

![Convergence plots](image.png)

(a) Minimum is found on a vertex  
(b) Minimum is found on a plane

Figure V.6: Convergence plot of the 2-D example problems solved with the Damped BFGS method

V.3.4 Algorithm 5: Newton’s Method

For the Newton’s method, the same parameter values as the steepest descent method were used. The extra parameters for the Newton’s method were set to:

- \( \text{initialSD} \leftarrow 4 \)
- \( \text{nSD} \leftarrow 5 \)

The \( \text{initialSD} \) parameter defines the number of steepest descent steps we take at the beginning of the method. This is to ensure that we have left the negative curvature area of the interpolation surface. In our experience, four steepest descent
iterations were sufficient in all test instances. The $nSD$ parameter, as in the quasi-
Newton methods defines how often we perform a steepest descent step in order to aid
convergence to the solution.

(a) Minimum is found on a vertex  
(b) Minimum is found on a plane

Figure V.7: Example solutions of 2-D problems with Newton’s method.

(a) Minimum is found on a vertex  
(b) Minimum is found on a plane

Figure V.8: Convergence plot of the 2-D example problems solved with Newton’s method

When the solution lies on the plane (Subfigure V.8b) the rate of convergence
becomes rapidly quadratic as expected after the Newton’s method kicks in at iteration
No. 5. The convergence for the case when the minimum lies on the vertex (Subfigure
V.8a) seems to be slightly better than the other methods since the number of iterations
decreases, not only in this example, but in many. We will later perform more tests to confirm this.

Newton’s method uses the information of the second derivatives so it is able to make a better guess of the next iterate. This is why it seems to provide directions that are more accurate, at least in the case when the optimum lies on the plane. Moreover, the Newton’s method is less sensitive to ill-conditioning of the Hessian matrix and we are using the exact formula to calculate it instead of an approximation as in the quasi-Newton case. Nevertheless, as we approach the boundary, numerical inaccuracies become more severe which can lead to erratic behavior.

![Figure V.9: Close up of Figure V.7b](image)

Now that we know how the methods behave in low dimensions we can begin to study their behavior with more variables. We begin with by solving a 30-D problem with the steepest descent method. Figure V.10a shows the plot of the optimality gap against the iterations just as in the last section. We notice that it takes a long time to converge to a reasonable accuracy and that there seem to be some steep jumps. The right side figure, shows the plot of the step length 2-norm against the iteration count. In this case, the reason for the jumps becomes clear, the algorithm is approaching
the boundary very soon, when the distance between the iterate $x_k$ and any constraint is less than the parameter $boundaryTol$, we begin projecting the gradient of the original objective function to the null space of the corresponding constraint as in formula \[\text{II.31}\]. The descent continues along the boundary until the threshold of another constraint is reached and the gradient is projected to the null space of these two constraints and the process continues until convergence. This is the reason for the jumps in the step length plot.

Moreover, this is the reason why the parameter $xTol$ should be smaller than the $boundaryTol$ parameter that defines the threshold of the constraints.

(a) Optimality gap plot
(b) Step length plot

Figure V.10: Convergence plot of a 30-D example problems solved with the steepest descent method

Figure V.11 shows the results of the same 30-D problem solved with Newton’s method. As we can see, the iterates still approach the boundary too rapidly, although there is a slight improvement in the sense that the number of jumps and iterations is slightly less.

We recall that the Hessian of the interpolation function becomes increasingly ill-conditioned as $x$ approaches the boundary. A safeguard within the interpolation function implementation returns the identity matrix as the Hessian when the threshold is surpassed. Therefore, Newton’s method as it was so far described becomes
equivalent to the steepest descent method after we reach the boundary for the first time. This is the reason for the modification on the algorithms that we propose in the next section.

(a) Optimality gap plot  
(b) Step length plot

Figure V.11: Convergence plot of the 30D example problems solved with Newton’s method

Finally, a similar conclusion is reached for the quasi-Newton methods implementation since we have the same safeguard for the Hessian matrix approximation and they do not improve this problem.

V.4 Solving a sequence of problems in a subdimensional space

As we have seen, once the iterates reach the boundary, the Hessian matrix or its approximation in the case of quasi-Newton methods, becomes ill-conditioned thus we cannot make use of that information in the full dimensional space. Therefore, we propose a modification to all four algorithms were once the algorithm steps into the threshold of one or more constraint planes, we fix the values of the corresponding variables and proceed to solve the problem in the subdimensional space. Thus we are always making use of the interpolation surface.
One important caveat that we need to consider, is that after finding new active constraints, we need to solve the *unconstrained subproblem*, were we fix the values of the variables corresponding to the active constraints and disregard all other constraints. If the point obtained is feasible in the original problem, the algorithm terminates since we have found the solution, otherwise, we continue solving the subproblem until we find another active constraint or one of the termination conditions is satisfied.

In the case of a quadratic objective function, it is fairly easy to find an expression to calculate the optimum of the *unconstrained subproblem*. We define \( x_K \) as the "known" variables, that is, the ones that correspond to the active constraints we have encountered so far, and \( x_U \) as the "unknown" variables. We can assume that the variables have a convenient order without loss of generality, and divide the matrix \( P \) of the paraboloid and the vector \( q \) in blocks that correspond to \( x_K \) and \( x_U \):

\[
\begin{align*}
  f(x) &= \frac{1}{2} \begin{bmatrix} x_U^\top & x_K^\top \end{bmatrix} \begin{bmatrix} P_{UU} & P_{UK} \\ P_{KU} & P_{KK} \end{bmatrix} \begin{bmatrix} x_U \\ x_K \end{bmatrix} + \begin{bmatrix} q_U^\top & q_K^\top \end{bmatrix} \begin{bmatrix} x_U \\ x_K \end{bmatrix} + r \\
  &= \frac{1}{2} x_U^\top P_{UU} x_U + \frac{1}{2} x_K^\top P_{KK} x_K + \frac{1}{2} x_U^\top P_{UK} x_K + q_U^\top x_U \\
  &\quad + \frac{1}{2} x_K^\top P_{KU} x_U + q_K^\top x_K + r
\end{align*}
\]

\[
\nabla_{x_U} f(x) = P_{UU} x_U + P_{UK} x_K + q_U \tag{V.3}
\]

Equating the derivative to zero and solving for \( x_U \) yields:

\[
x_U = -P_{UU}^{-1}(P_{UK} x_K + q_U) \tag{V.4}
\]

Algorithm 6 shows the pseudo code for the steepest descent algorithm including
the projection of the problem to subdimensional spaces. It is basically the same
procedure as algorithm 2 with the difference that after each iteration, the algorithm
checks if there are any new variables within the boundary threshold. If the test is false,
it continues normally, otherwise, the index of the known variables is updated and the
unconstrained problem for the unknown variables is solved to check for optimality.
The variations for the two quasi-Newton method and Newton’s method involve the
same modification and were implemented as well.

Note the change in the notation for algorithm 6. The superscript $k$ denotes the
iteration count whereas the subscripts $K$ and $U$ denote the indices that belong to
the set of known and unknown variables respectively. We will denote those sets as
$K$ and $U$. In addition, $A_U$ is represents the matrix of constraints $A$ where we have
eliminated the columns corresponding to the known variables.

\begin{algorithm}
\caption{Steepest descent method for the interpolation surface}
\begin{algorithmic}
\Require $\alpha_0 > 0, x_0 : Ax_0 \geq b, xTol > 0, gTol \geq 0, fTol > 0, maxIter \in \mathbb{N}$
\State $k \leftarrow 0$
\State $U^k \leftarrow \{1, \ldots, n\}$
\State $K^k \leftarrow \emptyset$
\State $F_S(x_{-1}) \leftarrow \infty$
\While{$k \leq maxIter$ and $\|\nabla F_S(x_k)\|_2 > gTol$ and $F_S(x_{k-1}) - F_S(x_k) > fTol$}
\State $\langle U^k, K^k \rangle = UpdateKnownVariables(x_k^U, x_k^K, A, b)$
\If{$|K^k \setminus K^{k-1}| \geq 1$}
\State $x_{unc} = -P_{U^k}^{-1}(P_U x_K^k + qU)$
\If{$A_U x_{unc} - b \geq 0$}
\State $x^* = x_K^k \cup x_{unc}$
\State \textbf{break}
\EndIf
\EndIf
\State $p_k \leftarrow -\nabla F_S(x_k)$
\State $p_k \leftarrow p_k / \|p_k\|_2$
\State $\alpha_k \leftarrow BacktrackingLineSearch(x, p, \alpha_0)$
\If{$\|\alpha_k p_k\|_2 < xTol$}
\State \textbf{break}
\EndIf
\State $x_{k+1} \leftarrow x_k + \alpha_k p_k$
\State $k \leftarrow k + 1$
\EndWhile
\end{algorithmic}
\end{algorithm}
V.5 Performance Tests

At this point we have a total of five different algorithms based on the interpolation surface:

1. Steepest descent: Algorithm 2

2. Subdimensional steepest descent: Algorithm 6

3. Subdimensional BFGS: Algorithm 3 with the modification described in section V.4.

4. Subdimensional Damped BFGS: Algorithm 4 with the modification described in section V.4.

5. Subdimensional Newton’s method: Algorithm 5 with the modification described in section V.4.

We compared these methods against the four Matlab’s Optimization Toolbox V6.1 (R2011b) algorithms included in the \textit{fmincon} function for general convex optimization problems:

- \textit{Active-set},

- \textit{Interior-point},

- \textit{Trust-region-reflective},

- \textit{SQP},

as well as with the three algorithms included in the \textit{quadprog} function, which are specifically designed for quadratic problems:

- \textit{Active-set},
• *Interior-point-convex*,

• *Trust-region-reflective*.

Moreover, we implemented a couple of standard methods for convex optimization, an active-set method for convex QPs as described in Chapter 16 of [29] and a log-barrier method as described in Chapter 11 of [20].

### V.5.1 Precision

All problems in the test set were first solved via the Matlab’s active-set algorithm from the `quadprog` function. Because of their nature, active-set methods guarantee a very high precision since they find the exact set of constraints that are active at the solution. Moreover, the active-set method from the `quadprog` is more reliable than that of that of Matlab’s `fmincon` function since it is specifically designed for QPs.

Due to time considerations, only the problem sets up to dimension 600-D were then solved by all five interpolation surface methods methods with the parameter values previously described. The larger problem sizes were not solved by the subdimensional methods because of the computing time growth as we will see in the next section.

Where applicable, the parameter `TolFun` of the Matlab methods, which is the stopping criteria parameter for the objective function value, was set to $2 \times 10^{-8}$ out of fairness considerations since it is a similar value to the mean precision obtained by our interpolation methods. In addition, the outer loop tolerance for the optimality gap in the standard log-barrier method was set to the same value.

Table [V.1](#) shows the number of "failed cases" for each of the methods. For the sake of comparison, these numbers correspond to problem sets of up to 600-D. As we remember, higher dimensional problems were not solved by the subdimensional methods due to time considerations. Moreover, we define a "failed case" as a case where the precision was lower than $2 \times 10^{-5}$. We think three orders of magnitude
away from the required precision is a suitable threshold to make this definition. This is necessary in order to be able to make a comparison between the methods but it does not necessarily mean that the method cannot solve the problem. In our experience, most of the "failed cases" achieved a suitable precision after adjusting the precision parameters. Therefore, the number of "failed cases" should not be taken as a measure of the quality of the methods. Furthermore, we show the mean precision of each of the methods (without taking into consideration the failed cases).

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>FAILED CASES</th>
<th>MEAN PRECISION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolation/Steepest descent</td>
<td>2</td>
<td>2.55e-08</td>
</tr>
<tr>
<td>Interpolation/Steepest descent/Subdimension</td>
<td>9</td>
<td>2.41e-08</td>
</tr>
<tr>
<td>Interpolation/BFGS-Subdimension</td>
<td>2</td>
<td>2.84e-08</td>
</tr>
<tr>
<td>Interpolation/Damped BFGS-Subdimension</td>
<td>2</td>
<td>1.89e-08</td>
</tr>
<tr>
<td>Interpolation/Newton-Subdimension</td>
<td>8</td>
<td>4.27e-08</td>
</tr>
<tr>
<td>Matlab/Fmincon/Active-set</td>
<td>0</td>
<td>2.70e-09</td>
</tr>
<tr>
<td>Matlab/Fmincon/Interior-point</td>
<td>0</td>
<td>4.35e-07</td>
</tr>
<tr>
<td>Matlab/Fmincon/Trust-region-reflective</td>
<td>0</td>
<td>6.28e-14</td>
</tr>
<tr>
<td>Matlab/Fmincon/sqp</td>
<td>9</td>
<td>5.88e-07</td>
</tr>
<tr>
<td>Matlab/Quadprog/Active-set</td>
<td>0</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>Matlab/Quadprog/Interior-point-convex</td>
<td>26</td>
<td>7.25e-07</td>
</tr>
<tr>
<td>Matlab/Quadprog/Trust-region-reflective</td>
<td>3</td>
<td>6.31e-07</td>
</tr>
<tr>
<td>Standard/Active-set</td>
<td>0</td>
<td>2.65e-14</td>
</tr>
<tr>
<td>Standard/Log-barrier</td>
<td>0</td>
<td>3.62e-08</td>
</tr>
</tbody>
</table>

Table V.1: Number of failed cases per algorithm out of the first 920 problems in the set (Dimensions 2-D to 600-D). For the mean precision calculation, the failed cases have been left out.

V.5.2 Computing Time

The next step in the performance tests is comparing the computing times for the algorithms. These running time tests do not intend to be a scientific comparison between the methods, nor we intend to reach a definitive evaluation of which method is better. All existing methods have advantages and disadvantages with respect to the others and even today, there is no agreement on which of the standard methods is
the best one to use. Moreover, the efficiency of the software depends not only on the theoretical properties of the methods but also on the way it was implemented, the choice of programming language, the parameter values, among other factors. The goal of the running time tests is to make a rough comparison between our interpolation surface methods and other popular methods in a more or less fair scenario and to evaluate the potential of the first ones.

The tests were carried out in a Macintosh workstation with two 2.4 GHz Quad-Core Intel Xeon processors running the Mac OS 10.6.7 operating system and the 64-bit R2011b (7.13.0.564) Matlab version. As we stated before, there are 27 sets with different dimensions ranging from 2 to 1000. Each set of a particular dimension consists of 40 problems. Due to computing time considerations, for the subdimensional interpolation methods, we only solved the problem sets of up to 600-D.

For the computing time measurements we used the `tic` and `toc` Matlab functions which provide the most reliable and precise elapsed time performance measurement available in Matlab. Since the R2006b version of Matlab, these functions are based on the wall clock time instead of the local time provided by the Operating System [45]. The elapsed time measurements were averaged for each problem set with a particular problem size to reduce the effects of randomness.

Figure V.12 shows the running time results for each of the five interpolation surface methods in a logarithmic scale. Although the methods seem to have similar computing times for low dimensions, it is clear that the steepest descent method without the modification proposed in section V.4 is the most efficient choice. The reason behind this is that the gradient and Hessian matrix of the interpolation function are very expensive to compute. In the subdimensional methods we need to calculate them continuously whereas in the simple steepest descent method the calculation of the gradient is greatly simplified once we reach the boundary after a few iterations. In addition, Newton’s method and the Damped BFGS method involve the solution of a
Moreover, contrary to what we would expect, the use of quasi-Newton and Newton’s methods do not seem to bring any significant improvement in terms of the number of iterations which could justify the additional expense of having to calculate the gradient and Hessian matrix or its approximation throughout the solution process. As we have said before, the fact that the Hessian matrix becomes severely ill-conditioned near the boundary of the feasible region and the severe numerical inaccuracies involved in its calculation in the neighbourhood of the constraint planes undermine the usual capacity of these methods to achieve super linear or quadratic convergence rates.

Figure V.13 shows the average number of iterations for each problem set according
to the number of dimensions. All methods behave roughly the same in terms of the number of iterations and none of them seem to have a clear advantage. However, as we noted before, the subdimensional methods involve more computations.

Figure V.13: Average number of iterations performed by the interpolation surface algorithms. Quasi-Newton and Newton’s methods do not improve the number of iterations performed by the steepest descent implementations.

After we reach the conclusion that the interpolation surface method using steepest descent described in algorithm 2 seems to be the most efficient option, we compare its performance to that of the other implementations of popular methods enumerated at the beginning of the section. Figure V.14 shows the average computing times of these implementations together with that of the interpolation with steepest descent method for problems with dimensions of up to 1000-D.
Figure V.14: Average running times of several popular methods implementations compared with the steepest descent interpolation method. Our interpolation method with steepest descent seems to be at least competitive with some of the methods.

Our method, while not better than most of the other implementations in the problem sizes that we have tested, seems to be at least competitive with some of them. In particular, it is clearly faster than the trust-region-reflective algorithm of Matlab’s fmincon function for larger problem sizes. It also seems to defeat the standard implementations of the active-set method and the log-barrier method. Moreover, it seems to have a very similar behavior to the rest of the active-set methods for problems
with a high number of dimensions as well. This is to be expected since the algorithm behaves partially like an active-set method, in particular once the iterates are close to the boundary. These are definitely encouraging results.

![Figure V.15: Zoom in on Figure V.14](image)

**V.5.3 Empirical Orders of Growth**

Probably one of the most important measures of effectiveness of an optimization method is its time complexity. The time complexity of an algorithm tells us the asymptotic behavior of the method as the number of dimensions increases. This is important since it lets us know whether an algorithm is tractable or not, i.e. if
an algorithm has exponential time complexity in the average case, then it will only be useful for very small problem sizes. On the other hand, if it is proven to have polynomial time complexity with a low power constant, then it has the potential to be scaled to a larger number of dimensions.

However, if an algorithm is shown to have worst-case exponential time complexity, it does not necessarily mean that it will not be able to solve large problems in practice. The Simplex method is a good example of this, since it has been shown to have exponential time complexity \[16\] but it normally exhibits a polynomial time complexity in practical problems.

To prove the theoretical time complexity of an optimization method can be a very difficult task itself and is out of the scope of this thesis. Therefore, we present the results for the empirical complexity exhibited by our methods and the other implementations during the running time tests for the sake of comparison.

For the empirical calculation, we have decided to use average computing time of the seven largest problem sets, i.e. 400-D to 1000-D, for all methods except for the subdimensional methods for which only the data corresponding to dimensions 400-D to 600-D is available. Data from lower-dimensional problem sets was not used since the Figures V.14 and V.15 show that the slope of the curves in the logarithmic scale is still changing for lower dimensions.

Assuming the algorithms exhibit polynomial time complexity in the number of dimensions, \( O(n^a) \), the quantity shown in the table is the power constant \( a \). We make use of linear least squares to fit a line in the logarithmic scale, the slope of that line is the constant \( a \) since it is equivalent to the exponent in the normal scale.
Table V.2: Average empirical complexity of the tested methods measured at dimensions 400-D to 1000-D, except for the subdimensional methods where it was measured only at the 400-D and 600-D due to computing time restrictions (*). The last column shows the growth order averaged over these two problem sizes.

As expected, the subdimensional interpolation surface algorithms show a very large complexity in the test cases. However, our interpolation surface method with steepest descent exhibits a very good complexity of order $O(n^{2.52})$. Actually, it is only beaten by three algorithms: the two Matlab’s interior-point methods and the trust-region-reflective algorithm of the Matlab’s quadprog function. This shows that the method definitely has potential and is competitive with the standard implementations of popular methods.
Chapter VI

Concluding Remarks

In this thesis we describe a new family of methods for solving Convex Optimization problems subject to linear constraints, which is based on the substitution of the original problem by an interpolation surface whose gradient field contains information leading to the optimum. In particular, we implemented and tested several versions of the method for the Quadratic Programming case with bounds on the variables.

The preliminary results are encouraging since our implementation of the method using the steepest descent technique seems to be competitive even with commercial implementations of several popular methods in terms of running time. In addition, the empirical complexity of the method seems to be superior to many of the other implementations that we compared against.

In Chapter 3, we introduced the key idea of our method, which is the generalization of the Shepard’s interpolation formula for optimization problems with linear constraints and we proved the favourable properties of the surface such as the interpolation of the objective function values at the constraint planes and the convergence of the gradient field to the null space of the constraint planes at the boundary.

There is still room for improvement in the implementation of the interpolation function. One of the main disadvantages of Shepard’s formula is that it leads to severe
numerical inaccuracies when the distances go to zero. For instance, the calculation of
the gradient involves terms of the form $d_i(x)^{2\mu}$, this means that for a choice of $\mu = 2$
and working with double precision in a machine using the *IEEE 754* standard, after
the distances decrease beyond $1 \times 10^{-4}$, numerical inaccuracies will start taking place
since the machine epsilon is approximately $2 \times 10^{-16}$. In this situation, the order in
which we make the operations is very important. We can also see that increasing the
value of $\mu$ is not such a good idea since it will make the numerical inaccuracies worse.

In Chapter 4, we described several descent strategies that use the information of
the interpolation surface to find the solution of the optimization problem. In Chapter
5, we presented the results of the implementations, concluding that the methods
that use curvature information such as the quasi-Newton methods and the Newton’s
method do not bring any benefit in this particular case. We demonstrated that the
simple steepest descent technique without solving the sequence of subdimensional
problems seems to be the best choice so far.

Up to this point, we have only tried traditional descent methods used normally
for unconstrained optimization problems, future research could involve developing an
ad-hoc method that takes full advantage of the interpolation surface and that is able
to produce better rates of convergence.

So far, we have tested the potential of this new type of convex optimization meth-
ods in Quadratic Programs with bounded variables, future work can also involve
exploring the potential of the method in other types of problems. Early tests show
that our method works well with quadratic problems with general linear constraints,
however, due to the nature of the operations involved in the calculation of the gradient
interpolation function for general linear constraints, it is very difficult to produce an
efficient implementation that we could compare against a commercial package. This
would require the use of three-dimensional matrices for code vectorization which are
unmanageable for large problem sizes or the use of *for loops* which are very inefficient
in Matlab’s environment. A competitive implementation for problems with general linear constraints should be carried out in a more efficient programming language such as C. With this work, we expect to encourage discussion from our peers in these and other directions.
APPENDICES
Appendix A

Matlab codes

In this Appendix we include the source code of the five algorithms mentioned in Chapter 5 which include:

1. Interpolation surface optimization using steepest descent.

2. Interpolation surface optimization by solving a sequence of subdimensional problems using steepest descent.

3. Interpolation surface optimization by solving a sequence of subdimensional problems using the BFGS method.

4. Interpolation surface optimization by solving a sequence of subdimensional problems using the damped BFGS method.

5. Interpolation surface optimization by solving a sequence of subdimensional problems using the Newton’s method.

The interpolation surface optimization using steepest descent makes use of the backtracking line search and the interpolation surface functions included.

The methods that solve a sequence of subdimensional problems share a different version of the backtracking line search and the interpolation surface functions which
are also included in the corresponding section. For the codes to function, all that is needed is to load the variables $P$, $q$ and $r$ corresponding to the quadratic function and $A$ and $b$ corresponding to the linear constraints (in this case, the bounds), the standard that we assume for the constraints is $Ax \geq b$. Moreover, if $A$ should be defined as a sparse matrix. These variables should be declared as global variables and an initial strictly feasible point should be provided.

### A.1 Interpolation Surface Optimization Using Steepest Descent

**Backtracking Line Search Function**

```matlab
function [alpha]=backtrackLineSearch(f,x,p,alpha0)
%
% BACKTRACKLINESEARCH. Calculates the alpha_k factor such that
% f(x_k + alpha_k * p,k) remains feasible and complies with the Armijo
% condition, starting with alpha0 and decreasing the value until the
% condition is met.
%
% [alpha]=BACKTRACKLINESEARCH(f,x,p,alpha0)
%
% Calls the following global variables:
% - A: Matrix of normals for the bound constraints. (A*x\geq b)
% - b: Vector of intercepts for the bound constraints. (A*x\geq b)
% - bBacktracking: Flag to mark that Backtracking is taking place.
%
% Input:
% - f: Function handle of the objective function.
% - x: Current iterate.
% - p: Direction of minimization.
% - alpha0: Initial value for scalar factor alpha.
%
% Output:
% - alpha: Scalar factor for the direction of minimization.
%
% Global variables
% global A, global b, global bBacktracking
%
% Activate global backtracking flag for the interpolation surface
% function
% bBacktracking=1;
%
% Parameters
% rho=0.5;%Decrement factor for t (step size)
% c1=10^-4;%Decrement factor for line slope
% alpha=alpha0;
%
% Ensure feasibility
while (min(A*(x+alpha*p)-b)<0)
    alpha=rho*alpha;
end
[fCurrent gCurrent]=f(x);
next=fix(x+alpha*p);
while (fNext>fCurrent+c1*alpha*(gCurrent'*p))
    alpha=rho*alpha;
    fNext=f(x+alpha*p);
```
% Break when the alpha is too small or when the function does not
% change anymore within the range.
if (alpha<10*eps || abs(fNext−fCurrent)<eps)
    break;
end
bBacktracking=0;
end

Interpolation Surface Function

function [fsurf gsurf bActiveConstr]=InterpSurfQP_bounds(x)

% INTERPSURFQP_BOUNDS. Generalized Shepard’s formula for Quadratic Programs subject
% to bounds on the variables:
% minimize (1/2)∗x'∗P∗x+q'∗x+r
% x
% subject to lb < x ≤ ub
% [fsurf gsurf bActiveConstr harm] = INTERPSURFQP_BOUNDS(x)

% Calls the following global variables:
% - P: Hessian matrix of quadratic function
% - q: First order coefficients vector of quadratic function
% - r: Offset term of quadratic function
% - A: Matrix of normals for the bound constraints. (A∗x≥b)
% - b: Vector of intercepts for the bound constraints. (A∗x≥b)
% - nDim: Number of dimensions
% - boundaryTol: Threshold parameter for the constraint planes
% - bBoundary: Flag to mark presence of active constraints at last iterate.
% - (0 -> no active constraints).
% - bBacktracking: Flag to mark that Backtracking is taking place.

% Input:
% - x: Point to interpolate

% Output:
% - fsurf: Interpolation surface value.
% - gsurf: Interpolation surface gradient.
% - bActiveConstr: Flag to mark the presence of active constraints at x.
%   (0 -> no active constraints).
% - hsurf: Hessian of the interpolation formula.

% Global variables
global A, global b, global P, global q, global r, global nDim,
global boundaryTol, global bBoundary, global bBacktracking

% Power value of interpolation formula
mu=2;

% Quadratic function definition
F0= @(x) (1/2)*sum((x’.∗P).’∗x)+q’∗x+r;

% Distance to all constraints.
D=abs(A∗x−b);

% Index of constraints where the iterate is beyond the threshold.
idxNearConstraint=abs(D)>(boundaryTol);

% Calculate interpolation surface value when iterate is not in the threshold
% of any constraint
if (all(~idxNearConstraint))
    bActiveConstr=0;
    % Calculate interpolation function value
    nPoints=length(D);
    % Weights numerator.
end
102

Wnum=(D) .ˆ −mu;

%Projection values
xProj=x(:,ones(nPoints,1))−D(:,ones(nDim,1))'.∗A';
fi=F0(xProj);

%Weights denominator
Wden=sum(Wnum);

%Calculate surface value
fsurf=fi.(Wnum/Wden);

%Calculate gradient vector
if (nargout>1)
    Dml=−mu.∗(D) .ˆ(−(mu+1));
    %Calculate first term of gradient
    Df1a=((Dml)/Wden).∗fi ;
    Dsum=((A').(Dml));
    Df1b=((Dsum(:,ones(nPoints,1))/Wden).∗((Wnum/(Wden)).∗fi ));
    Df1=Df1a−Df1b;
    %Calculate second term of gradient
    Df0=P∗xProj+q(:,ones(nPoints,1));
    Df0(logical(A'))=0;
    Df2=Df0.(Wnum/Wden);
    gsurf=Df1+Df2;
end

%If iterate is beyond boundary threshold.
else
    %Flag for active constraint
    bActiveConstr=1;
    %Define a second, smaller boundary threshold. This new threshold is
    %necessary when backtracking is being done near the main boundary
    %threshold and it is designed to avoid too short steps when changing
    %the formula for the fsurf calculation.
    idxNearConstraint2=D≤boundaryTol∗1e−5;
    if (all(~idxNearConstraint2)&&bBoundary==0&&bBacktracking==1)
        nPoints=length(D);
        Wnum=(D) .ˆ −mu;
        %Projection values
        xProj=x(:,ones(nPoints,1))−D(:,ones(nDim,1))'.∗A';
        fi=F0(xProj);
        Wden=sum(Wnum);
        %Calculate surface value
        fsurf=fi.(Wnum/Wden);
    else
        %Return the exact value at x.
        fsurf=F0(x);
    end
    if (nargout>1)
        %Project the gradient to the planes of the active constraints
        vecProjection=all(~A(idxNearConstraint,:)',2);
        gsurf=vecProjection.∗(P∗x+q);
    end
end

A.1.1 Steepest Descent Function

function [x,fval,exitflag,nIter,x_history,Δ History]=...
% Sets the following global variables:
% - nDim: Number of dimensions
% - boundaryTol: Threshold parameter for the constraint planes
% - bBoundary: Flag to mark presence of active constraints at last iterate.
% (0 -> no active constraints).

% Input:
% - f: Function handle of the interpolation surface
% - x0: Initial point
% - options: Structure for defining parameters
%   - options.alpha0 (Default: 1)
%   - options.xTol (Default: 1e-12)
%   - options.fTol (Default: eps)
%   - options.gTol (Default: eps)
%   - options.boundaryTol (Default: 8e-10)
%   - options.maxIter (Default: 100*nDim)

% Output:
% - x: Solution point
% - fval: Minimum value
% - exitflag: Cause of exit
%   - 1 Step length too small
%   - 0 Maximum number of iterations reached
%   - -1 Gradient norm is too small
%   - -2 Decrease in function value is too small
% - nIter: Number of iterations
% - x_history: Matrix of recorded iterates
% - ∆ History: Vector of recorded step lengths

%Global variables
global boundaryTol, global nDim, global bBoundary
nDim=length(x0);

%Set parameters
if (nargin<3)
    alpha0=1;
    xTol=1e-12;
    fTol=eps;
    gTol=eps;
    boundaryTol=8e-10;
    maxIter=100*nDim;
else if (nargin==3)
    alpha0=options.alpha0;
    xTol=options.xTol;
    fTol=options.fTol;
    gTol=options.gTol;
    boundaryTol=options.boundaryTol;
    maxIter=options.maxIter;
end

%Initialize variable to record iterates
x_history=zeros(nDim,maxIter);

%Initialize variable to record step sizes
∆ History=zeros(maxIter,1);
x_k=x0;
[f_k g_k bBoundary]=f(x_k);
%Dummy value in order to enter first while loop
f_kOld=Inf;

%Loop counter
k=1;
while (k<=maxIter&&\norm(g_k)>gTol&&(f_kOld-f_k)>fTol)
    %Steepest descent direction:
    p_k=-g_k;
    %Normalize if possible
    norm_p_k=\norm(p_k);
    if (norm_p_k>eps)
        %...
104

\[ a_k = \text{backtrackLineSearch}(f, x_k, p_k, \alpha_0); \]
\[ s_k = a_k \cdot p_k; \]
\[ \Delta_k = \|s_k\|; \]
\[ \%\text{Break if step lengths is too small or if direction yields 0-vector.} \]
\[ \text{if } (\Delta_k < \text{xtol} \text{ and } \|p_k\| = 0) \]
\[ \text{break;} \]
\[ \%\text{Update iterate} \]
\[ x_k = x_k + s_k; \]
\[ \%\text{Update function and gradient values} \]
\[ f_{kOld} = f_k; \]
\[ bBoundaryOld = bBoundary; \]
\[ [f_k, g_k, bBoundary] = f(x_k); \]
\[ \%\text{Set Infinity for function value at k-1 when we pass the boundary} \]
\[ \%\text{threshold for the first time to avoid early stopping due to change} \]
\[ \%\text{of formula for function value.} \]
\[ \text{if } (bBoundary \neq bBoundaryOld) \]
\[ f_{kOld} = \text{Inf}; \]
\[ \%\text{Record iterate and step length} \]
\[ x_{history}(; , k+1) = x_k; \]
\[ e_k \text{; History}(k) = \Delta_k; \]
\[ k = k+1; \]
\[ \%\text{Record stopping criteria used} \]
\[ \text{if } (k > \text{maxIter}) \]
\[ \text{exitflag} = 0; \]
\[ \text{elseif } (\|g_k\| \leq gTol) \]
\[ \text{exitflag} = -1; \]
\[ \text{elseif } (f_{kOld} - f_k) \leq fTol \]
\[ \text{exitflag} = -2; \]
\[ \text{else} \]
\[ \text{exitflag} = 1; \]
\[ \text{end} \]
\[ x = x_k; \]
\[ fval = F0(x); \]
\[ nIter = k - 1; \]
\[ x_{history}(; , k+1) = x_k; \]
\[ x_{history}(; , k+2:end) = []; \]
\[ e_k \text{; History}(k:end) = []; \]
\[ \text{end} \]

A.2 Methods That Solve a Sequence of Subdimensional Problems

Backtracking Line Search Function

function [alpha] = backtrackLineSearch_SS(f, x, p, alpha0)

%BACKTRACKLINESEARCH_SS. Calculates the alpha_k factor such that
% f(x_k + alpha_k * p_k) remains feasible and complies with the Armijo
% condition, starting with alpha0 and decreasing the value until the
% conditions are met. This function is to be used with the algorithms:
% Interp_SpestDescent_SS, BFGS_SS, DampedBFGS_SS and Newton_SS.
% 
% [alpha] = BACKTRACKLINESEARCH_SS(f, x, p, alpha0)
% 
% Calls the following global variables:
% - AUnk: Matrix of normals for the bound constraints corresponding to
105

% the unknown variables. (A\times g\geq b)
% b.Unk: Vector of intercepts for the bound constraints corresponding to
% the unknown variables. (A\times g\geq b)
% idxKnown: Index of known variables, corresponding to the active
% constraints
% Input:
% f: Function handle of the objective function.
% x: Current iterate.
% p: Direction of minimization.
% alpha0: Initial value for scalar factor alpha.
% Output:
% alpha: Scalar factor for the direction of minimization.

% Global variables
global A.Unk, global b.Unk, global idxKnown
% Parameters
rho=0.5; % Decrement factor for t (step size).
cl=10^{-4}; % Decrement factor for line slope.
alpha=alpha0;
% Ensure feasibility
while (min(A.Unk\times (x(-idxKnown)+alpha*p)-b.Unk)<0)
    alpha=rho*alpha;
end
[f.Current g.Current]=f(x);
x.Next=x;
x.Next(-idxKnown)=x.Next(-idxKnown)+alpha*p;
f.Next=f(x.Next);
while (f.Next>f.Current+cl*alpha*(g.Current*p))
    alpha=rho*alpha;
x.Next=x;
x.Next(-idxKnown)=x.Next(-idxKnown)+alpha*p;
f.Next=f(x.Next);
end

Interpolation Surface Function

function [fsurf gsurf bActiveConstr hsurf]=InterpSurfQP_SS_bounds(x)

% INTERPSURFQP_SS_BOUNDS. Generalized Shepard's formula for Quadratic
% Programs subject to bounds on the variables.
% This function is to be used with the algorithms:
% Interp_SteepestDescent_SS, BFGS_SS, DampedBFGS_SS and Newton_SS.
% Calls the following global variables:
% P: Hessian matrix of quadratic function.
% q: First order coefficients vector of quadratic function.
% r: Offset term of quadratic function.
% nDim: Number of dimensions.
% nDim_Unk: Number unknown variables at the current iteration.
% boundaryTol: Threshold parameter for the constraint planes.
% bBoundary: Flag to mark presence of active constraints at last iterate.
% (0 -> no active constraints).
% bBacktracking: Flag to mark that Backtracking is taking place.
% idxKnown: Index of known variables, corresponding to the active constraints.
% A_Unk: Matrix consisting of the rows and columns of A corresponding to unknown variables at the current iteration.
% b_Unk: Vector consisting of elements of b corresponding to unknown variables at the current iteration.

% Input:
% x: Point to interpolate.

% Output:
% fsurf: Interpolation surface value.
% gsurf: Interpolation surface gradient.
% bActiveConstr: Flag to mark the presence of active constraints at x. (0 -> no active constraints).
% hsurf: Hessian of the interpolation formula.

% Global variables
global P, global q, global r, global nDim, global A_Unk, global b_Unk, global nDim_Unk

% Power value of interpolation formula
mu=2;

% Quadratic function definition
F0=@(x) (1/2)*sum((x'*P)' *x) + q' * (x) + r;

% Distance to all constraints.
D=abs(A_Unk * x (¬idxKnown) − b_Unk);
% Index of constraints where the iterate is beyond the threshold.
idxNearConstraint= D ≤ boundaryTol;
% Calculate interpolation surface value when iterate is not in the threshold of any constraint
if (all(¬idxNearConstraint))
    bActiveConstr=0;

% Interpolation value
nPoints=length(D);
% Weight's numerator.
Wnum=(D) ^(-mu);
% Projection values
xProj=x(:, ones(nPoints, 1));

% Calculate interpolation surface value when iterate is not in the threshold of any constraint
if (nargout>1)
    Dm1=-mu.*(D) ^(-mu-1);
    % Calculate first term of gradient
    Df1a=(A_Unk)' *((Dm1/Wden) * fi');
    Dsum=((A_Unk)' * (Dm1));
    Df1b=(Dsum(:, ones(nPoints, 1))/Wden)*((Wnum/Wden) * fi');
    Df1=(Df1a-Df1b);
    % Calculate second term of gradient
    Px_Active=P(-idxKnown, idxKnown) * x(idxKnown);
    Df0=P(-idxKnown,-idxKnown) * x Proj(-idxKnown,:)+...
    Px_Active (:, ones(nPoints, 1)) + q(-idxKnown, ones(nPoints, 1));
    Df0(logical(A_Unk'))=0;
    Df2=Df0*((Wnum/Wden));
    gsurf=Df1+Df2;
    % Calculate Hessian matrix
    if (nargout>3)
\[ Dm_2 = (\mu^2 + \mu) \cdot (D) \cdot (\mu + 2) \]

% Calculate first term of Hessian matrix
spDm1F = spdiags ((Dm1 * fi'), 0, nPoints, nPoints);
spDm2F = spdiags ((Dm2 * fi') * Wden, 0, nPoints, nPoints);
spDm2 = spdiags (Dm2, 0, nPoints, nPoints);
spWnumF = spdiags ((Wnum * fi') / Wden', 0, nPoints, nPoints);
H1a = Dsum(:, ones(nPoints, 1)) * (spDm1F * AUnk);
H1 = H1 - H1a' + (AUnk' * spDm2F * AUnk -
   - (fi * (Wnum)) * AUnk' * spDm2 * AUnk -
   + Dsum(:, ones(nPoints, 1)) * 2 * spDm1F * AUnk -
   + Dsum(:, ones(nPoints, 1)) *
   + 2 * spWnumF * Dsum(:, ones(nPoints, 1))') ;

% Calculate second term of Hessian matrix
H2 = Df0 * (spdiags (Dm1, 0, nPoints, nPoints) * AUnk -
   - spdiags (Wnum / Wden, 0, nPoints, nPoints) -
   + Dsum(:, ones(nPoints, 1))');

% Calculate first term of Hessian matrix
H3 = H2';

% Calculate fourth term of Hessian matrix
H4a = AUnk' * spdiags (Wnum / Wden, 0, nPoints, nPoints) * AUnk;
H4b = sum (Wnum / Wden) * speye (nDimUnk) - H4a;
H4 = P(¬idxKnown, ¬idxKnown) * H4b -
   - H4a * P(¬idxKnown, ¬idxKnown) - spdiags (spdiags (...
   P(¬idxKnown, ¬idxKnown), 0, nDimUnk, nDimUnk));

hsurf = H1 + (H2 + H3) / Wden + H4;

A.2.1 Steepest Descent Function
function [x,fval,exitflag,nIter,x_history,∆History]=...
INTERP_STEEPEST_DESCENT_SS_QP(f,x0,options)

% INTERP_STEEPEST_DESCENT_SS_QP: Convex optimization algorithm. 
% Steepest Descent method for the generalization of Shepard’s
% formula for quadratic programs.
% 
% min f(x)
% x
% 
% where f(x) is the interpolation surface function.
% 
% [x,fval,exitflag,nIter,x_history,∆History]=...
% INTERP_STEEPEST_DESCENT_SS_QP(f,x0,options)
% 
% Sets the following global variables:
% 
% - nDim: Number of dimensions of the original problem.
% - nDim_Unk: Number unknown variables at the current iteration.
% - boundaryTol: Threshold parameter for the constraint planes
% - bBoundary: Flag to mark presence of active constraints at
% last iterate. (0 -> no active constraints).
% - idxKnown: Index of known variables, corresponding to the
% active constraints.
% - A_Unk: Matrix consisting of the rows and columns of A corresponding
% to unknown variables at the current iteration.
% - b_Unk: Vector consisting of elements of b corresponding to unknown
% variables at the current iteration.
% 
% Calls the following global variables:
% 
% - A: Matrix of normals for the bound constraints. (A*x≥b)
% - b: Vector of intercepts for the bound constraints. (A*x≥b)
% - P: Hessian matrix of quadratic function
% - q: First order coefficients vector of quadratic function
% 
% Input:
% 
% - f: Function handle of the interpolation surface
% - x0: Initial point
% - options: Structure for defining parameters
%   - options.alpha0 (Default:1)
%   - options.xTol (Default:1e–12)
%   - options.fTol (Default:eps)
%   - options.gTol (Default:eps)
%   - options.boundaryTol (Default:8e–10)
%   - options.maxIter (Default:100*nDim)
% 
% Output:
% 
% - x: Solution point
% - fval: Minimum value
% - exitflag: Cause of exit
%   - 1 Step length too small
%   - 2 Maximum number of iterations reached
%   - 1 Gradient norm is too small
%   - 2 Decrease in function value is too small
% - nIter: Number of iterations
% - x_history: Matrix of recorded iterates
% - ∆History: Vector of recorded step lengths
% 
% Global variables
% global A, global b, global P, global q, global nDim,
% global A_Unk, global b_Unk, global nDim_Unk
% global idxKnown, global boundaryTol
% nDim=length(x0);
% nDim_Unk=nDim;
% A_Unk=A;
% b_Unk=b;

if (nargin<3)
  alpha0=1;
end
xTol=1e−12;
fTol=eps;
gTol=eps;
boundaryTol=8e−10;
maxIter=100∗nDim;
else if ( nargin==3)
alpha0=options.alpha0;
xTol=options.xTol;
fTol=options.fTol;
gTol=options.gTol;
boundaryTol=options.boundaryTol;
maxIter=options.maxIter;
end
%Initialize variable to record iterates
x_history=zeros(nDim,maxIter);
%Initialize variable to record step sizes
DHistory=zeros(maxIter,1);
%Vector of known variables at x0;
idxKnown=false(nDim,1);
xk=x0;
[fk,gk]=f(xk);
%Dummy value in order to enter first while loop
fkOld=Inf;
%Loop counter
k=1;
while (k≤maxIter&&norm(gk)>gTol&&(fkOld−fk)>fTol)
%Update the index for known variables.
idxKnownOld=idxKnown;
bTest1=(A(1:nDim,:)*xk−b(1:nDim))≤boundaryTol;
bTest2=(A(nDim+1:end,:)*xk−b(nDim+1:end))≤boundaryTol;
idxKnown=bTest1|bTest2;
%Index of new active constraints
newIdx=logical(idxKnown−idxKnownOld);
%If there are new Active bounds found
if (any(newIdx))
%Record the exact value of the elements of x that correspond to the
%new active constraints
xk(newIdx)=b([newIdx; false(nDim,1)]).∗bTest1(newIdx)−...
b([false(nDim,1);newIdx]).∗bTest2(newIdx);
A_Unk=A([−idxKnown;−idxKnown],−idxKnown);
b_Unk=b([−idxKnown;−idxKnown]);
%Update the number of unknown variables
nDim_Unk=nDim_Unk−sum(newIdx);
%Solve unconstrained problem for the unknown variables
xUnc=−(P(−idxKnown,−idxKnown)
\...)
(P(−idxKnown,idxKnown)∗x_k(idxKnown)+q(−idxKnown));
%If feasible, record result and terminate
if (all(A_Unk∗xUnc−b_Unk≥0))
x_k(−idxKnown)=xUnc;
break;
else
[fk,gk]=f(x_k);
end
end
%Steepest descent direction:
p_k=−g_k;
%Normalize if possible
norm_p_k=norm(p_k);
if (norm_p_k>eps)
p_k=p_k/norm_p_k;
end
[a_k]=backtrackLineSearch_SS(f,x_k,p_k,alpha0);
s_k=a_k∗p_k;
Dk=norm(s_k);
%Break if step lengths is too small or if direction yields 0−vector.
if (Dk<xTol||norm(p_k)==0)
break;
end
% Update iterate
x_k(-idxKnown)=x_k(-idxKnown)+s_k;
% Update function and gradient values
f_kOld=f_k;
[f_k g_k]=f(x_k);
% Record iterate and step length
x_history(:,k+1)=x_k;
Delta_History(k)=Delta_k;
k=k+1;
end
% Record stopping criteria used
if (k>maxIter)
exitflag=0;
elseif(norm(g_k)<=g_Tol)
exitflag=-1;
elseif((f_kOld-f_k)<=f_Tol)
exitflag=-2;
else
exitflag=1;
end
x=x_k;
f_val=F0(x);
nIter=k;

A.2.2 BFGS Method Function

function [x,fval,exitflag,nIter,x_history,D_History]=...
Interp_BFGS_SS_QP(f,x0,options)

% INTERP_BFGS_SS_QP Convex optimization algorithm. BFGS
% method for the generalization of Shepard’s formula for quadratic programs
% subject to bounds on the variables.
%
% min f(x)
% x
% where f(x) is the interpolation surface function.
%
% [x,fval,exitflag,nIter,x_history,D_History]=...
% Interp_BFGS_SS_QP(f,x0,options)

% Sets the following global variables:
% - nDim: Number of dimensions of the original problem.
% - nDim_Unk: Number of unknown variables at the current iteration.
% - boundaryTol: Threshold parameter for the constraint planes
% - bBoundary: Flag to mark presence of active constraints at last iterate.
% - idxKnown: Index of known variables, corresponding to the active constraints
% - A_Unk: Matrix consisting of the rows and columns of A corresponding to ...
% unknown variables at the current iteration.
% - b_Unk: Vector consisting of elements of b corresponding to unknown
% variables at the current iteration.
%
% Calls the following global variables:
% - A: Matrix of normals for the bound constraints. (A*x>=b)
% - b: Vector of intercepts for the bound constraints. (A*x>=b)
% - P: Hessian matrix of quadratic function
% - q: First order coefficients vector of quadratic function
%Input:
% - f: Function handle of the interpolation surface
% - x0: Initial point
% - options: Structure for defining parameters
%  % options: alpha0 (Default: 1)
%  % options.xTol (Default: 1e-12)
%  % options.fTol (Default: eps)
%  % options.gTol (Default: eps)
%  % options.initialSD (Default: 4)
%  % options.nSD (Default: 5)
%  % options.sdBackoff (Default: .5)
%  % options.minCurvature (Default: 1e-8)
%  % options.H_0 (Default: eye(nDim))
%  % options.boundaryTol (Default: 8e-10)
%  % options.maxIter (Default: 100*nDim)

%Output:
% - x: Solution point
% - fval: Minimum value
% - exitflag: Cause of exit
%  % 1 Step length too small
%  % 0 Maximum number of iterations reached
%  % -1 Gradient norm is too small
%  % -2 Decrease in function value is too small
%  % - nIter: Number of iterations
%  % - x_history: Matrix of recorded iterates
%  % - ΔHistory: Vector of recorded step lengths

%Global variables
global A, global b, global P, global q, global nDim
global A-Unk, global b-Unk, global nDim-Unk
global idxKnown, global boundaryTol
nDim=length(x0);
nDim-Unk=nDim;
A-Unk=A;
b-Unk=b;

%Set parameters
if (nargin<3)
    alpha0=1;
    xTol=1e-12;
    fTol=eps;
    gTol=eps;
    initialSD=5;
    nSD=5;
    sdBackoff=.5;
    minCurvature=1e-8;
    H_0=eye(nDim);
    boundaryTol=8e-10;
    maxIter=100*nDim;
elseif (nargin==3)
    alpha0=options.alpha0;
    xTol=options.xTol;
    fTol=options.fTol;
    gTol=options.gTol;
    initialSD=options.initialSD;
    nSD=options.nSD;
    sdBackoff=options.sdBackoff;
    minCurvature=options.minCurvature;
    boundaryTol=options.boundaryTol;
    maxIter=options.maxIter;
end

%Initialize variable to record iterates
x_history=zeros(nDim,maxIter);
x_history(:,1)=x0;

%Initialize variable to record step sizes
ΔHistory=zeros(maxIter,1);
Vector of known variables at x0;
idxKnown=false(nDim,1);
x_k=x0;
[f_k g_k]=f(x_k);
%Dummy value in order to enter first while loop
f_kOld=Inf;
%Flag to mark the resetting of H_k
bResetH_k=0;
%Flag to mark that the algorithm has found another active constraint.
b_NewActiveConstraint=0;
%Loop counter
k=1;
while (k≤maxIter&&norm(g_k)>gTol&&(mod(k−1,nSD)̸=0||(f_kOld−f_k)>fTol))
%Update the index for known variables.
idxKnown=idxKnown;
bTest1=(A(1:nDim,:)*x_k−b(1:nDim))≤boundaryTol;
bTest2=(A(nDim+1:end,:)*x_k−b(nDim+1:end))≤boundaryTol;
idxKnown=bTest1|bTest2;
%Index of new active constraints
newIdx=logical(idxKnown−idxKnownOld);
%If there are new Active bounds found
if (any(newIdx))
%Record the exact value of the elements of x that correspond to the new active constraints
x_k(newIdx)=b(newIdx;false(nDim,1))∗bTest1(newIdx)−... bTest2(newIdx);
A_Unk=A([−idxKnown;−idxKnown]∗idxKnown);
b_Unk=b([−idxKnown;−idxKnown]);
%Update the number of unknown variables
nDim_Unk=nDim_Unk+sum(newIdx);
%Solve unconstrained problem for the unknown variables
xUnc=−P(−idxKnown,−idxKnown)
(P(−idxKnown,idxKnown)∗x_k(idxKnown)+q(−idxKnown));
%If feasible, record result and terminate
if (all(A_Unk∗xUnc−b_Unk≥0))
x_k(−idxKnown)=xUnc;
break;
else
[f_k g_k]=f(x_k);
b_NewActiveConstraint=1;
end
% Update H_k
if (k==1||b_NewActiveConstraint==1)
H_k=H_0(−idxKnown,−idxKnown);
else
y_k=g_k−g_kOld;
st_y=s_k∗y_k;
s_yt=s_k∗y_k';
if (st_y>minCurvature)
H_k=(eye(nDim_Unk)−s_yt/st_y)∗H_k...
∗(eye(nDim_Unk)−s_yt'/st_y)+(s_k∗s_k')/st_y;
end
end
%Calculate quasi−Newton step if current iteration is not a steepest descent iteration.
if (k≥initialSD&&mod(k,nSD)=0)
p_k=−H_k*g_k;
norm_p_k=norm(p_k);
%If p_k is a descent step, H_k needs to be resetted
if (p_k'*g_k>0)
bResetH_k=1;
H_k=H_0(−idxKnown,−idxKnown);
end
%Calculate steepest descent direction if:
% Current iteration is a steepest descent iteration.
% Newton direction yielded an ascent direction (p_k'*g_k>0).
% Newton direction yielded an invalid direction.
if (k<initialSD || mod(k,nSD)==0|| bResetH_k || any(isnan(p_k)) || any(isinf(p_k)))
    p_k=g_k;
    %Normalize p_k
    norm_p_k=norm(p_k);
    if (norm_p_k>eps)
        p_k=p_k/norm_p_k;
    end
    bResetH_k=0;
end
%Adjust alpha0 when the norm of p_k is too small in order to avoid early termination.
if (norm_p_k<100*boundaryTol)
    alpha_0=boundaryTol/norm_p_k;
else
    alpha_0=alpha0;
end
[a_k]=backtrackLineSearch_SS(f,x_k,p_k,alpha_0);
%Apply a backoff factor in the initial steepest descent iterations %to avoid getting close to the boundary too soon.
if (k<initialSD)
    s_k=sdBackoff*a_k*p_k;
else
    s_k=a_k*p_k;
end
%Break if step lengths is too small or if direction yields 0-vector.
∆_k=norm(s_k);
if (∆_k<xTol)
    break;
end
%Update iterates
x_k(~idxKnown)=x_k(~idxKnown)+s_k;
%Update function value and gradient
g_kOld=g_k;
f_kOld=f_k;
[f_k g_k]=f(x_k);
%Record iterate and step length
x_history(:,k+1)=x_k;
∆_History(k)=∆_k;
%Clear flag
b_NewActiveConstraint=0;
k=k+1;
end
%Record stopping criteria used
if (k>maxIter)
exitflag=0;
elseif(norm(g_k)≤gTol)
exitflag=-1;
elseif((f_kOld-f_k)≤fTol)
exitflag=-2;
else
    exitflag=1;
end
x=x_k;
fval=F0(x);
nIter=k;
x_history(:,k+1:end)=[];
∆_History(k+1:end)=[];
function [x,fval,exitflag,nIter,x_history,Δ,History]=...
  Interp_DampedBFGS_SS_QP(f,x0,options)
%
% Interp_DampedBFGS_SS_QP Convex optimization algorithm. BFGS
% method for the generalization of Shepard’s formula for quadratic programs
% subject to bounds on the variables.
% where f(x) is the interpolation surface function.
%
% [x,fval,exitflag,nIter,x_history,Δ,History]=...
  Interp_DampedBFGS_SS_QP(f,x0,options)
%
% Sets the following gobal variables:
% nDim: Number of dimensions of the original problem.
% nDim_Unk: Number unknown variables at the current iteration.
% boundaryTol: Threshold parameter for the constraint planes
% (0 -> no active constraints).
% idxKnown: Index of known variables, corresponding to the active constraints
% unknown variables at the current iteration.
% b_Unk: Vector consisting of elements of b corresponding to unknown
% variables at the current iteration.
% A: Matrix of normals for the bound constraints. (A∗x≥b)
% b: Vector of intercepts for the bound constraints. (A∗x≥b)
% P: Hessian matrix of quadratic function
% q: First order coefficients vector of quadratic function
%
%!Input:
%! f: Function handle of the interpolation surface
%! x0: Initial point
%! options: Structure for defining parameters
%! options:alpha0 (Default:1)
%! options.xTol (Default:1e−12)
%! options.fTol (Default:eps)
%! options.gTol (Default:eps)
%! options.initialSD (Default: 4)
%! options.nSD (Default:5)
%! options.sdBackoff (Default:.5)
%! options.minCurvature (Default: 1e−8)
%! options.B 0 Default (eye(nDim))
%! options.boundaryTol (Default:8e−10)
%! options.maxIter (Default:100*nDim)
%
%!Output:
%! x: Solution point
%! fval: Minimum value
%! exitflag: Cause of exit
! 1 Step length too small
! 0 Maximum number of iterations reached
! −1 Gradient norm is too small
! −2 Decrease in function value is too small
%! nIter: Number of iterations
%! x_history: Matrix of recorded iterates
%! Δ,History: Vector of recorded step lengths
%
%Global variables

global A, global b, global P, global q, global nDim,
global A_Unk, global b_Unk, global nDim_Unk
global idxKnown, global boundaryTol

nDim=length(x0);
nDim_Unk=nDim;
A_Unk=A;
b_Unk=b;

%Set parameters
if ( nargin<3 )
    alpha0=1;
xTol=le-12;
fTol=eps;
gTol=eps;
initialSD=5;
nSD=5;
sDBackoff=.5;
end
elseif ( nargin==3 )
    alpha0=options.alpha0;
xTol=options.xTol;
fTol=options.fTol;
gTol=options.gTol;
initialSD=options.initialSD;
nSD=options.nSD;
sDBackoff=options.sDBackoff;
boundaryTol=options.boundaryTol;
maxIter=options.maxIter;
end

%Initialize variable to record iterates
x_history=zeros(nDim,maxIter);
x_history(:,1)=x0;

%Initialize variable to record step sizes
Delta_History=zeros(maxIter,1);

%Vector of known variables at x0
idxKnown=false(nDim,1);
x_k=x0;
[f_k,g_k]=f(x_k);

%Dummy value in order to enter first while loop
f_kOld=Inf;

%Flag to mark the resetting of H_k
bResetB_k=0;
%Flag to mark that the algorithm has found another active constraint.
b_NewActiveConstraint=0;
%Loop counter
k=1;
while ( k<=maxIter&&(norm(g_k)>gTol||all(~idxKnown)&(mod(k-1,nSD)<>0||(f_kOld-f_k)>=fTol)) )
    %Update the index for known variables.
    idxKnown_Old=idxKnown;
    bTest1=(A(1:nDim,:)∗x_k−b(1:nDim))<=boundaryTol;
    bTest2=(A(nDim+1:end,:)∗x_k−b(nDim+1:end))<=boundaryTol;
    idxKnown=bTest1|bTest2;
    %Index of new active constraints
    newIdx=logical(idxKnown−idxKnown_Old);
    %If there are new Active bounds found
    if (any(newIdx))
        %Record the exact value of the elements of x that correspond to the new active constraints
        x_k(newIdx)=b([newIdx;false(nDim,1)])∗bTest1(newIdx)
        b([false(nDim,1);newIdx])∗bTest2(newIdx);
        A_Unk=A([~idxKnown;~idxKnown],[~idxKnown]);
        b_Unk=b([~idxKnown;~idxKnown]);
        %Update the number of unknown variables
        nDim_Unk=nDim_Unk−sum(newIdx);
        %Solve unconstrained problem for the unknown variables
        xUnc=−P(~idxKnown,−idxKnown)∗x_k(idxKnown)+q(~idxKnown);
        %If feasible, record result and terminate
        if (all(A_Unk∗xUnc−b_Unk>=0))
            x_k(~idxKnown)=xUnc;
            break;
    end
    %Update the number of known variables
    idxKnown_Old=idxKnown;
    %Update the number of unknown variables
    nDim_Unk=nDim_Unk−sum(newIdx);
end
135     else
136         [f.k g.k] = f(x.k);
137         b(NewActiveConstraint) = 1;
138     end
139 % Update H_k
140     if (k==1 || b(NewActiveConstraint)==1)
141         bResetB.k = 1;
142         B.k = B.0 (~idxKnown, ~idxKnown);
143     else
144         y.k = g.k - g.kOld;
145         st.y = s.k' * y.k;
146         SBS.k = (s.k' * B.k * s.k);
147         rhs = (0.2) * sBS.k;
148 % Theta value
149         if (st.y >= rhs)
150             theta.k = 1;
151         else
152             theta.k = (0.8) * sBS.k / (sBS.k - st.y);
153         end
154         r.k = theta.k * y.k + (1 - theta.k) * B.k * s.k;
155         B.k = B.0 (~idxKnown, ~idxKnown);
156     end
157 %Calculate quasi–Newton step if current iteration is not a steepest descent
158 %iteration.
159     if (k>initialSD && mod(k, nSD) ̸= 0)
160         p.k = -B.k * g.k;
161         norm.p.k = norm(p.k);
162         if (p.k' * g.k > 0)
163             bResetB.k = 1;
164             B.k = B.0 (~idxKnown, ~idxKnown);
165         end
166 %Calculate steepest descent direction if:
167 % - Current iteration is a steepest descent iteration.
168 % - Newton direction yielded an ascent direction (p.k' * g.k > 0).
169 % - Newton direction yielded an invalid direction.
170     if (k<initialSD || mod(k, nSD) == 0 || bResetB.k || any(isnan(p.k)) || any(isinf(p.k)))
171         p.k = g.k;
172 %Normalize p.k
173         norm.p.k = norm(p.k);
174         if (norm.p.k > eps)
175             p.k = p.k / norm.p.k;
176         end
177         bResetB.k = 0;
178 end
179 %Adjust alpha0 when the norm of p.k is too small in order to avoid
180 %early termination.
181     if (norm.p.k < 100 * boundaryTol)
182         alpha0 = boundaryTol / norm.p.k;
183     else
184         alpha0 = alpha0;
185     end
186 [a.k] = backtrackLineSearch_SS(f, x.k, p.k, alpha0);
187 %Apply a backoff factor in the initial steepest descent iterations
188 %to avoid getting close to the boundary too soon.
189 if (k<initialSD)
190         s.k = sdBackoff * a.k * p.k;
191 else
192         s.k = a.k * p.k;
193 end
194 ∆k = norm(s.k);
195 if (∆k <= xTol)
196     break;
197 end
198 %Update iterates
199 x.k (~idxKnown) = x.k (~idxKnown) + s.k;
A.2.4 Newton’s Method Function

```matlab
function ... [x, fval, exitflag, nIter, x_history, ∆History] = Interp_Newton_SS_QP(f, x0, options)
% INTERP_NEWTON_SS_QP Convex optimization algorithm. Newton’s method for the generalization of Shepard’s formula for quadratic programs
% subject to bounds on the variables.
% min f(x)
% x
% where f(x) is the interpolation surface function.
% %
% [x, fval, exitflag, nIter, x_history, ∆History] = ...
% Interp_Newton_SS_QP(f, x0, options)
% % Sets the following global variables:
% - nDim: Number of dimensions of the original problem.
% - nDimUnk: Number unknown variables at the current iteration.
% - boundaryTol: Threshold parameter for the constraint planes
% - bBoundary: Flag unknown variables at the current iteration.
% - idxKnown: Index of known variables, corresponding to known variables at the current iteration.
% - AUnk: Matrix consisting of the rows and columns of A corresponding to unknown variables at the current iteration.
% - bUnk: Vector consisting of elements of b corresponding to unknown variables at the current iteration.
% % Calls the following global variables:
% - A: Matrix of normals for the bound constraints. (A*x ≥ b)
% - b: Vector of intercepts for the bound constraints. (A*x ≥ b)
% - P: Hessian matrix of quadratic function
% - q: First order coefficients vector of quadratic function
```
% Input:
% - f: Function handle of the interpolation surface
% - x0: Initial point
% - options: Structure for defining parameters
%  options.alpha0 (Default: 1)
%  options.xTol (Default: 1e-12)
%  options.fTol (Default: eps)
%  options.gTol (Default: eps)
%  options.initialSD (Default: 4)
%  options.nSD (Default: 5)
%  options.sdBackoff (Default: .5)
%  options.boundaryTol (Default: 8e-10)
%  options.maxIter (Default: 100*nDim)

% Output:
% - x: Solution point
% - fval: Minimum value
% - exitflag: Cause of exit
%  1 Step length too small
%  0 Maximum number of iterations reached
%  -1 Gradient norm is too small
%  -2 Decrease in function value is too small
% - nIter: Number of iterations
% - xHistory: Matrix of recorded iterates
% - ∆History: Vector of recorded step lengths

% Global variables
global A, global b, global P, global q, global nDim,
global A_Unk, global b_Unk, global nDim_Unk
global idxKnown, global boundaryTol

nDim = length(x0);
nDim_Unk = nDim;
A_Unk = A;
b_Unk = b;

% Set parameters
if (nargin < 3)
    alpha0 = 1;
    xTol = 1e-12;
    fTol = eps;
    gTol = eps;
    initialSD = 5;
    nSD = 5;
    sdBackoff = .5;
    boundaryTol = 8e-10;
    maxIter = 100*nDim;
elseif (nargin == 3)
    alpha0 = options.alpha0;
    xTol = options.xTol;
    fTol = options.fTol;
    gTol = options.gTol;
    initialSD = options.initialSD;
    nSD = options.nSD;
    sdBackoff = options.sdBackoff;
    boundaryTol = options.boundaryTol;
    maxIter = options.maxIter;
end

% Initialize variable to record iterates
x_history = zeros(nDim, maxIter);

% Initialize variable to record step sizes
∆History = zeros(maxIter, 1);

% Vector of known variables at x0
idxKnown = false(nDim, 1);

x_k = x0;
[f_k g_k foo h_k] = f(x_k);

% Dummy value in order to enter first while loop
119

f_kOld=Inf;

%Loop counter
k=1;

while(k<=maxIter&knorm(g_k)>gTol&(f_kOld-f_k)>fTol)

%Update the index for known variables.
dxKnown_Old=dxKnown;
bTest1=(A(1:ndim,:)∗x_k−b(1:ndim))≤boundaryTol;
bTest2=(A(ndim+1:end,:)∗x_k−b(ndim+1:end))≤boundaryTol;
dxKnown=bTest1∨bTest2;

%Index of new active constraints
newIdx=logical(dxKnown−dxKnown_Old);

%If there are new Active bounds found
if(any(newIdx))

%Record the exact value of the elements of x that correspond to the
%new active constraints
x_k(newIdx)=b(1:ndim,1)∗bTest1(newIdx)−... −b(1:ndim,1)∗bTest1(newIdx);
A_Unk=A([−idxKnown,−idxKnown,−idxKnown]);
b_Unk=b([−idxKnown,−idxKnown]);

%Update the number of unknown variables
nDim_Unk=nDim_Unk+sum(newIdx);

%Solve unconstrained problem for the unknown variables
xUnc=−P(¬idxKnown,¬idxKnown)∗(P(¬idxKnown,idxKnown)∗x_k(idxKnown)+q(¬idxKnown));

%If feasible, record result and terminate
if(all(A_Unk∗xUnc−b_Unk≥0))
x_k(¬idxKnown)=xUnc;
break;
else
[f_k g_k h_k]=f(x_k);
end
end

%Calculate Newton step if current iteration is not a steepest descent
%Iteration.
if(k>initialSD&&mod(k,nSD)≠0)
p_k=−h_k\g_k;
end

%Calculate steepest descent direction if:
%− Current iteration is a steepest descent iteration.
%− Newton direction yielded an ascent direction (p_k^Tg_k>0).
%− Newton direction yielded an unvalid direction.
if(k>initialSD||mod(k,nSD)==0||(p_k^Tg_k>0)||any(isnan(p_k))||any(isinf(p_k)))
p_k=−g_k;

%Normalize if possible
norm_p_k=norm(p_k);
if(norm_p_k>eps)
p_k=p_k/norm_p_k;
end
end

%Adjust alpha0 when the norm of p_k is too small in order to avoid
%early termination.
if(norm_p_k<100∗boundaryTol)
alpha_0=boundaryTol/norm_p_k;
else
alpha_0=alpha0;
end
[a_k]=backtrackLineSearch_SS(f,x_k,p_k,alpha_0);

%Apply a backoff factor in the initial steepest descent iterations
%to avoid getting close to the boundary too soon.
if(k>initialSD)
s_k=s_backoff*a_k∗p_k;
else
s_k=a_k∗p_k;
end

%Break if step lengths is too small or if direction yields 0−vector.
Δ_k=norm(s_k);
if(Δ_k<epsilon)break;
%Update iterates
x_{k}(-\text{idxKnown})=x_{k}(-\text{idxKnown})+s_{k};

%Update function value, gradient and hessian if next iteration is a Newton
%iteration, otherwise only update function value and gradient.
if (k>\text{initialSD}-\text{ld}kmod(k+1,nSD)\neq0)
f_{k}\text{Old}=f_{k};
[f_{k}\ g_{k}\ h_{k}]=f(x_{k});
else
f_{k}\text{Old}=f_{k};
[f_{k}\ g_{k}]=f(x_{k});
end

%Record iterate and step length
x_{\text{history}}(:,k+1)=x_{k};
\Delta_{\text{History}}(k)=\Delta_{k};
k=k+1;
end

%Record stopping criteria used
if (k>\text{maxIter})
exitflag=0;
elseif (\text{norm}(g_{k})\leq\text{gTol})
exitflag=-1;
elseif ((f_{k}\text{Old}-f_{k})\leq\text{fTol})
exitflag=-2;
else
exitflag=1;
end
x=x_{k};
fval=F0(x);
nIter=k;
x_{\text{history}}(:,k+1)=x_{k};
x_{\text{history}}(:,k+2:end)=[];
\Delta_{\text{History}}(k+1:end)=[];
end
Bibliography/References


