Interior-Point Algorithms for Quadratic Programming

Thomas Reslow Krüth

Kongens Lyngby 2008
IMM-M.Sc-2008-19

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-M.Sc: ISSN 0909-3192

Preface

This Master thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in the period of September 1st 2007 to February 29th 2008 under the supervision of John B. Jørgensen.

Kongens Lyngby, February 2008

Thomas Reslow Krüth

Contents

Chapter 1

Introduction

Quadratic Programs appear in several different applications. For example in finance (portfolio optimization), model predictive control, and also as subproblems in sequential Quadratic Programming and augmented Lagrangian methods. This thesis deals with solving quadratic programs using interior-point methods. The main focus is to understand the techniques involved in designing interior-point methods and to explore different ways of improving their computational performance. The exploration of interior-point methods can not be based solely on pure theoretical discussions, so to gain some practical experience interior-point methods are implemented and tested. The mathematical programming language MATLAB is used to implement the methods and conduct various experiments to test and compare the methods.

1.1 Quadratic Programming

A Quadratic Program (QP) is a special type of optimization problem consisting of a quadratic objective function $f(x)$ which we want to minimize with respect to a vector $x \in \mathbb{R}^n$ subject to some affine equality and inequality constraints. A general QP can be formulated as shown in (1.1)

$$\min_{x\in\mathbb{R}^n} \quad f(x) = \frac{1}{2}x^T Gx + g^T x \tag{1.1a}$$
$$\text{s.t.} \quad A^T x \geq b \tag{1.1b}$$
$$C^T x = d \tag{1.1c}$$

where $g \in \mathbb{R}^n$, $b \in \mathbb{R}^{m_A}$ and $d \in \mathbb{R}^{m_C}$ are vectors and $G \in \mathbb{R}^{n\times n}$, $A \in \mathbb{R}^{n\times m_A}$ and $C \in \mathbb{R}^{n\times m_C}$ are matrices.
In this thesis please note that lower case letters denote vectors or scalars and upper case letters denote matrices.
The inequality and equality constraints define a feasible region in which the solution to the problem must be located in order for the constraints to be satisfied. The matrix G is commonly referred to as the Hessian. More formally the feasible set Ω is defined as the set of points x that satisfy the constraints as indicated in (1.2).

$$\Omega = \{x \mid A^T x = b;\ C^T x \geq d\} \tag{1.2}$$

To visualize the concept of feasible and infeasible regions we use the small example (1.3) where only inequality constraints are present, Nocedal and Wright [1], p. 492.

$$\min_{x\in\mathbb{R}^2} \quad f(x) = 2x_1 + 3x_2 + 4x_1^2 + 2x_1x_2 + x_2^2 \tag{1.3a}$$
$$\text{s.t.} \quad x_1 - x_2 \geq 0 \tag{1.3b}$$
$$-x_1 - x_2 \geq -4 \tag{1.3c}$$
$$-x_1 \geq -3 \tag{1.3d}$$

Rewriting this in the notation introduced in (1.1) yields

$$\min_{x\in\mathbb{R}^2} \quad \frac{1}{2}x^T Gx + g^T x = \frac{1}{2}\begin{bmatrix} x_1 & x_2 \end{bmatrix}\begin{bmatrix} 8 & 2 \\ 2 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 2 & 3 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$
$$\text{s.t.} \quad A^T x \geq b \Rightarrow \begin{bmatrix} 1 & -1 \\ -1 & -1 \\ -1 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ -4 \\ -3 \end{bmatrix}$$

In figure 1.1 the problem is shown with the purpose of visualizing the feasible region and the infeasible region. The grey areas represent the infeasible region and the white area represents the feasible region. The solution to the problem, $x = [0.1667, -1.6667]^T$, is marked with a cross.

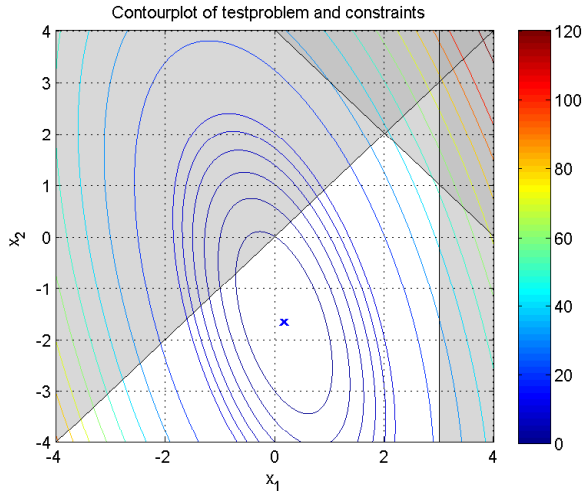

Figure 1.1: *Example of feasible and infeasible region.*

In this case the global minimum of the QP coincides with the global minimum of the function $f(x)$. In many cases the global minimum of $f(x)$ will be located in the infeasible region and the solution to the constrained problem will be situated on one of the constraints.

In this thesis we will restrict our attention to convex QP's. These are problems in which the objective function $f(x)$ is convex, i.e. the Hessian G is positive semidefinite.

1.2 Interior-Point Methods

QP's can be solved using several different methods. In this presentation we use interior-point methods that belong to the class of linear and nonlinear programming methods. For solving linear and quadratic programs interior-point methods have been proven to work well in practice and the theory for the methods is well developed. Therefore interior-point methods are widely used in applications.

Interior-point methods solve problems iteratively such that all iterates satisfy the inequality constraints strictly. They approach the solution from either the interior or exterior of the feasible region but never lie on the boundary of this region. Each iteration of the method is computationally expensive but can make significant progress towards the solution, [1] p. 393. These particular characteristics are what seperates interior-point methods from other methods.

To set up the equations enabling us to design the interior-point methods we use the general theory on constrained optimization by defining a Lagrangian function and setting up Karush-Kuhn-Tucker (KKT) conditions for the QP's we wish to solve. The KKT-conditions, or optimality conditions, are conditions that must be satisfied for a vector x to be a solution of a given QP.
The interior-point methods presented in this thesis solve the dual problem simultaneously with the primal problem and for this reason these interior-point methods are also referred to as primal-dual interior-point methods.

1.3 Structure of thesis

The structure of the thesis more or less follows the chronological development of the project. The thesis is divided into five chapters. The contents of each chapter is summarized below.

Interior-Point Method for an inequality constrained QP In this chapter we introduce the techniques involved in setting up the predictor-corrector (PC) method, which is one of the most widely used interior-point methods. For simplicity we develop the method for an inequality constrained QP. The method is implemented and tested.

General QP The developed PC method is extended to handle QP's with both inequality and equality constraints. The unavailability of test problems for this type of QP motivates the development of a QPS reader for MATLAB.

QPS reader A QPS reader allows access to a set of 138 QP test problems. In this chapter the implementation of a QPS reader for MATLAB is explained and tested.

Multiple Centrality Corrections The Multiple Centrality Corrections (MCC) method is introduced as a modification of the PC method. Implementations of the MCC method are tested and the computional performance of the PC method and the MCC method is compared.

Further modifications Two methods to further modify the PC method and MCC method are presented in this chapter. The first is a new step length strategy that has the potential to improve both methods. The second is a modification aimed at improving the MCC method. Both modifications are tested.

Chapter 2

Interior-Point Method for an inequality constrained QP

In this chapter we develop a primal-dual interior-point method for solving the inequality constrained convex quadratic program shown in (2.1) where

$$\min_{x \in \mathbb{R}^n} \quad f(x) = \frac{1}{2} x^T G x + g^T x \tag{2.1a}$$

$$\text{s.t.} \quad A^T x \geq b \tag{2.1b}$$

x is a $n \times 1$ vector where n is the number of variables, G is a $n \times n$ matrix, g is a $n \times 1$ vector, A is a $n \times m$ matrix, where m is the number of inequality constraints, and b is a $m \times 1$ vector.
The purpose of developing an algorithm for solving this simplified version of the general quadratic program is to gain a better understanding of the techniques and steps involved in designing an interior-point method. As we will see in the next chapter, the extensions needed to solve the general quadratic program, are fairly trivial to include.
The sections in this chapter will cover the steps involved in designing the algorithm. Additionally we will implement the algorithm in MATLAB and test it to see how well it performs.

2.1 Optimality conditions

In this section we state the optimality (KKT) conditions for the inequality constrained QP and show how to use Newton's method to progress iteratively towards the solution.
First we state the Lagrangian function for the problem (2.1) in (2.2) where λ_i is the Lagrange multiplier for the i'th inequality constraint, a_i^T is the i'th row in A^T and b_i is the i'th element in b.

$$L(x, \lambda) = \frac{1}{2} x^T G x + g^T x - \sum_{i=1}^{m} \lambda_i (a_i^T x - b_i) \tag{2.2}$$

The optimality conditions are stated in (2.3), where $i = 1, ..., m$.

$$\nabla_x L(x, \lambda) = 0 \Rightarrow Gx + g - A\lambda = 0 \tag{2.3a}$$
$$A^T x - b \geq 0 \tag{2.3b}$$
$$\lambda_i (A^T x - b)_i = 0 \tag{2.3c}$$
$$\lambda \geq 0 \tag{2.3d}$$

The conditions are rewritten as shown in (2.4) by introducing the slackvector $s = A^T x - b$, $s \geq 0$ to simplify notation.

$$Gx + g - A\lambda = 0 \tag{2.4a}$$
$$s - A^T x + b = 0 \tag{2.4b}$$
$$s_i \lambda_i = 0 \tag{2.4c}$$
$$(\lambda, s) \geq 0 \tag{2.4d}$$

We now define the function $F(x, \lambda, s)$ in (2.5) such that the roots of this function are solutions to the first three optimality conditions in (2.4). The residual vectors r_d, r_p and $r_{s\lambda}$ are also defined in these equations. We note that the third term in F is nonlinear and that $S = \text{diag}(s_1, s_2, ..., s_m)$ is a diagonal matrix with the elements of the slackvector s on the diagonal, $\Lambda = \text{diag}(\lambda_1, \lambda_2, ..., \lambda_m)$ is a diagonal matrix with the elements of the Lagrange multiplier vector on the

diagonal and $e = (1, 1, ..., 1)^T$ is a $m \times 1$ vector containing ones.

$$F(x, \lambda, s) = \begin{bmatrix} Gx - A\lambda + g \\ s - A^T x + b \\ S\Lambda e \end{bmatrix} = \begin{bmatrix} r_d \\ r_p \\ r_{s\lambda} \end{bmatrix} \tag{2.5}$$

Because we consider convex problems we know that G is positive semidefinite and furthermore that the contraints are affine which means that the optimality conditions stated above are both necessary and sufficient. Additionally the solution vector will be a global minimizer.
By applying Newton's method to $F(x, \lambda, s) = 0$ we obtain a search direction from the current iterate (x_k, λ_k, s_k), [1] p. 395, as shown in (2.6) where $J(x, \lambda, s)$ is the Jacobian of $F(x, \lambda, s)$.

$$J(x, \lambda, s) \begin{bmatrix} \Delta x \\ \Delta \lambda \\ \Delta s \end{bmatrix} = -F(x, \lambda, s) \tag{2.6}$$

Newton's method forms a linear model for F around the current point and obtains the search direction $(\Delta x, \Delta \lambda, \Delta s)$ by solving the system (2.6), Wright [2] p. 6.
Writing out the Jacobian we obtain the following system of equations shown in (2.7).

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ r_{s\lambda} \end{bmatrix} \tag{2.7}$$

If we solve this system iteratively we should reach the optimal point. In practice however this raw approach is not used because a full Newton step will generally be infeasible. Instead a line search is performed along the Newton direction and we find the new iterate as shown in (2.8) where k is the current iterate and $\alpha \in [0, 1]$ is the line search parameter (step length).

$$(x_{k+1}, \lambda_{k+1}, s_{k+1}) = (x_k, \lambda_k, s_k) + \alpha(\Delta x, \Delta \lambda, \Delta s) \tag{2.8}$$

One drawback of using Newton's method is that the nonnegativity constraints $(\lambda, s) \geq 0$ can not be taken directly into account. The difficulties in designing

an algorithm stems from satisfying these constraints.

2.2 Predictor-corrector method

In practice the predictor-corrector method proposed by Mehrotra is used. As indicated by the name there is both a predictor and a corrector step involved in the algorithm. In order to explain the purpose of each of these steps it is necessary to explain a few concepts first, including that of the central path. The basic idea is to solve the system (2.7) and then set up a second system of equations to correct this step by modifying the right hand side of (2.7).

2.2.1 Central Path and other concepts

The central path is defined in (2.9).

$$F(x_\tau, \lambda_\tau, s_\tau) = \begin{bmatrix} 0 \\ 0 \\ \tau e \end{bmatrix}, \quad (\lambda_\tau, s_\tau) > 0 \tag{2.9}$$

It is an arc (curve) of strictly feasible points that is parametrized by a scalar $\tau > 0$, [1] p. 397. We note that points on the central path are strictly feasible and that $r_d = 0$ and $r_p = 0$ for these points.
The idea is to have the iterates (x_k, λ_k, s_k) progress along this central path and to have τ decrease with each step. As τ approaches zero, the equations (2.9) will approximate the previously defined optimality conditions better and better. This means that the central path follows a path to the solution such that $(\lambda, s) > 0$ and such that the pairwise products $s_i\lambda_i$ are decreased to zero at the same rate. The purpose of using this concept is that we expect to obtain the fastest rate of convergence by having the iterates follow (approximate) the central path. From (2.9) we also see that the points on the central path satisfy a slightly perturbed version of the optimality conditions, [1] p. 481, with the only difference, compared to the optimality conditions, being the term τe on the right hand side.

In addition to the central path we also define the complementarity measure μ

shown in (2.10) and a centering parameter $\sigma \in [0, 1]$.

$$\mu = \frac{s^T \lambda}{m} \tag{2.10}$$

From (2.10) we see that the complementarity measure tells us something about the value of the pairwise products $s_i \lambda_i$. It tells us something about how useful a computed search direction is by determining if a computed step has or has not reduced $s^T \lambda$ significantly. If μ is reduced significantly little centering is needed and a value of σ that reflects this is chosen ($\sigma = 0$ in this case). If on the other hand μ is not reduced then the computed search direction is not that useful and a larger value of σ is chosen.
In practice we start by solving the system (2.11) obtaining the so-called affine scaling direction $(\Delta x^{\text{aff}}, \Delta \lambda^{\text{aff}}, \Delta s^{\text{aff}})$ and then determine a step length α^{aff} for this step. Note that we will use aff in superscript to mark various vectors and scalars associated with the computation of the affine scaling direction. The computation of $(\Delta x^{\text{aff}}, \Delta \lambda^{\text{aff}}, \Delta s^{\text{aff}})$ is also referred to as the predictor step.

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta \lambda^{\text{aff}} \\ \Delta s^{\text{aff}} \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ r_{s\lambda} \end{bmatrix} \tag{2.11}$$

Next the complementarity measure for the current iterate is computed along with a predicted complementarity measure μ^{aff} for the computed Newton step as indicated in (2.12).

$$\mu^{\text{aff}} = \frac{(s + \alpha^{\text{aff}} \Delta s^{\text{aff}})^T (\lambda + \alpha^{\text{aff}} \Delta \lambda^{\text{aff}})}{m} \tag{2.12}$$

By comparing the values of μ and μ^{aff} we determine if the computed affine scaling direction is a good search direction. If for example $\mu^{\text{aff}} << \mu$ we have a significant reduction in the complementarity measure and the search direction is good so little centering is needed.

In practice the centering parameter σ is computed using the formula (2.13) which has proven to work well in practice.

$$\sigma = \left(\frac{\mu^{\text{aff}}}{\mu} \right)^3 \tag{2.13}$$

As stated above the value of the centering parameter is chosen to reflect to which extent we need to center the search direction we have obtained with the predictor step.
If $\sigma = 0$ the standard Newton step is used. If on the other hand $\sigma = 1$ the step is used to center i.e. move the iterate closer to the central path and little or no progress is made in reducing the complementarity measure. This however enables us to take a longer step in the next iteration.
A visualization of how σ varies with μ^{aff} for a fixed μ is provided in figure 2.1.

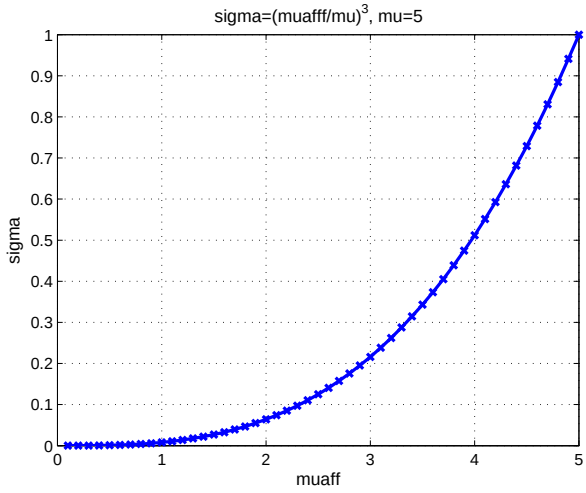

Figure 2.1: *Centering parameter as a function of the predicted complementarity measure.*

To set up a centering step for the algorithm we set $\tau = \sigma\mu$ (Newton step toward the point where $s_i\lambda_i = \sigma\mu$) and get the system shown in (2.14).

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x^{\text{cen}} \\ \Delta\lambda^{\text{cen}} \\ \Delta s^{\text{cen}} \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ r_{s\lambda} - \sigma\mu e \end{bmatrix} \tag{2.14}$$

Before stating the corrector step we will briefly see how to compute the step length.

2.2.2 Step length computation

After computing a search direction for the predictor step, and later on the corrector step, we need to decide how long a step we can take in the computed search direction so that we do not violate $(\lambda, s) > 0$.
Several approaches can be applied to computing the step length and different approaches influence how rapidly the algorithm will converge. In this presentation we use the same approach in choosing a scaling parameter for both the predictor and corrector step. To begin with we use a simple approach by scaling the step in the search direction using a single parameter $\alpha \in [0, 1]$ for all three directions $(\Delta x, \Delta\lambda, \Delta s)$ as shown in (2.15).

$$(x_{k+1}, \lambda_{k+1}, s_{k+1}) = (x_k, \lambda_k, s_k) + \alpha(\Delta x, \Delta\lambda, \Delta s) \tag{2.15}$$

In the following we will use the choice of a scaling parameter $\alpha^{\text{aff}} \in [0, 1]$ for the affine scaling direction as an example.
Specifically we want to choose the largest α^{aff} such that (2.16) and (2.17) are satisfied.

$$\lambda + \alpha^{\text{aff}}\Delta\lambda^{\text{aff}} \geq 0 \tag{2.16}$$
$$s + \alpha^{\text{aff}}\Delta s^{\text{aff}} \geq 0 \tag{2.17}$$

We will look at these equations seperately and for convenience use two seperate scaling parameters, $\alpha_\lambda^{\text{aff}}$ for (2.16) and α_s^{aff} for (2.17), such that $\alpha^{\text{aff}} = \min(\alpha_\lambda^{\text{aff}}, \alpha_s^{\text{aff}})$.
For each of the equations there are three cases based on the sign of $\Delta\lambda^{\text{aff}}$ and Δs^{aff} respectively. These cases are shown in (2.18) for (2.16).

$$\Delta\lambda^{\text{aff}} > 0 : \alpha_\lambda^{\text{aff}} = 1 \tag{2.18a}$$
$$\Delta\lambda^{\text{aff}} = 0 : \alpha_\lambda^{\text{aff}} = 1 \tag{2.18b}$$
$$\Delta\lambda_i^{\text{aff}} < 0 : \lambda_i + \alpha_\lambda^{\text{aff}}\Delta\lambda_i^{\text{aff}} = 0 \Rightarrow \alpha_\lambda^{\text{aff}} = \frac{-\lambda_i}{\Delta\lambda_i} \tag{2.18c}$$

If all the elements of the vector $\Delta\lambda^{\text{aff}}$ are positive or equal to zero (2.16) is satisfied because $\lambda > 0$ and $\alpha^{\text{aff}} \in [0, 1]$. So in these cases the maximum

allowable step is obtained by setting $\alpha^{\text{aff}} = 1$.
If on the other hand some of the elements of $\Delta\lambda^{\text{aff}}$ are negative we need to choose $\alpha_\lambda^{\text{aff}}$ such that its value accommodates the smallest value in the vector $\Delta\lambda^{\text{aff}}$.

From the above cases we see that $\alpha_\lambda^{\text{aff}}$ can be chosen as indicated in (2.19).

$$\alpha_\lambda^{\text{aff}} = \min_{i:\Delta\lambda_i<0} \left(1, \min \frac{-\lambda_i}{\Delta\lambda_i^{\text{aff}}}\right) \tag{2.19}$$

The same arguments can be used to set up the three cases for (2.17) and α_s^{aff} is chosen equivalently to $\alpha_\lambda^{\text{aff}}$ as shown in (2.20).

$$\alpha_s^{\text{aff}} = \min_{i:\Delta s_i<0} \left(1, \min \frac{-s_i}{\Delta s_i^{\text{aff}}}\right) \tag{2.20}$$

As mentioned previously α^{aff} can then be chosen as $\alpha^{\text{aff}} = \min(\alpha_\lambda^{\text{aff}}, \alpha_s^{\text{aff}})$.
The described approach for determining a step length for the predictor step is also employed to determine a step length α for the corrector step.
Later on we will consider if the use of more than one scaling parameter will improve the performance of the algorithm.

2.2.3 Corrector step

Having stated the purpose of the central path, the complementarity measure and the centering parameter we are now ready to look at which modifications to make to the right hand side of (2.7) to set up the corrector step.
As stated earlier the algorithm begins by solving (2.21).

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta\lambda^{\text{aff}} \\ \Delta s^{\text{aff}} \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ r_{s\lambda} \end{bmatrix} \tag{2.21}$$

A full step in this direction yields (because of the third block row in (2.21))

$$(s_i + \Delta s_i^{\text{aff}})(\lambda_i + \Delta\lambda_i^{\text{aff}}) = \tag{2.22}$$

$$s_i\lambda_i + s_i\Delta\lambda_i^{\text{aff}} + \lambda_i\Delta s_i^{\text{aff}} + \Delta s_i^{\text{aff}}\Delta\lambda_i^{\text{aff}} = \Delta s_i^{\text{aff}}\Delta\lambda_i^{\text{aff}} \tag{2.23}$$

The term $\Delta s_i^{\text{aff}}\Delta\lambda_i^{\text{aff}} \neq 0$ so by taking this affine scaling step we introduce a linearization error. To compensate for this error we use a corrector step and solve the system (2.24) where $\Delta S^{\text{aff}} = \text{diag}(\Delta s_1^{\text{aff}}, \ldots, \Delta s_m^{\text{aff}})$ and $\Lambda^{\text{aff}} = \text{diag}(\Delta\lambda_1^{\text{aff}}, \ldots, \Delta\lambda_m^{\text{aff}})$.

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x^{\text{cor}} \\ \Delta\lambda^{\text{cor}} \\ \Delta s^{\text{cor}} \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ r_{s\lambda} + \Delta S^{\text{aff}}\Delta\Lambda^{\text{aff}}e \end{bmatrix} \tag{2.24}$$

So to summarize we obtain an affine scaling direction by solving the system (2.21). This predictor step is used to compute the centering parameter and to define the right hand side for the corrector and centering step, [1] p. 408.
The system we solve in practice to obtain the search direction contains both the centering and corrector contributions as shown in (2.25).

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ r_{s\lambda} + \Delta S^{\text{aff}}\Delta\Lambda^{\text{aff}}e - \sigma\mu e \end{bmatrix} \tag{2.25}$$

Note that the coefficient matrix for the system solved in the predictor step and the coefficient matrix for the system solved in the corrector step are identical such that we only need to compute a factorization of this matrix once. This observation is an essential feature of the predictor-corrector method.
Furthermore we note that the predictor step optimizes by reducing complementarity products and that the corrector step keeps the current iterate away from the boundary of the feasible region and close to the central path, [7] p. 137.

2.2.4 Stopping criteria for the PC method

To terminate the sequence of iterations, and thereby the method, we need to define some stopping criteria to detect when the method has reached the solution. In this thesis we use the simple stopping criteria suggested by Jørgensen

[3].
First of all we introduce a maximum number of iterations M to ensure that the algorithm stops. A number in the range 50-200 is chosen as we expect the algorithm to converge fairly quickly because each iteration can make significant progress towards the solution.
In addition to this we introduce a lower bound ϵ on the norms of the residual vectors r_d and r_p.

$$r_d = Gx + g - A\lambda$$
$$r_p = s - A^T x + b$$

$$\|r_d\| \geq \epsilon \tag{2.26}$$
$$\|r_p\| \geq \epsilon \tag{2.27}$$

The purpose of defining a lower tolerance for the residual vectors is that we want the program to stop if it has found a pair of vectors (x, λ) or a vector s that fulfils the optimality conditions (2.28) with satisfactory numerical precision.

$$Gx + g - A\lambda = 0 \tag{2.28a}$$
$$s - A^T x + b = 0 \tag{2.28b}$$

We also introduce a lower bound on the absolute value of the complementarity measure to ensure that the algorithm stops if the complementarity products are reduced to zero or a value close to zero.

$$\mu = \frac{s^T \lambda}{m}$$

$$|\mu| \geq \epsilon \tag{2.29}$$

It is possible to use more advanced stop criteria and more advanced criteria are in some special cases necessary for the algorithm to converge. In this presentation the stopping criteria introduced above are sufficient.

2.2.5 Algorithm

The full algorithm is stated below.
Note that the algorithm requires a starting point (x_0, λ_0, s_0). The starting point does not need to be in the feasible region. This is accomplished by requiring that $(\lambda_0, s_0) > 0$ and having the right hand side contain the residual vectors r_d and r_p instead of zeros to prevent infeasibility, [2] pp. 11-12.

- *Input*
  (x_0, λ_0, s_0) and G, A, g, b
- *Compute residuals and complementarity measure*

$$\begin{aligned} r_d &= Gx_0 + g - A\lambda_0 \\ r_p &= s_0 - A^T x_0 + b \\ r_{s\lambda} &= S_0 \Lambda_0 e \end{aligned}$$

$$\mu = \frac{s_0^T \lambda_0}{m}$$

- *Start while loop (terminate if stopping criteria are satisfied)*
- *Predictor step:*
  Solve (5.1.3) *to obtain an affine scaling direction* $(\Delta x^{\text{aff}}, \Delta\lambda^{\text{aff}}, \Delta s^{\text{aff}})$ *for setting up the right hand side for the corrector step and to obtain a good value of the centering parameter* σ

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta\lambda^{\text{aff}} \\ \Delta s^{\text{aff}} \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ r_{s\lambda} \end{bmatrix} \tag{2.30}$$

- *Compute* α^{aff}

$$\begin{aligned} \lambda + \alpha^{\text{aff}} \Delta\lambda^{\text{aff}} &\geq 0 \\ s + \alpha^{\text{aff}} \Delta s^{\text{aff}} &\geq 0 \end{aligned}$$

- *Compute* μ^{aff}

$$\mu^{\text{aff}} = \frac{(s + \alpha^{\text{aff}} \Delta s^{\text{aff}})^T (\lambda + \alpha^{\text{aff}} \Delta\lambda^{\text{aff}})}{m}$$

- *Compute centering parameter* σ

$$\sigma = \left(\frac{\mu^{\text{aff}}}{\mu} \right)^3$$

- *Corrector and centering step:*
  Solve (2.31) *to obtain search direction* $(\Delta x, \Delta\lambda, \Delta s)$

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ r_{s\lambda} + \Delta S^{\text{aff}} \Delta\Lambda^{\text{aff}} e - \sigma\mu e \end{bmatrix} \tag{2.31}$$

- *Compute* α

$$\lambda + \alpha\Delta\lambda \geq 0$$
$$s + \alpha\Delta s \geq 0$$

- *Update* (x, λ, s)
- *Update residuals and complementarity measure*

$$r_d = Gx + g - A\lambda$$
$$r_p = s - A^T x + b$$
$$r_{s\lambda} = S\Lambda e$$
$$\mu = \frac{s^T\lambda}{m}$$

- *End while loop*

2.3 Implementation and Test

In this section we describe some practical details concerning the implementation of the interior-point method for a QP with inequality constraints and test the implementation with a simple test problem to validate that the implementation works as intended and to see how well it performs.

2.3.1 Implementation

In practice we will not solve the full system (2.32) but instead split the system into three seperate equations such that we solve one of those equations and use back substitution to solve the remaining two equations.

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ r_{s\lambda} \end{bmatrix} \tag{2.32}$$

The second block row gives us the equation (2.33).

$$\Delta s = -r_p + A^T \Delta x \tag{2.33}$$

The third block row coupled with (2.33) gives us (2.34). We note that S is easy to invert because it is a diagonal matrix with positive entries.

$$\Delta\lambda = -S^{-1}(r_{s\lambda} + \Lambda\Delta s) = S^{-1}(-r_{s\lambda} + \Lambda r_p) - S^{-1}\Lambda A^T \Delta x \tag{2.34}$$

Using the first block row with (2.34) gives us (2.35).

$$\begin{aligned} -r_d &= G\Delta x - A\Delta\lambda && \text{(2.35a)} \\ &= (G + AS^{-1}\Lambda A^T)\Delta x - AS^{-1}(-r_{s\lambda} + \Lambda r_p) \Rightarrow && \text{(2.35b)} \\ -r_d &= \bar{G}\Delta x + \bar{r} && \text{(2.35c)} \end{aligned}$$

where

$$\begin{aligned} \bar{G} &= G + A(S^{-1}\Lambda)A^T = G + ADA^T \\ \bar{r} &= A\left(S^{-1}(r_{s\lambda} - \Lambda r_p)\right) \end{aligned}$$

We can therefore solve the system (2.32) by first determining Δx from (2.36) and then use back substitution to compute $\Delta\lambda$ and Δs from (2.34) and (2.33).

$$\bar{G}\Delta x = -\bar{g} = -(r_d + \bar{r}) \tag{2.36}$$

Note that the above equations are intended for a general case where we do not take the structure of for example the inequality constraints into consideration. If for example the inequalities where of the form $b_l \leq Ax \leq b_u$ we would not want to perform the computation $AD_lA^T + AD_uA^T$ as it would be more benificial to first add the diagonal matrices $D_l + D_u$ and then multiply them with A and A^T.

In practice we compute a Cholesky factorization of $\bar{G}$, to solve (2.36), as shown in (2.37) where L is a lower triangular matrix.

$$\bar{G} = LL^T \tag{2.37}$$

So that we solve the two equations in (2.38).

$$LY = -\bar{g} \tag{2.38a}$$
$$L^T \Delta x = Y \tag{2.38b}$$

It is important to note that we only have to perform the computationally expensive factorization of the matrix once per iteration because the matrix $\bar{G}$ does not change during a single iteration.

In practice we do not take a full step but use a scaling parameter η to dampen the step to ensure convergence as indicated in (2.40).

$$(x_{k+1}, \lambda_{k+1}, s_{k+1}) = (x_k, \lambda_k, s_k) + \eta\alpha(\Delta x, \Delta\lambda, \Delta s) \tag{2.39}$$

The value of η is essential for the algorithm converging or not converging. If the value is too large we will have non-convergence, a value that is too small will lead to slow convergence so one has to determine a value such that the trade off is acceptable. In this case $\eta = 0.95$ seems to be a good choice. Larger values leads to non-convergence.
The described algorithm has been implemented in MATLAB as the function `pcQP_simplev1.m` and is shown in appendix B.1.

2.3.2 Test

To test the implemented algorithm we set up a simple test problem of the form (2.40) to which we know the solution.

$$\min_{x\in\mathbb{R}^n} \quad f(x) = \frac{1}{2}x^T Gx + g^T x \tag{2.40a}$$
$$\text{s.t.} \quad l_{vec} \leq x \leq u_{vec} \tag{2.40b}$$

This problem is known as a boxconstrained QP for obvious reasons.
We let G be the identity matrix of size $n \times n$, let g be a randomly generated vector of size $n \times 1$ and define a lower and an upper bound, l_{vec} and u_{vec} on x such that $l_{vec} \leq x \leq u_{vec}$ where l_{vec} is a vector of size $n \times 1$ containing the scalar l n times and u_{vec} is a vector of size $n \times 1$ containing the scalar u n times. The problem is well suited as an initial test problem because we know that the solution is $-g$ and because we can vary the size of the problem.
To formulate the test problem in the notation introduced previously (2.41) we get the matrix A and the vector b of size $2n \times 1$ shown in (2.42), where I is the identity matrix of size $n \times n$.

$$\min_{x\in\mathbb{R}^n} \quad f(x) \quad = \frac{1}{2}x^T Gx + g^T x \tag{2.41a}$$
$$\text{s.t.} \quad A^T x \geq b \tag{2.41b}$$

$$A = [I,\ -I] \tag{2.42a}$$
$$b = [l_{vec}; -u_{vec}] \tag{2.42b}$$

To validate that the implementation works as intended we set up the test problem in MATLAB and use the implemented function `pcQP_simplev1.m` to solve it.
Using 10 variables and the inputs stated in table 2.1, i.e. starting points x_0, λ_0 and s_0, where λ_0 and s_0 have lengths $2n$. tolerance ϵ, maximum number of iterations M, we get the result shown in table 2.2.

To generate the random vector g we used the MATLAB function `randn` but specified a particular random vector, via the option 'seed', so that the results can be reproduced. The implemented algorithm used 17 iterations in this case.

From table 2.2 we see that the implemented algorithm finds the expected solution $-g$ with machine precision and it would therefore seem that the implemented algorithm works as intended.

n	10
x_0	$[0, 0, \ldots, 0]^T$
λ_0	$[1, 1, \ldots, 1]^T$
s_0	$[1, 1, \ldots, 1]^T$
ϵ	1e-16
M	100
seed	100

Table 2.1: *Inputs for test of implemented predictor-corrector method for inequality constrained QP.*

i'th element	$x(i)$	$-g(i)$	$x(i) - (-(g(i))$
1	-0.9085	-0.9085	0
2	2.2207	2.2207	0
3	0.2391	0.2391	0
4	-0.0687	-0.0687	0
5	2.0202	2.0202	0
6	0.3641	0.3641	0
7	0.0813	0.0813	0
8	1.9797	1.9797	0
9	-0.7882	-0.7882	0
10	-0.7366	-0.7366	0

Table 2.2: *Result from test of predictor-corrector method for inequality constrained QP.*

The MATLAB script `test1.m`, used to generate these results, can be seen in appendix B.5.

It would also be interesting to see how the algorithm performs for an increasingly larger amount of variables. In figure (2.2) we have plotted computation time as a function of n using the simple test problem introduced above with an increasingly larger number of variables.

From figure 2.2 we see that the computation time required to solve the problem seems to rise rather steeply as n increases to 1000. This emphasizes the importance of making the program code as efficient as possible to avoid any sort of unnecessary computation and the importance of being able to reuse the factorization when solving the systems for the predictor and corrector step.
In figure 2.3 the number of iterations are plotted as a function of the number of variables with the purpose of seeing if the algorithm is consistently using few iterations when $n \to \infty$.

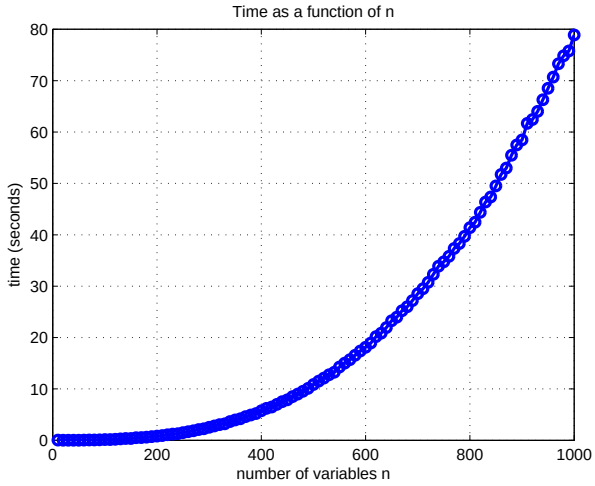

Figure 2.2: *Plot of time in seconds as a function of the number of variables.*

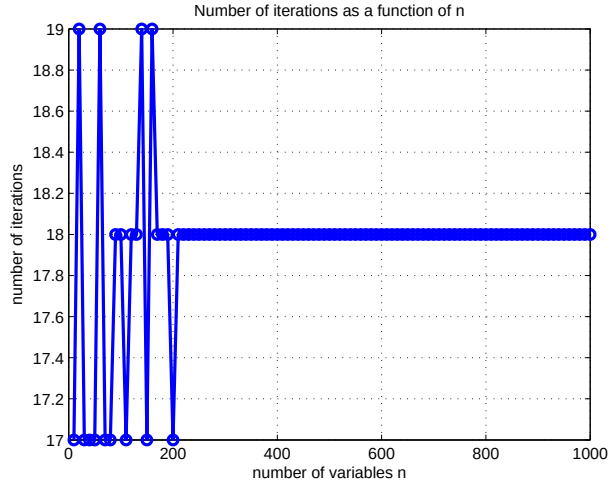

Figure 2.3: *Plot of number of iterations as a function of the number of variables.*

Even though the test problem is simple it is still able to provide us some insight into what happens when the number of variables become large. From figure 2.3 we see that the number of iterations are quite consistent, ranging from 17-19 iterations with most runs using 18 iterations to determine a solution. For more complicated problems however it is not certain that the algorithm will use as

few iterations as in the above example.

One of the drawbacks of Mehrotra's predictor-corrector method is that it might encounter some difficulties if it is trying to approximate a central path that has some very sharp turns. This might result in a poor or incorrect local approximation of the central path and ultimately lead to non-convergence, [2] p. 203. Furthermore we assume that a full step is possible in the corrector direction which is not always the case. We also assume that it is possible to correct all complementarity products to the same value which can sometimes be a too agressive approach. There has also been reports of the method behaving erratically when a predictor direction is used from highly infeasible and not well-centered points, [8] p. 2, p. 5.

In this chapter the techniques in designing an interior-point method for inequality constrained quadratic programs were introduced and a method was implemented in MATLAB. The system of equations in the predictor and corrector part of the method could be solved by splitting the system into three seperate equations. Using a Cholesky factorization to solve one of the equations allowed the use of back substitution to solve the remaining two equations. Using a simple test problem the implementation was tested and it was concluded that the implemented program was working as intended. Furthermore the computation time and the number of iterations were investigated for an increasingly larger number of variables. This showed that the number of iterations was consistently low and that the computation time rose steeply when the number of variables was increased.

Chapter 3

General QP

In this chapter we extend the method introduced in the previous chapter to solve a QP with both equality and inequality constraints.
As stated in the previous chapter the extensions from the inequality constrained QP case to the general QP case are fairly trivial so comments will be sparse with reference to the previous chapter. The development of a method for a QP with inequality and equality constraints follows the approach suggested by Jørgensen [3].

3.1 Optimality conditions

The general QP is stated in (3.1) where A is now a $n \times m_A$ matrix describing the equality constraints and C is a $n \times m_C$ matrix describing the inequality constraints. b and d are $m_A \times 1$ and $m_C \times 1$ vectors respectively. m_A is the number of equality constraints and m_C is the number of inequality constraints.

$$\min_{x \in \mathbb{R}^n} \quad \frac{1}{2} x^T G x + g^T x \tag{3.1a}$$
$$\text{s.t.} \quad A^T x = b \tag{3.1b}$$
$$C^T x \geq d \tag{3.1c}$$

In (3.2) the Lagrangian $L(x, y, z)$ is stated where y is the $m_A \times 1$ vector containing the Lagrange multipliers for the equality constraints and z is the $m_C \times 1$ vector containing the Lagrange multipliers for the inequality constraints.

$$L(x, y, z) = \frac{1}{2} x^T G x + g^T x - y^T (A^T x - b) - z^T (C^T - d) \tag{3.2}$$

For the general QP we get the following optimality conditions.

$$\nabla_x L(x, y, z) = Gx + g - Ay - Cz = 0 \tag{3.3a}$$
$$A^T x - b = 0 \tag{3.3b}$$
$$C^T x - d \geq 0 \tag{3.3c}$$
$$z \geq 0 \tag{3.3d}$$
$$z_i (C^T x - d)_i = 0, \quad i = 1, 2, \ldots, m_C \tag{3.3e}$$

By introducing the slackvector $s = C^T x - d \geq 0$ the optimality conditions are rewritten.

$$Gx + g - Ay - Cz = 0 \tag{3.4a}$$
$$A^T x - b = 0 \tag{3.4b}$$
$$s - C^T x + d = 0 \tag{3.4c}$$
$$(z, s) \geq 0 \tag{3.4d}$$
$$s_i z_i = 0 \tag{3.4e}$$

Once again we introduce a function F whose roots coincide with the solution to the optimality conditions and furthermore define the residual vectors as shown

in (3.5).

$$F(x,y,z,s) = \begin{bmatrix} Gx + g - Ay - Cz \\ A^T x - b \\ s - C^T x + d \\ SZe \end{bmatrix} = \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{sz} \end{bmatrix} \tag{3.5}$$

Newton's method is used to obtain the following system of equations.

$$J(x,y,z,s) \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = -F(x,y,z,s) \tag{3.6}$$

Writing out the Jacobian $J(x,y,z,s)$ we get the system (3.7).

$$\begin{bmatrix} G & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{sz} \end{bmatrix} \tag{3.7}$$

3.2 Predictor-corrector method

Again we want the iterates to follow the central path so again we set up a predictor and a corrector step.
The predictor step then becomes the solution of the system (3.8).

$$\begin{bmatrix} G & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta y^{\text{aff}} \\ \Delta z^{\text{aff}} \\ \Delta s^{\text{aff}} \end{bmatrix} = - \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{sz} \end{bmatrix} \tag{3.8}$$

After computing the search direction $(\Delta x^{\text{aff}}, \Delta y^{\text{aff}}, \Delta z^{\text{aff}}, \Delta s^{\text{aff}})$ we need to determine a step length α^{aff}. This is also done equivalently to the inequality constrained QP case as shown in (3.9).

$$z + \alpha^{\text{aff}} \Delta z^{\text{aff}} \geq 0 \tag{3.9a}$$
$$s + \alpha^{\text{aff}} \Delta s^{\text{aff}} \geq 0 \tag{3.9b}$$

Computing the complementarity measure μ in (3.10) and the predicted complementarity measure μ^{aff} in (3.11) then enables us to compute the centering parameter σ in (3.12).

$$\mu = \frac{s^T z}{m_C} \tag{3.10}$$

$$\mu^{\text{aff}} = \frac{(s + \alpha^{\text{aff}} \Delta s^{\text{aff}})^T (z + \alpha^{\text{aff}} \Delta z^{\text{aff}})}{m_C} \tag{3.11}$$

$$\sigma = \left(\frac{\mu^{\text{aff}}}{\mu} \right)^3 \tag{3.12}$$

The value of σ will once again reflect how much centering is needed for the current iterate.
Including the corrector term $\Delta S^{\text{aff}} \Delta Z^{\text{aff}} e$ to compensate for linearization errors, we arrive at the corrector step (3.13).

$$\begin{bmatrix} G & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{sz} + \Delta S^{\text{aff}} \Delta Z^{\text{aff}} e - \sigma \mu e \end{bmatrix} \tag{3.13}$$

The step length α is obtained by satisfying the equations (3.14).

$$z + \alpha \Delta z \geq 0 \tag{3.14a}$$
$$s + \alpha \Delta s \geq 0 \tag{3.14b}$$

Compared to the inequality constrained case, we use one additional stopping criteria, because of the extra residual vector, such that we use the four criteria listed in (3.15) and (3.16) plus a maximum number of iterations.

$$\|r_L\| \geq \epsilon \tag{3.15a}$$
$$\|r_A\| \geq \epsilon \tag{3.15b}$$
$$\|r_C\| \geq \epsilon \tag{3.15c}$$

$$|\mu| \geq \epsilon \tag{3.16}$$

Finally the next iterate is obtained by damping the step with a scalar η.

$$(x_{k+1}, y_{k+1}, z_{k+1}, s_{k+1}) = (x_k, y_k, z_k, s_k) + \eta\alpha(\Delta x, \Delta y, \Delta z, \Delta s) \tag{3.17}$$

3.3 Implementation and Test

The purpose of this section is to explain important practical details to consider when implementing the described interior-point method for the general QP and to test the implementation.

3.3.1 Implementation

One major difference between the inequality constrained case and the general case, lies in how to solve the systems for the predictor and corrector step.
For the inequality constrained case we were able to reduce the system and use backward substitution to compute each seach direction. We are also able to reduce the system in the general case but the resulting systems are a bit more complicated than in the inequality constrained case. To illustrate the procedure for both the predictor and the corrector step we use the system (3.18).

$$\begin{bmatrix} G & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{sz} \end{bmatrix} \tag{3.18}$$

The fourth block row gives

$$S\Delta z + Z\Delta s = -r_{sz} \Rightarrow \Delta s = -Z^{-1}(r_{sz} + S\Delta z) \tag{3.19}$$

Using the third block row coupled with (3.19) we get

$$-C^T\Delta x\Delta s = -r_C \Rightarrow -C^T\Delta x - Z^{-1}(r_{sz} + S\Delta z) = -r_C \Rightarrow \tag{3.20a}$$
$$-C^T\Delta x - Z^{-1}S\Delta z = -r_C + Z^{-1}r_{sz} \tag{3.20b}$$

Rewriting this we arrive at the augmented system (3.21).

$$\begin{bmatrix} G & -A & -C \\ -A^T & 0 & 0 \\ -C^T & 0 & -Z^{-1}S \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = - \begin{bmatrix} r_L \\ r_A \\ r_C - Z^{-1}r_{sz} \end{bmatrix} \tag{3.21}$$

The third block row from this augmented system now gives

$$-C^T\Delta x - Z^{-1}S\Delta z = -r_C + Z^{-1}r_{sz} \Rightarrow \tag{3.22a}$$
$$\Delta z = -(S^{-1}Z)C^T\Delta x + (S^{-1}Z)(r_C - Z^{-1}r_{sz}) \tag{3.22b}$$

Using (3.22) with the first block row from the augmented system results in (3.23).

$$G\Delta x - A\Delta y - C\Delta z = -r_L \Rightarrow \tag{3.23a}$$
$$G\Delta x - A\Delta y - C\left(-(S^{-1}Z)C^T\Delta x + (S^{-1}Z)(r_C - Z^{-1}r_{sz})\right) = -r_L \Rightarrow \tag{3.23b}$$
$$(G + C(S^{-1}Z)C^T)\Delta x - A\Delta y = -r_L + C(S^{-1}Z)(r_C - Z^{-1}r_{sz}) \tag{3.23c}$$

(3.23) can now be used to set up a new augmented system (3.24).

$$\begin{bmatrix} G + C(S^{-1}Z)C^T & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = - \begin{bmatrix} r_L - C(S^{-1}Z)(r_C - Z^{-1}r_{sz}) \\ r_A \end{bmatrix} \tag{3.24}$$

We can not reduce the augmented system (3.24) further so we stop here.
The purpose of using an augmented system is that it has advantages in stability and flexibility, [2] p. 211. There are however also some important drawbacks if we choose to solve the augmented system (3.24). Mainly that very small and very large elements in the coefficient matrix might occur in the last iterations potentially causing numerical instability, [2] p. 211, p. 229.
Both coefficient matrices in the augmented systems, (3.21) and (3.24), are indefinite so a Cholesky factorization can not be used to solve these systems. In practice we will not solve the second augmented system because it might cause numerical instability and destroy sparsity preservation, [2] p. 229. We will therefore focus on solving the first augmented system (3.21) or the full system (3.18).
The algorithm has been implemented in MATLAB as the function `pcQP_gen.m` and can be seen in appendix B.1.

3.3.2 Test

To verify that the implementation works as intended we will be using a couple of test problems from a collection of convex QP problems, Maros and Meszaros [4]. All QP problems in this test set are available in QPS format which is an extension of the MPS format. MATLAB however does not have a built-in QPS reader. So to use the QPS files as test problems it was necessary to construct a QPS reader for MATLAB that fits our specific needs. The MATLAB QPS reader `QPSreader.m` can be seen in appendix B.4. In the next chapter we provide an overview of the QPS format and the implemented reader.
Note that the collection of test problems contains information about each test problem including the solution (optimal value f) to each the problem. This makes it easy to test our implementation because we simply have to compare the value $f(x) = \frac{1}{2}x^TGx+g^Tx$ found by the implemented interior-point method with the optimal value listed in [4].

To validate that the implementation works as intended we test `pcQP_gen.m` by solving the two test problems `cvxqp1_m` and `cvxqp1_s` from the test set. The test is also used to decide which of the two systems, (3.18) or (3.21), to solve and how to solve them, i.e which factorization to use.
In table 3.1 the results for each test problem is given where n is the number of variables, 'Solution' is the optimal solution provided in the repository [4], '`pcQP_gen.m`' is the solution found by the program, 'Its' is the number of iterations used and 'Time' is the amount of computation time (in seconds) spent. A LU factorization of the coefficient matrix in the full system (3.18) was applied to obtain the results in table 3.1.

QP problem	Solution	pcQP_gen.m	Its	Time
cvxqp1_m	0.108751157e+07	0.108751157e+07	29	1650
cvxqp1_s	0.115907181e+05	0.115907181e+05	23	3

Table 3.1: *Test problem results using LU factorization.*

From table 3.1 we see that the implemented algorithm finds the expected solution with satisfying numerical precision and we therefore assume that the algorithm is working as intended. The two problems are very different in regards to the number of variables, 1000 and 100 respectively, which influences the amount of computation time spent, such that the program spent around 1650 seconds computing the solution for `cvxqp1_m` and only around 3 seconds computing the solution for `cvxqp1_s`.
We repeat the above test runs solving the augmented system (3.21) using a LDL^T factorization. A LDL^T factorization of the symmetric indefinite matrix T has the form (3.25), where P is a permutation matrix, L is a unit lower triangular matrix and D is a block diagonal matrix, [8] p. 221.

$$PTP^T = LDL^T \tag{3.25}$$

The results are shown in table 3.2.

QP problem	Solution	pcQP_gen.m	Its	Time
cvxqp1_m	0.108751157e+07	0.108751157e+07	22	21.20
cvxqp1_s	0.115907181e+05	0.115907181e+05	16	0.13

Table 3.2: *Test problem results using a LDL^T factorization.*

From table 3.2 we see that we find the correct solution again and we therefore conclude that performing a LDL^T factorization of the system (3.21) also works as intended. There is one major difference between the two approaches however. When solving `cvxqp1_m` using (3.18) and a LU factorization the computing time was around 1650 seconds. With the LDL^T approach the computation time was only around 21 seconds. Taking this into consideration it would seem much more profitable to use the second approach and solve the augemted system (3.21) using the LDL^T factorization. A reduction in computing time for `cvxqp1_s` was also achieved, going from around 3 seconds to less than 1 second.

In this chapter the predictor-corrector method was extended to solve QP problems with both inequality and equality constraints. To solve the problems in practice we found that it is most profitable to set up an augmented system and solve this system using a LDL^T factorization. The method was implemented in

MATLAB and tested using a few problems from a collection of QP test problems.

Chapter 4

QPS reader

We develop a QPS reader with the purpose of extracting QP test problems such that these test problems can be used to test the developed interior-point methods. The QPS reader plays an important role in this thesis because it allows us to test the developed interior-point methods and compare their computational performance. The lack of access to QP's with both inequality and equality constraints was what motivated the implementation of a QPS reader for MATLAB. In this chapter we describe how the QPS reader was designed and implemented.

4.1 The QPS format

The QPS format was developed by Maros and Meszaros [4] as a standard format for defining a QP problem. It is an extension to the MPS format, the standard format for defining a LP problem, which was developed by IBM and based on punch cards. In addition to developing the QPS format Maros and Meszaros also compiled a test set of 138 QP problems available in QPS format. In [4] solutions for each of the 138 QP problems are provided.
In their presentation of the QPS format Maros and Meszaros assume that the QP problem has the form (4.1), where G is symmetric and positive semidefinite and $A \in \mathbb{R}^{m \times n}$.

$$\min_{x \in \mathbb{R}^n} \quad f(x) = g_0 + g^T x + \frac{1}{2} x^T G x \tag{4.1a}$$
$$s.t. \quad Ax = b \tag{4.1b}$$
$$l \leq x \leq u \tag{4.1c}$$

A QPS file consists of eight parts seperated by the keywords listed in table 4.1.

Keyword	Description
`NAME`	Name of the QP problem
`ROWS`	Names and types of constraints
`COLUMNS`	Entries of the constraintmatrix
`RHS`	Entries in the right hand side vector
`RANGES`	Double inequalities
`BOUNDS`	Specifies different types of bounds on individual variables
`QUADOBJ`	Lower triangular part of G
`ENDATA`	End of QPS file

Table 4.1: *QPS file keywords.*

For further information on the sections in a QPS file see [4], [5] and the next section.
To test the MATLAB implementation of our QPS reader we will be using the example `qptest` introduced in [4] and stated in (4.2).

$$\min \quad f(x,y) = 4 + 1.5x - 2y + 0.5(8x^2 + 2xy + 2yx + 10y^2) \tag{4.2a}$$
$$s.t. \quad 2x + y \geq 2 \tag{4.2b}$$
$$-x + 2y \leq 6 \tag{4.2c}$$
$$0 \leq x \leq 20, \quad y \geq 0 \tag{4.2d}$$

The QPS format version `qptest.qps` of (4.2) is given below.

```
NAME                QPexample
ROWS
  N OBJ.FUNC
  G R------1
  L R------2
COLUMNS
```

```
    C------1  R------1  0.200000e+01   R------2  -.100000e+01
    C------1  OBJ.FUNC  0.150000e+01
    C------2  R------1  0.100000e+01   R------2  0.200000e+01
    C------2  OBJ.FUNC  -.200000e+01
RHS
    RHS1      OBJ.FUNC  -.400000e+01
    RHS1      R------1  0.200000e+01   R------2  0.600000e+01
RANGES
BOUNDS
 UP BND1      C------1  0.200000e+02
QUADOBJ
    C------1  C------1  0.800000e+01
    C------1  C------2  0.200000e+01
    C------2  C------2  0.100000e+02
ENDATA
```

Each line in a QPS file is divided into fields as shown in table 4.2, [4] p. 4.

Field	Position
F1	2-3
F2	5-12
F3	15-22
F4	25-36
F5	40-47
F6	50-61

Table 4.2: *QPS fields.*

As an example consider the following line from the example above.

```
 UP BND1      C------1  0.200000e+02
```

The bound type (in this case `UP`) appears in position 2-3, the name of the bound in position 5-12, the variable name in position 15-22 and the value of the bound in position 25-36.

Note that the constant g_0 is given as $-g_0$ in the section `RHS`. The `RANGES` section is empty for this problem and there is only a single bound present in the `BOUNDS` section specified for the variable `C------1`. Also note that the entries in the vector g are listed in `COLUMNS`.
The example was chosen because of its simple nature making it easy to compare the elements in the QP problem with the entries in the QPS file and eventually the output from the implemented QPS reader.

4.2 Implementation and Test

The implemented QPS reader `QPSreader.m` takes a filename as input and contains both the main function `QPSreader` and a subfunction `ReadQPSdimensions` as shown in figure 4.1.

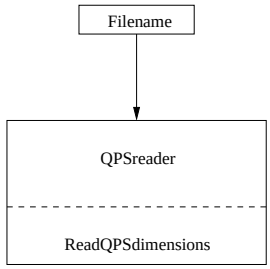

Figure 4.1: *Structure of implemented QPS reader.*

The MATLAB function `QPSreader.m` can be seen in appendix B.4.

Before beginning the actual extraction of entries from the QPS file, `ReadQPSDimensions` is used to scan the QPS file for the dimensions of the QP problem. This allows us to preallocate space for various vectors and matrices used in the actual QPS reader and thereby save computation time. The output from `ReadQPSDimension` is the number of variables n, the number of constraints m of the form $Ax = b$, $Ax \leq b$ or $Ax \geq b$ (all included in the constraint matrix A_{raw}), the number of nonzeros nz in the constraint matrix and the number of off-diagonal entries qnz in the lower triangular part of the Hessian G. Specifically nz and qnz are used to preallocate space for the sparse matrices A_{raw} and G.
n and nz are determined by scanning through the `COLUMNS` section, m is determined by scanning through the `ROWS` section and qnz is determined by scanning through the `QUADOBJ` section.

By using `ReadQPSdimensions` to scan the file `qptest.qps` we get the output listed in table 4.3.

Number	Output
n	2
m	2
nz	4
qnz	1

Table 4.3: *Output from scanning example QPS file.*

Comparing the values in table 4.3 to both the QP listed in (4.2) and the QP listed in QPS format we see that the numbers found by `ReadQPSdimensions` are correct.
The output from `ReadQPSdimensions` is used to preallocate space for matrices and vectors in the actual QPS reader as mentioned above.

`QPSreader` scans through a QPS file one section and one line at the time. From each line in each section it extracts relevant information. Sections are seperated by the keywords in table 4.1. Below we explain how information is extracted from each section.

- `ROWS` The constraint names read from `ROWS` are put in a hashtable as keys and their indexes are stored as values. This saves us a substantial amount of computation time because constraint names read from other parts of the QPS files can very quickly be compared to the keys in a hashtable. Additionally the constrainttype ('E','L' or 'G') of each constraint is saved in the vector 'constraintType' and given as an output of the function. The entries in constraintType are used later for identifying constraints in the constraint matrix.

- `COLUMNS` Before scanning through `COLUMNS` we preallocate space for the matrix A_{raw}, the vector g_{vec} and the vector rhs_{raw} using the outputs m, n and nz from `ReadQPSdimensions`. We then scan through `COLUMNS` putting variable names and their index in a new hashtable for later use. Simultaneously with this we fill the matrix A_{raw} containing all constraints of the form $Ax = b$, $Ax \leq b$ or $Ax \geq b$ by getting relevant information from the hashtable containing the constraint names. The entries in g_{vec} are also extracted from this section of the QPS file.

- `RHS` In the `RHS` section we again use the hashtable of constraint names to form the vector rhs_{raw} for A_{raw} by comparing the constraint name from each line in `RHS` with the names saved in the hashtable formed during the scan of `ROWS`.

- `RANGES` The `RANGES` section also uses the hashtable formed in `ROWS` for comparisons and extracts entries for the vector $ranges_{raw}$ and a vector containing the indexes for each range. The indexes will be needed later to identify which range connects to which entry in rhs_{raw}.

- `BOUNDS` From `BOUNDS` we read the entries for the vector l containing the lower bounds on x and the vector u containing the upper bounds on x such that $l \leq x \leq u$. This process is complicated by the possible appearance of ten different ways of indicating various types of lower and upper bounds

on individual variables. The indicators are `LO`, `UP`, `FR`, `FX`, `MI`, `PL`, `BV`, `LI`, `UI` and `SC`. If no indicator is listed for a variable then, by default, the lower bound 0 and the upper bound $+\infty$ is used for that variable. For that reason the vector l is initialized to be filled with zeros and the vector u is initialized to be filled with the entry $+\infty$. If a bound of type `FX` (fixed bound) is encountered the entry in l and the entry in u for that particular variable is set to the same value. Bounds of type `BV`, `LI`, `UI` and `SC` describe the presence of either a binary, integer or semi-cont variable, These are not relevant for this presentation so the implemented QPS reader is not equipped to handle these types of variables. Note that the vectors l and u given as output from the function are always of length n.

- `QUADOBJ` Finally space for the matrix G is preallocated using the outputs n and qnz from `ReadQPSdimensions` and the entries for G is determined by reading through `QUADOBJ`. The hashtable formed during the scan of `COLUMNS` is used for comparisons and to determine the position of each entry in G.

- `ENDATA` The QPS file is ended by the keyword `ENDATA`.

In table 4.4 the outputs from the implemented MATLAB function `QPSreader.m` are listed.

Output	Description
Name	Name of QP problem
A_raw	Matrix with entries from `COLUMNS`
rhs_raw	Vector with entries from `RHS`
constraintType	Constraint types
lo_vec	Lower bounds with entries from `BOUNDS`
up_vec	Upper bounds with entries from `BOUNDS`
ranges_raw	Vector with entries from `RANGES`
idxranges_raw	Vector with indexes as entries from `RANGES`
G	Matrix (Hessian) with entries from `QUADOBJ`
g_vec	Vector with entries from `COLUMNS`
g0	Scalar from `RHS`
bv_li_ui_sc	Indicates presence of binary, integer or semi-cont variables
m	Number of constraints in `COLUMNS`
n	Number of variables
nz	Number of nonzeros in A_{raw}

Table 4.4: *Output from QPS reader.*

Below are listed some of the output obtained from `QPSreader` for our example

file `qptest.qps`.

$$A_{raw} = \begin{bmatrix} 2 & 1 \\ -1 & 2 \end{bmatrix}, \quad rhs_{raw} = \begin{bmatrix} 2 \\ 6 \end{bmatrix} \tag{4.3}$$

Comparing A_{raw} and rhs_{raw} in (4.3) to the listed file `qptest.qps` and the equations (4.2) we see that the QPS reader extracts this part of information correctly.

$$g_{vec} = \begin{bmatrix} 0 \\ -2 \end{bmatrix}, \quad g_0 = 4 \tag{4.4}$$

g and g_0 are also extracted as expected as shown in (4.4).

$$l = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad u = \begin{bmatrix} 20 \\ +\infty \end{bmatrix} \tag{4.5}$$

Note that the vectors l and u might contain entries of $-\infty$ or $+\infty$ as is the case for this example as shown in (4.5). This is because of how the output is constructed, i.e. the vectors l and u in $l \leq x \leq u$ have to be of length n so if there is no upper or lower bound specified for a variable a $+\infty$ or $-\infty$ is placed in l or u.

$$G = \begin{bmatrix} 8 & 2 \\ 2 & 10 \end{bmatrix} \tag{4.6}$$

Finally G is also extracted correctly as shown in (4.6). Note that the full matrix is returned by the QPS reader and not just the lower triangular part.

In this chapter we reviewed the design and implementation of a QPS reader for MATLAB and succesfully tested it on a small example. A single example is not enough to provide a satisfactory amount of confidence in the implementation. Later we use the function to read QPS files from the test set provided in [4] and solve the problems using the developed methods which will further improve the confidence in the developed QPS reader.
The development of the QPS reader was very much an iterative process as different ways of constructing QPS files was discovered. Even with extensive testing it is however not possible to rule out that someone might construct a

QPS file that the implemented QPS reader would fail to succesfully read.
Note that the implemented reader is designed for QPS files only and will not work for other variations of the MPS format. Also note that the reader can not handle binary, integer or semi-cont variables. Furthermore it is advisable to check if the output from `QPSreader.m` needs to be converted in some way to fit into the solver the user wishes to use. The output is constructed such that there is a lot of freedom to form needed inputs.
For the notation used in this presentation the function `convertFormat.m` has been designed to convert the output from the QPS reader to a form suitable for the implemented QP solvers. Note that `convertFormat` is currently incapable of converting output from a non-empty `RANGES`-section into one of the QP forms used in this thesis.

Chapter 5

Multiple Centrality Corrections

In this chapter we modify the predictor-corrector method to obtain faster convergence. The modifications are based on Gondzio [7]. The techniques are described for linear programming but we will of course be using them for quadratic programming.

Both the predictor-corrector (PC) method and the multiple centrality corrections (MCC) method introduced in this chapter compute a Newton direction which requires factorizing a matrix and then using the factorization to solve a system and subsequently perform a number of back substitutions. Because the computational cost of factorizing is typically significantly larger than the cost of solving and back substituting (backsolving) it is profitable to add more backsolves with the purpose of reducing the number of factorizations. One of the differences between the predictor-corrector method and the multiple centrality corrections method is that the PC method uses two backsolves for each iteration and that the MCC method allows recursive iterations such that more than two backsolves are possible for each iteration, [8] p. 1.

By modifying the PC method we also hope to remove some of the drawbacks of this method mentioned in chapter 2.

5.1 MCC method

Faster convergence is achieved by enlarging the step lengths for each iteration and by improving the centrality of the current iterate using multiple corrections. By improving the centrality of the current iterate we increase the chances of taking a long step in the following iteration.
To explain the techniques in the method we use the inequality constrained QP (5.1).

$$\min_{x\in\mathbb{R}^n} \quad f(x) = \frac{1}{2}x^T Gx + g^T x \tag{5.1a}$$
$$\text{s.t.} \quad A^T x \geq b \tag{5.1b}$$

The extensions needed to solve the general QP problem are again fairly trivial to include with reference to chapter 3.

Specifically we want our search direction $(\Delta x, \Delta\lambda, \Delta s)$ to be composed of a predictor part $(\Delta^p x, \Delta^p\lambda, \Delta^p s)$ and a corrector part (modified centering direction) $(\Delta^m x, \Delta^m\lambda, \Delta^m s)$ as shown in (5.2).

$$(\Delta x, \Delta\lambda, \Delta s) = (\Delta^p x, \Delta^p\lambda, \Delta^p s) + (\Delta^m x, \Delta^m\lambda, \Delta^m s) \tag{5.2}$$

The predictor part $(\Delta^p x, \Delta^p\lambda, \Delta^p s)$ is obtained by computing a predictor-corrector direction in the same way as in the predictor-corrector method, i.e. by setting up the optimality conditions and first solving the system (5.3) where $r_d = Gx - A\lambda + g$, $r_p = s - A^T x + b$ and $r_{s\lambda} = S\Lambda e$.

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x^a \\ \Delta\lambda^a \\ \Delta s^a \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ r_{s\lambda} \end{bmatrix} \tag{5.3}$$

We then compute α^a, μ^a and σ as shown in (5.4) and explained in chapter 2.

$$\lambda + \alpha^a \Delta\lambda^a \geq 0 \tag{5.4a}$$
$$s + \alpha^a \Delta s^a \geq 0 \tag{5.4b}$$
$$\mu^a = \frac{(s + \alpha^a \Delta s^a)^T (\lambda + \alpha^a \Delta\lambda^a)}{m} \tag{5.4c}$$
$$\sigma = \left(\frac{\mu^a}{\mu}\right)^3 \tag{5.4d}$$

Finally $(\Delta^p x, \Delta^p \lambda, \Delta^p s)$ is computed from (5.5) providing a predictor direction for the multiple centrality corrections part.

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x^p \\ \Delta\lambda^p \\ \Delta s^p \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ r_{s\Lambda} + \Delta S^a \Delta\Lambda^a e - \sigma\mu e \end{bmatrix} \tag{5.5}$$

In the MCC method we want to increase the step length α^p determined for Δ^p. The new step length is computed as shown in (5.6) where $\tilde{\alpha}$ denotes the new and larger step length that we want to achieve and $\delta_\alpha \in [0, 1]$.

$$\tilde{\alpha} = \min(\alpha^p + \delta_\alpha, 1) \tag{5.6}$$

By increasing the step length we make a hypothetical further move along the predictor direction to the trial point $(\tilde{x}, \tilde{\lambda}, \tilde{s})$ defined in (5.7). We then seek to define a corrector direction that drives the trial point to a better centered target in the neighborhood of the central path such that the inequality constraints are not violated.
Note that by moving to the trial point we will in general cross the the boundary of the feasible region [7] p. 138.

$$(\tilde{x}, \tilde{\lambda}, \tilde{s}) = (x, \lambda, s) + \tilde{\alpha}(\Delta x^p, \Delta\lambda^p, \Delta s^p) \tag{5.7}$$

As mentioned above we want to drive the trial point back to the vicinity of the central path so that we still ensure that the inequality constraints are not violated. To accomplish this we compute the complementarity products for the trial point and define the target $\tilde{v}$ in (5.8).

$$\tilde{v} = \tilde{S}\tilde{\Lambda}e \in \mathbb{R}^n \tag{5.8}$$

We then project the computed complementarity products componentwise on a hypercube $H = [\beta_{min}\tilde{\mu}, \beta_{max}\tilde{\mu}]^n$ where $\tilde{\mu} = \sigma\mu$ and define the target v_t in (5.9).

$$v_t = \pi(\tilde{v}|H) \in \mathbb{R}^n \tag{5.9}$$

σ again denotes the centering parameter we have used previously and μ is the complementarity measure for the current iterate.
The projection is done because it is unlikely that all the complementarity products $\tilde{s}_i\tilde{\lambda}_i$ will align to the value $\sigma\mu$ that defines the central path. Some products will be smaller than $\sigma\mu$ and some will be larger than $\sigma\mu$ but we correct only the outliers instead of trying to correct all the products $\tilde{s}_i\tilde{\lambda}_i$ to $\sigma\mu$ [8] p. 6.
Using v_t in (5.9) as a target means that each element in $\tilde{S}\tilde{\Lambda}e$ larger than $\beta_{max}\tilde{\mu}$ is set equal to $\beta_{max}\tilde{\mu}$ and each element smaller than $\beta_{min}\tilde{\mu}$ is set equal to $\beta_{min}\tilde{\mu}$. By doing so each complementarity pair is bounded from below and above defining a neighbourhood of the central path [8], p. 3
We compute the corrector direction $(\Delta^m x, \Delta^m\lambda, \Delta^m s)$ by solving the system (5.10).

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta^m x \\ \Delta^m\lambda \\ \Delta^m s \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ v_t - \tilde{v} \end{bmatrix} \tag{5.10}$$

The term $v_t - \tilde{v}$ can be badly scaled if there are very large complementarity products present in $\tilde{v}$. To prevent this causing potential numerical instability we replace elements in $v_t - \tilde{v}$ smaller than $-\beta_{max}\tilde{\mu}$ with the value $-\beta_{max}\tilde{\mu}$.
Having computed both the predictor search direction $(\Delta^p x, \Delta^p\lambda, \Delta^p s)$ and the modified centering direction $(\Delta^m x, \Delta^m\lambda, \Delta^m s)$ we can compute the search direction $(\Delta x, \Delta\lambda, \Delta s)$.

$$(\Delta x, \Delta\lambda, \Delta s) = (\Delta^p x, \Delta^p\lambda, \Delta^p s) + (\Delta^m x, \Delta^m\lambda, \Delta^m s) \tag{5.11}$$

We determine a step length α using the same technique as previously.

$$\lambda + \alpha\Delta\lambda \geq 0 \tag{5.12}$$
$$s + \alpha\Delta s \geq 0 \tag{5.13}$$

And update the iterate using a dampening factor η.

$$(x_{k+1}, \lambda_{k+1}, s_{k+1}) = (x_k, \lambda_k, s_k) + \eta\alpha(\Delta x, \Delta\lambda, \Delta s) \tag{5.14}$$

If we want to repeat the correction process and use more than a single correction we simply let $(\Delta x, \Delta\lambda, \Delta s)$ computed from (5.11) become the new predictor direction $(\Delta^p x, \Delta^p\lambda, \Delta^p s)$ and repeat the process of determining a new trial point, defining new targets and determining a new corrector direction $(\Delta^m x, \Delta^m\lambda, \Delta^m s)$.

5.1.1 Stopping criteria for the MCC method

The stopping criteria for the MCC method are equivalent to those used for the PC method, i.e. a maximum number of iterations, lower bounds on the norms of the residual vectors r_d and r_p and a lower bound on the absolute value of the complementarity measure μ.
In addition to this we want to measure how much improvement we gain from using an additional correction. If the improvement is not sufficient we want to stop correcting. In practice we operate with an inner while-loop that at the latest terminates when it reaches a previously defined maximum number of corrections K. We might want to stop correcting before all K corrections are computed if we do not gain enough progress.
To measure the progress we compare the step length α determined for the direction $(\Delta x, \Delta\lambda, \Delta s)$ with the step length α^p determined from the predictor step. If α does not increase by at least a fraction of δ_α we want to stop correcting. In practice the stopping criterium is formulated as shown in (5.15) where γ is a fixed tolerance.

$$\alpha < \alpha_p + \gamma\delta_\alpha \tag{5.15}$$

5.1.2 Number of corrections

There is still the question of how many corrections to use. The use of more corrections does not necessarily translate into savings in compation time so we need to determine how many corrections to allow for each iteration of the MCC method such that we maximize the use of each factorization.
In [7] the maximum number of corrections is based on a comparison between the

cost of factorizing and the cost of backsolving. Note that the approach is based on experience with comparing the predictor-corrector method with the multiple centrality corrections method for linear programming problems. Specifically for the computation of Cholesky factorizations. Furthermore certain disadvantages are stated in [7] p. 149 concerning the use of this method for example the effect of different computer architectures.
It is suggested to compute the ratio $r_{f/s}$.

$$r_{f/s} = \frac{E_f}{E_s} \tag{5.16}$$

E_f and E_s are defined as shown in (5.17) and (5.18).
l_i is the number of off-diagonal nonzero entries in column i of the Cholesky matrix, m is the number of columns in the Cholesky matrix and n is the number of constraints.

$$E_f = \sum_{i=1}^{m} l_i^2 \tag{5.17}$$

$$E_s = 2\sum_{i=1}^{m} l_i + 12n \tag{5.18}$$

The idea of this approach is to allow multiple corrections if the current factorization was expensive to compute compared to the cost of solving the system, such that an expensive factorization is thoroughly used before moving on to the next iterate.
If $r_{f/s} \leq 10$ no centrality corrector is allowed and the method is reduced to the predictor-corrector method. If $r_{f/s} > 10$ one corrector is allowed, two correctors are allowed if $r_{f/s} > 30$ and three correctors are allowed if $r_{f/s} > 50$.

The described approach is easy to relate to the inequality constrained QP's introduced previously because the method used to solve these problems also involve the computation of Cholesky factorizations. We therefore assume that we can use the approach above for simple QP problems.
It is not clear however how to use the approach for a MCC method designed to solve an inequality and equality constrained (general) QP because we instead of computing a Cholesky factorization compute a LDL^T factorization. We could for example just count the number of off-diagonal elements in L from the LDL^T factorization instead of counting the the elements in the Cholesky matrix but a

few tests show that this slows down the implemented method significantly.
Formulating a new approach for determining the maximum number of corrections K is not an easy task so we will restrict our attention to analyzing if computation time is reduced by allowing one or two additional corrections in the MCC method compared to the PC method.
Note that an easy solution to the problem of determing a maximum number of allowable corrections would be to measure the time required to factorize for each iteration and base the maximum number of allowable iterations on the time measured. This approach however requires some experience with how to make intervals of time to match appropriate values of the maximum number of allowable corrections.

5.1.3 Algorithm

The full MCC method for the inequality constrained QP is stated below. Note that the first part is identical to the PC method.
As input the method requires a starting guess (x_0, λ_0, s_0) with $\lambda_0 > 0$ and $s_0 > 0$.

- *Input*
  (x_0, λ_0, s_0) and G, A, g, b
- *Compute residuals and complementarity measure*

$$\begin{aligned} r_d &= Gx_0 + g - A\lambda_0 \\ r_p &= s_0 - A^T x_0 + b \\ r_{s\lambda} &= S_0 \Lambda_0 e \end{aligned}$$

$$\mu = \frac{s_0^T \lambda_0}{m}$$

- *Start while loop (terminate if stop criteria are satisfied)*
- *Affine-scaling step:*

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x^a \\ \Delta \lambda^a \\ \Delta s^a \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ r_{s\lambda} \end{bmatrix}$$

- *Compute* α^a

$$\begin{aligned} \lambda + \alpha^a \Delta \lambda^a \geq 0 \\ s + \alpha^a \Delta s^a \geq 0 \end{aligned}$$

- *Compute* μ^a

$$\mu^a = \frac{(s + \alpha^a \Delta s^a)^T (\lambda + \alpha^a \Delta \lambda^a)}{m}$$

- *Compute centering parameter* σ

$$\sigma = \left(\frac{\mu^a}{\mu} \right)^3$$

- *Corrector and centering step:*

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta x^c \\ \Delta \lambda^c \\ \Delta s^c \end{bmatrix} = - \begin{bmatrix} 0 \\ 0 \\ \Delta S^a \Delta \Lambda^a e - \sigma \mu e \end{bmatrix}$$

- *Predictor direction:*

$$(\Delta x^p, \Delta \lambda^p, \Delta s^p) = (\Delta x^a, \Delta \lambda^a, \Delta s^a) + (\Delta x^c, \Delta \lambda^c, \Delta s^c)$$

- *Compute* α^p

$$\lambda + \alpha^p \Delta \lambda \geq 0$$
$$s + \alpha^p \Delta s \geq 0$$

- *Compute enlarged step length* $\tilde{\alpha}$

$$\tilde{\alpha} = \min(\alpha^p + \delta_\alpha, 1)$$

- *Determine maximum number of corrections* K
- *Multiple correctors:*

- *Start while loop (stop if maximum number of corrections is reached)*
  Compute trial point:

$$(\tilde{x}, \tilde{\lambda}, \tilde{s}) = (x, \lambda, s) + \tilde{\alpha}(\Delta x^p, \Delta \lambda^p, \Delta s^p)$$

  Compute target and project onto hypercube:

$$\tilde{v} = \tilde{S}\tilde{\Lambda} e \in \mathbb{R}^n$$

$$v_t = \pi(\tilde{v}|H) \in \mathbb{R}^n$$

  Prevent values in $v_t - \tilde{v}$ from being too small

Solve to obtain search direction $(\Delta^m x, \Delta^m \lambda, \Delta^m s)$

$$\begin{bmatrix} G & -A & 0 \\ -A^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta^m x \\ \Delta^m \lambda \\ \Delta^m s \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \\ v_t - \tilde{v} \end{bmatrix}$$

Compute search direction:

$$(\Delta x, \Delta \lambda, \Delta s) = (\Delta^p x, \Delta^p \lambda, \Delta^p s) + (\Delta^m x, \Delta^m \lambda, \Delta^m s)$$

- *Compute* α

$$\lambda + \alpha \Delta \lambda \geq 0$$
$$s + \alpha \Delta s \geq 0$$

- *Test for improvement:*
  If $\quad (\alpha \geq \alpha_p + \gamma \delta_\alpha)$

$$(\Delta^p x, \Delta^p \lambda, \Delta^p s) = (\Delta x, \Delta \lambda, \Delta s)$$

  else

$$(\Delta x, \Delta \lambda, \Delta s) = (\Delta^p x, \Delta^p \lambda, \Delta^p s)$$

  end ifelse

- *End while loop*

- *Update* (x, λ, s)

- *Update residuals and complementarity measure*

$$r_d = Gx + g - A\lambda$$
$$r_p = s - A^T x + b$$
$$r_{s\lambda} = S\Lambda e$$

$$\mu = \frac{s^T \lambda}{m}$$

- *End while loop*

To extend the method to handle general QP's we only have to replace the systems we solve in the predictor step and in the corrections part along with adding a few extra inputs and defining different residual vectors as stated in chapter 3.

5.2 Implementation and Test

In this section we explain some of the details involved in implementing the MCC method in MATLAB and test if the implementation works as intended. We will test two seperate implementations of the method. An implementation designed to solve inequality constrained QP's and an implementation designed to solve general QP's.
For both versions of the method we will however be using the same values of δ_α, η, β_{min}, β_{max} and γ. The values used in the implementations are listed in table 5.1 and are identical to those suggested in [7].

η	0.99995
δ_α	0.1
β_{min}	0.1
β_{max}	10
γ	0.1

Table 5.1: *Fixed values for MCC method.*

The value of the tolerance ϵ will vary depending on which problem we want to solve. For problems to which we know the solution with machine precision we set $\epsilon = 1e-16$. For problems from the test set [4] we set $\epsilon = 1e-10$ because we know the solution to these problems with an accuracy of ten digits.

Additionally we will compare the performance of the MCC method with the performance of the PC method by applying the two methods to the same test problems and subsequently by comparing the number of iterations and compation time used by each method.

5.2.1 Inequality constrained QP's

First we test the implementation of the method that solves QP's of the form (5.19).

$$\min_{x\in\mathbb{R}^n} \quad f(x) = \frac{1}{2}x^T Gx + g^T x \tag{5.19a}$$

$$\text{s.t.} \quad A^T x \geq b \tag{5.19b}$$

To solve the systems in the predictor and corrector parts we again use a Cholesky factorization followed by backward substitution as we did for the PC method.

The implemented MATLAB function `mccQP_simplev1.m` can be seen in appendix B.2.
To validate that the implementation works as intended we reuse the test problem (5.20) where G is an identity matrix of size $n \times n$, g is a randomly generated vector of size $n \times 1$ and l_{vec} and u_{vec} are lower and upper bounds on x respectively such that $l_{vec} \leq x \leq u_{vec}$ where l_{vec} is a vector of size $n \times 1$ containing the scalar l n times and u_{vec} is a vector of size $n \times 1$ containing the scalar u n times.

$$\min_{x \in \mathbb{R}^n} \quad f(x) = \frac{1}{2} x^T G x + g^T x \tag{5.20a}$$

$$\text{s.t.} \quad l_{vec} \leq x \leq u_{vec} \tag{5.20b}$$

We use the inputs and fixed values stated in table 5.2 which are the same as those we used to test the PC method.

n	10
x_0	$[0, 0, \ldots, 0]^T$
λ_0	$[1, 1, \ldots, 1]^T$
s_0	$[1, 1, \ldots, 1]^T$
ϵ	1e-16
M	100
seed	100

Table 5.2: *Inputs for test of implemented MCC method for QP with inequality-constraints.*

Using the inputs and fixed values above we get the result shown in table 5.3 where the vector $-g$ is the exact solution of the problem.

From table 5.3 we see that the solution determined by the implemented method is the same as the exact solution $-g$. It would therefore seem that the implementation works as intended at least for this particular problem.
The MATLAB file `test2.m` that was used to set up the problem generate the results in table 5.3 can be seen in appendix B.5.
Note that we have allowed only one correction for the method in this case because the factorizations are too simple for us to gain anything from using more than one additional correction.

To compare the MCC method with the PC method we let $n \to \infty$ in the simple

i'th element	$x(i)$	$-g(i)$	$x(i) - (-(g(i))$
1	-0.9085	-0.9085	0
2	2.2207	2.2207	0
3	0.2391	0.2391	0
4	-0.0687	-0.0687	0
5	2.0202	2.0202	0
6	0.3641	0.3641	0
7	0.0813	0.0813	0
8	1.9797	1.9797	0
9	-0.7882	-0.7882	0
10	-0.7366	-0.7366	0

Table 5.3: *Result of test run for QP with inequality-constraints.*

test problem and solve each problem generated in this way with both methods again allowing only one additional correction for the MCC method. We use the same inputs as those stated in table 5.2 with the exception of n which we gradually increase.
In figure 5.1 we have plotted the number of iterations as a function of n for both methods.

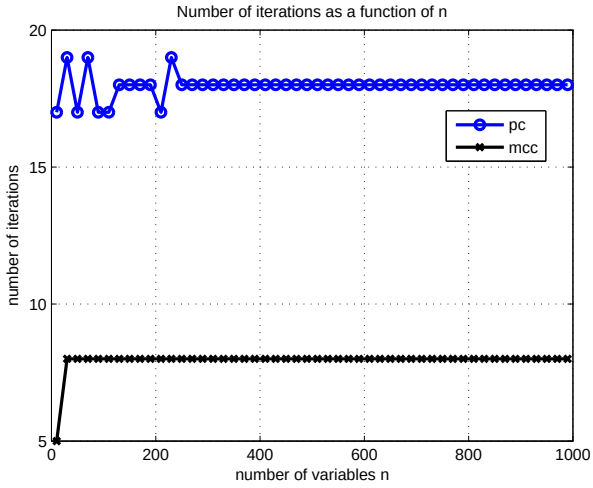

Figure 5.1: *Number of iterations as a function of n.*

From figure 5.1 we see that we obtain a significant reduction in the number of iterations by using the MCC method compared to the PC method. Most problems are solved using around 18 iterations for the PC method while the

MCC method use around 8 resulting in a reduction of around 55%.
To see if these savings in the number of iterations also translates into savings in computation time we have plotted computation time as a function of n in figure 5.2.

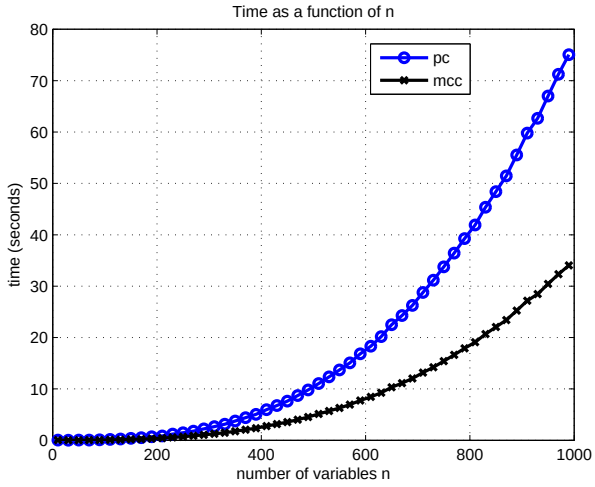

Figure 5.2: *Computation time as a function of n.*

As n steadily grows we see from figure 5.2 that we save more and more time by using the MCC method instead of the PC method. For $n = 1000$ we save more than 50% computation time by using the MCC method.
The savings in computation time are visualized in figure 5.3.

For inequality constrained QP's it would seem beneficial to use the MCC method based on the results above.
The MATLAB scripts `plotnvsiter.m` and `plotnvstime.m` used to generate the figures can be seen in appendix B.5.

5.2.2 General case

Next we want to test if the implemented MCC method for solving the general QP problem (5.21) works as intended and how it performs on some selected test problems compared to the PC method.

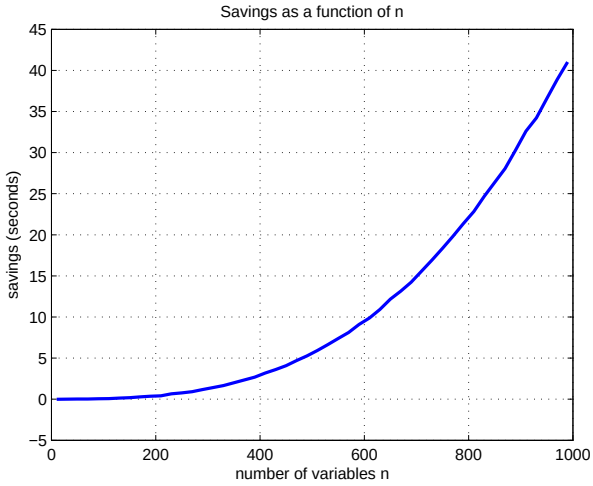

Figure 5.3: *Computation time savings.*

$$\min_{x\in\mathbb{R}^n} \quad \frac{1}{2}x^TGx + g^Tx \tag{5.21a}$$

$$\text{s.t.} \quad A^Tx = b \tag{5.21b}$$

$$C^Tx \geq d \tag{5.21c}$$

To test the method we use a selection of 38 test problems from the test set [4]. This also allows us to test the QPS reader further.
As starting values we use $x_0 = 0.5 \times (1, 1, \ldots, 1)^T$, $\lambda_0 = (1, 1, \ldots, 1)^T$, $y_0 = (1, 1, \ldots, 1)^T$ and $s_0 = (1, 1, \ldots, 1)^T$.
In table 5.4 results are listed from using the implemented MCC methods `mccQP_gen.m` and `mccQP_simplev1.m` to solve the 38 test problems. In the table and in the following tables 'Its'=number of iterations, 'Time'=computation time in seconds, 'nc'=not converged within 200 iterations, 'Solution'=solution provided for the QP problem in [4] and K=maximum number of corrections allowed. The dimensions of each of the test problems can be seen in appendix A.

From the table we see that the implemented method converges within the maximum number of iterations for all the test problems with the exception of `cvxqp3_m`, `qscorpio`, `qscrs8` and `qshare2b`. Why the implemented method can not solve these particular problems is not clear. It could for example be because of some special structure or property of the problems or that our implemented method somehow gets stuck and fails to make sufficient progress towards

QP problem	Its	Time	Solution	MCC method	K
aug3dcqp	6	3.41	9.93362148e+02	9.93373563e+02	2
aug3dqp	6	3.68	6.75237673e+02	6.75264456e+02	2
cont-050	19	21.97	-4.56385090e+00	-4.49873507e+00	2
cvxqp1_m	47	35.02	0.108751157e+07	0.108751156+06	2
cvxqp1_s	10	0.15	0.115907181e+05	0.115907181e+05	2
cvxqp2_m	36	16.25	0.820155431e+06	0.820155431+06	2
cvxqp2_s	9	0.13	0.812094048e+04	0.812094048e+03	2
cvxqp3_m	nc	nc	0.136282874e+07	nc	2
cvxqp3_s	13	0.20	0.119434322e+05	0.119434340e+04	2
dual1	8	0.10	3.50129662e-02	3.50129808e-02	2
dual2	6	0.18	3.37336761e-02	3.37336763e-02	2
dual3	7	0.19	1.35755839e-01	1.35755842e-01	2
dual4	5	0.15	7.46090842e-01	7.46091187e-01	2
gouldqp2	7	0.56	1.84275341e-04	1.84657702e-04	2
gouldqp3	5	0.64	2.06278397e+00	2.06307997e+00	1
hs21	51	0.06	-9.99599999e+01	-9.97357626e+01	2
hs35	6	0.04	1.11111111e-01	1.11111111e-01	2
hs53	43	0.09	4.09302326e+00	4.09302325e+00	2
hs76	7	0.05	-4.68181818e+00	-4.68181818e+00	2
lotschd	9	0.07	0.239841589+04	0.23985889e+04	2
mosarqp1	12	46.60	-0.952875441e+03	-0.952875443e+03	2
mosarqp2	11	3.68	-0.159748211e+04	-0.159748212e+04	2
qpcblend	21	0.12	-7.84254092e-03	-7.66678075e-03	2
qptest	15	0.04	0.437187500e+01	0.437187500e+01	2
qscorpio	nc	nc	1.88050955e+03	nc	2
qscrs8	nc	nc	9.04560014e+02	nc	2
qscsd1	15	0.77	8.66666668e+00	8.66669358e+00	2
qscsd6	15	1.47	5.08082139e+01	5.08476299e+01	1
qscsd8	9	2.91	9.40763574e+02	9.40808758e+02	2
qsctap1	16	0.72	1.41586111e+03	1.41586111e+03	2
qsctap2	12	3.35	1.73502650e+03	1.73502651e+03	2
qsctap3	12	5.43	1.43875468e+03	1.43875473e+03	2
qshare2b	nc	nc	1.17036917e+04	nc	2
stcqp1	8	47.74	1.55143555e+05	1.55143555e+05	2
stcqp2	11	27.38	2.23273133e+04	2.23273135e+04	2
tame	4	0.04	3.47098798e-30	0.00000000e+00	2
values	17	0.32	-0.139662114e+01	-0.139662114e+01	2
zecevic2	10	0.09	-4.12500000e+00	-4.12500000e+00	2

Table 5.4: *Results from testing implemented MCC method.*

the solution.
For the remaining problems for which the implemented MCC methods does converge, we see that the solutions are in some cases not nearly as accurate as the solutions taken from [4]. This is for example the case for the problems `aug3dcqp`, `aug3dqp`, `cont-050`, `gouldqp2`, `gouldqp3`, `hs21` and `qpcblend`. Furthermore a large amount of iterations is used to converge in some cases. For example the method uses more than 35 iterations to solve the problems `cvxqp1_m`, `cvxqp2_m` and `hs53`. This is not what we would expect having designed the method to use as few iterations as possible to reduce the number of factorizations. Based on these observations it would seem that there are some issues with the corrections part of the method that are not quite working as intended.
In table 5.5 we have solved the same 38 problems with the implemented PC methods `pcQP_gen.m` and `pcQP_simplev1.m` so that we can compare the performances of the PC and MCC method.

First thing to note compared to the results from table 5.5 is that the implemented PC method generally appears to produce more accurate results for example for the problems `aug3dcqp`, `aug3dqp` and `gouldqp2`. The number of iterations also seem a bit more stable in the sense that they do not exceed 23 except for the three problems `qscorpio`, `qscrs8` and `qshare2b`.
To get a better visualization of the comparison between the number of iterations and amount of computation time used by both methods these results are listed together in table 5.6.

From table 5.6 we see that the MCC method uses fewer iterations than the PC method for 23 out of the 38 problems. Fewer iterations does not however translate into savings in computation time in all cases. This is typically the case for problems with few variables and few constraints and in these cases we also see from the table that the difference between the computation time used by the MCC method and the computation time used by the PC method is small. For example the MCC method uses 4 iterations less to solve the problem `dual2` but it is 0.08 seconds slower than the PC method. For problems with more variables and constraints the fewer number of iterations does translate into computation time savings. This is for example the case for the problem `stcqp1` where the MCC method used only 8 iterations compared to the 13 used by the PC method and only 47.74 seconds compared to the 72.25 seconds spent by the PC method. These observations suggest that the MCC method is more efficient for larger problems compared to the PC method.

To provide a visual example of the difference between the PC and MCC methods we will use the problem `qptest` as an example. `qptest` is one of the few test problems with only two variables and a small amount of constraints making it suitable for visual analysis. We previously used this test problem to explain parts of a QPS file in chapter 4. The problem is shown in (5.22).

QP problem	Its	Time	Solution	PC method
aug3dcqp	11	5.84	9.93362148e+02	9.93362147e+02
aug3dqp	11	6.43	6.75237673e+02	6.75237678e+02
cont-050	11	12.23	-4.56385090e+00	-4.56382677e+00
cvxqp1_m	22	21.20	0.108751157e+07	0.108751157e+06
cvxqp1_s	16	0.13	0.115907181e+05	0.115907181e+05
cvxqp2_m	23	13.00	0.820155431e+06	0.820155431e+06
cvxqp2_s	17	0.15	0.812094048e+04	0.812094048+06
cvxqp3_m	19	27.79	0.136282874e+07	0.13628274e+07
cvxqp3_s	13	0.12	0.119434322e+05	0.119434322e+05
dual1	11	0.17	3.50129662e-02	3.50129702e-02
dual2	10	0.10	3.37336761e-02	3.37336764e-02
dual3	10	0.11	1.35755839e-01	1.35755842e-01
dual4	9	0.09	7.46090842e-01	7.46090845e-01
gouldqp2	12	0.93	1.84275341e-04	1.84296498e-04
gouldqp3	11	0.88	2.06278397e+00	2.06278397e+00
hs21	21	0.14	-9.99599999e+01	-9.99599999e+01
hs35	13	0.05	1.11111111e-01	1.11111111e-01
hs53	9	0.04	4.09302326e+00	4.09302326e+00
hs76	14	0.02	-4.68181818e+00	-4.68181818e+00
lotschd	20	0.05	0.239841589+04	0.23984164e+04
mosarqp1	11	40.01	-0.952875441e+03	-0.952875443e+03
mosarqp2	11	3.22	-0.159748211e+04	-0.159748212e+04
qpcblend	18	0.20	-7.84254092e-03	-7.75881879e-03
qptest	18	0.01	0.437187500e+01	0.437187500e+01
qscorpio	53	1.38	1.88050955e+03	1.88050955e+03
qscrs8	91	8.42	9.04560014e+02	9.04569185e+02
qscsd1	12	0.43	8.66666668e+00	8.66668734e+00
qscsd6	16	1.47	5.08082139e+01	5.08082324e+01
qscsd8	17	5.27	9.40763574e+02	9.40763574e+02
qsctap1	22	0.69	1.41586111e+03	1.41586111e+03
qsctap2	19	5.21	1.73502650e+03	1.73502650e+03
qsctap3	19	8.18	1.43875468e+03	1.43875468e+03
qshare2b	36	0.08	1.17036917e+04	1.17037872e+04
stcqp1	13	72.25	1.55143555e+05	1.55143555e+05
stcqp2	12	29.17	2.23273133e+04	2.23273133e+04
tame	13	0.03	3.47098798e-30	0.00000000e+00
values	21	0.28	-0.139662114e+01	-0.139662114e+01
zecevic2	17	0.03	-4.12500000e+00	-4.12500000e+00

Table 5.5: *Results from testing implemented PC method.*

QP problem	Its PC	Its MCC	Time PC	Time MCC
aug3dcqp	11	6	5.84	3.41
aug3dqp	11	6	6.43	3.68
cont-050	11	19	12.23	21.97
cvxqp1_m	22	47	21.20	35.02
cvxqp1_s	16	10	0.13	0.15
cvxqp2_m	23	36	13.00	16.25
cvxqp2_s	17	9	0.15	0.13
cvxqp3_m	19	nc	27.79	nc
cvxqp3_s	13	13	0.12	0.20
dual1	11	8	0.17	0.10
dual2	10	6	0.10	0.18
dual3	10	7	0.11	0.19
dual4	9	5	0.09	0.15
gouldqp2	12	7	0.93	0.56
gouldqp3	11	5	0.88	0.64
hs21	21	51	0.14	0.06
hs35	6	13	0.05	0.04
hs53	9	43	0.04	0.09
hs76	14	7	0.02	0.05
lotschd	20	9	0.05	0.07
mosarqp1	11	12	40.01	46.60
mosarqp2	11	11	3.22	3.68
qpcblend	18	21	0.20	0.12
qptest	18	15	0.01	0.04
qscorpio	53	nc	1.38	nc
qscrs8	91	nc	8.42	nc
qscsd1	12	15	0.43	0.77
qscsd6	16	15	1.47	1.47
qscsd8	17	9	5.27	2.91
qsctap1	22	16	0.69	0.72
qsctap2	19	12	5.21	3.35
qsctap3	19	12	8.18	5.43
qshare2b	36	nc	0.08	nc
stcqp1	13	8	72.25	47.74
stcqp2	12	11	29.17	27.38
tame	13	4	0.03	0.04
values	21	17	0.28	0.32
zecevic2	17	10	0.03	0.09

Table 5.6: *Comparison of the PC and MCC method.*

$$\begin{aligned}
\min \quad & f(x,y) = 4 + 1.5x - 2y + 0.5(8x^2 + 2xy + 2yx + 10y^2) && \text{(5.22a)}\\
s.t. \quad & 2x + y \geq 2 && \text{(5.22b)}\\
& -x + 2y \leq 6 && \text{(5.22c)}\\
& 0 \leq x \leq 20, \quad y \geq 0 && \text{(5.22d)}
\end{aligned}$$

In figure 5.4 we have plotted the sequence of iterations for both the PC method and the MCC method for the problem `qptest` to provide a visualization of how the two methods converge towards the solution.
Both methods start in the point $x = (0.5, 0.5)$ and stop when they reach the solution $x \approx (0.76, 0.48)$. The PC method uses 18 iterations to reach the solution and the MCC method uses 15 iterations to reach the solution.
The grey area in the plot represents part of the infeasible region and the white area represents part of the feasible region. The line 'c1' dividing the two regions is the line $y = 2 - 2x$ stemming from the constraint $2x + y \geq 2$ which is active at the solution.
Iterations are marked with crosses $\times$. It is impossible to spot all the iterations because many of them are bunched very close to each other near the solution.

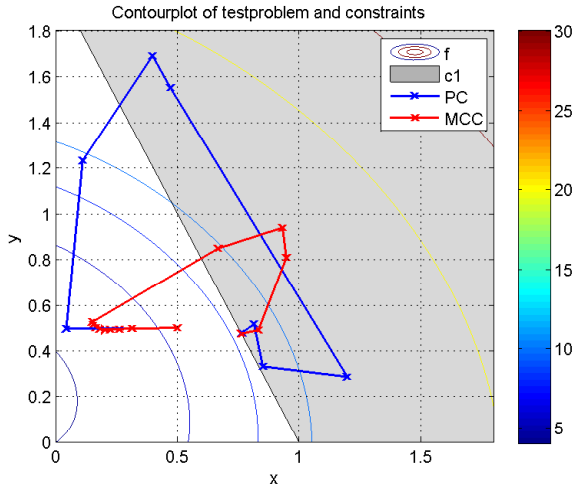

Figure 5.4: *Visual comparison of PC and MCC method.*

Figure 5.4 shows that the PC and MCC method follow very different paths towards the solution. The PC method consistently takes fairly long steps towards the solution and then takes a lot of shorter steps near the solution. The MCC method seems more flexible in the sense that it mixes long and short steps early

in the sequence of iterations. This could be interpreted as improving the centrality of the current iterate and then being able to take a longer steps in the following iterations.
The scrip files `demo_v1.m` and `demo_v2.m` used to generate the results in this section can be seen in appendix B.5.

The above testing have provided some belief that the MCC method has the potential to be more efficient than the PC method particularly for problems where the number of variables and constraints is large. It is also clear however that there is much room for improving the method such that it performs on a more consistent level i.e. converging to the optimal solution in all cases, determining solutions with better numerical precision and not using a sporadical large amount of iterations.
A number of things come to mind when thinking about how to possibly improve the method. First of all the same starting point is used for all problems which is definitely not optimal. However lack of knowledge about the problems hinder us in making better guesses for the starting points. Then there is the question of how many corrections to allow for each iteration. In the above experiments we fixed the maximum number of corrections K allowed to either 1 or 2 because we do not yet have the experience to for example base the choice of K on the time required to factorize. Other things that might improve the performance of the implemented method include experimenting with the fixed values in table 5.1 and perhaps testing an adaptive choice of the step reduction factor η.
In the next chapters we explore methods to modify the developed methods further with the purpose of improving their numerical performance.

Chapter 6

Further modifications

In this chapter we explore strategies to improve our previously designed interior-point methods. First we present a new step length strategy that could potentially improve both the PC and the MCC method. Then we present a modification aimed at improving the MCC method. Finally we make suggestions to other modifications that might improve the performance of the interior-point methods.

6.1 Step length strategy

Numerical experience has shown that faster covergence can be obtained by using different step lengths for each of the search directions, [1] p. 483. In this section we review a step length strategy proposed by Curtis and Nocedal [6].
In [6] the new step length strategy is developed based on a QP with the form (6.1) where $A \in \mathbb{R}^{m \times n}$ and m is the number of equality constraints.

$$\min_{x\in\mathbb{R}^n} \quad f(x) = \frac{1}{2}x^T Gx + g^T x \tag{6.1a}$$
$$\text{s.t.} \quad Ax = b \tag{6.1b}$$
$$x \geq 0 \tag{6.1c}$$

For convenience we will also use this form. The Lagrangian function and the optimality conditions for (6.1) are listed in (6.2) and (6.3) respectively.

$$L(x, y, z) = \frac{1}{2}x^T Gx + g^T x - y(Ax - b) - zx \tag{6.2}$$

$$A^T y + z - Gx - g = 0 \tag{6.3a}$$
$$Ax = b \tag{6.3b}$$
$$XZe = 0 \tag{6.3c}$$
$$(x, z) \geq 0 \tag{6.3d}$$

Setting up a PC method and a MCC method for a QP of the form (6.1) can be done equivalently to the previous chapters with very few changes needed. A MATLAB implementation of the PC method `pcQP_simplev2.m` designed to solve QP's of the form (6.1) has been developed and can be viewed in appendix B.1.

To set up the new step length strategy we make use of the previously applied strategy (6.4) with a single scaling parameter $\alpha \in [0, 1]$.

$$(x_{k+1}, y_{k+1}, z_{k+1}) = (x_k, y_k, z_k) + \alpha(\Delta x, \Delta y, \Delta z) \tag{6.4}$$

With the old step length strategy we selected α as shown in (6.5).

$$\alpha_x = \min_{i:\Delta x_i<0} \left(1, \min \frac{-x_i}{\Delta x_i}\right) \tag{6.5a}$$
$$\alpha_z = \min_{i:\Delta z_i<0} \left(1, \min \frac{-z_i}{\Delta z_i}\right) \tag{6.5b}$$
$$\alpha = \min\{\alpha_x, \alpha_z\} \tag{6.5c}$$

With the new strategy we use a slightly different approach such that we instead compute the step lengths $\bar{\alpha}_x$, $\bar{\alpha}_z$ and $\bar{\alpha}$ as shown in (6.6) where $0 < \beta < 1$.

$$\bar{\alpha}_x = \beta \min_{i:\Delta x_i<0} \left(1, \min \frac{-x_i}{\Delta x_i}\right) \tag{6.6a}$$

$$\bar{\alpha}_z = \beta \min_{i:\Delta z_i<0} \left(1, \min \frac{-z_i}{\Delta z_i}\right) \tag{6.6b}$$

$$\bar{\alpha} = \min\{\bar{\alpha}_x, \bar{\alpha}_z\} \tag{6.6c}$$

The strength of using the old strategy is a guaranteed reduction of primal and dual infeasibility, [6] p. 4. Furthermore the use of a dampening parameter η (or in this case β) results in a reduction in complementarity. It is however possible to reduce the primal and dual infeasibility as well as the complementarity by using multiple step lengths. To accomplish this it is suggested to select step lengths $(\alpha_x, \alpha_y, \alpha_z)$ such that the merit function $\phi(x, y, z)$ in (6.7) is reduced.

$$\phi(x,y,z) = \|b-Ax\|^2+\|-A^Ty-z+Gx+g\|^2+x^Tz = \|r_p\|^2+\|r_d\|^2+x^Tz \tag{6.7}$$

ϕ is chosen such that the norms of the residual vectors are reduced (primal and dual infeasibility) along with the complementarity products x^Tz. All of these terms should be small near the optimal solution, [6] p. 3.
Specifically step lengths α_x, α_y and α_z are computed as the unique solution to (6.8).

$$\min_{\alpha_x,\alpha_y,\alpha_z\in\mathbb{R}} \quad \phi(x+\alpha_x\Delta x, y+\alpha_y\Delta y, z+\alpha_z\Delta z) \tag{6.8a}$$

$$\text{s.t.} \quad (\alpha_x, \alpha_z) \in A_1 \cup A_2 \tag{6.8b}$$

The two sets A_1 and A_2 are defined as shown in (6.9) and (6.10).

$$A_1 = \{(\alpha, \bar{\alpha}) \mid 0 \leq \alpha \leq \bar{\alpha} \tag{6.9}$$

$$A_2 = \begin{cases} \{(\bar{\alpha}, \alpha_z) \mid \bar{\alpha} \leq \alpha_z \leq \bar{\alpha}_z\} & \text{if} \quad \bar{\alpha}_x \leq \bar{\alpha}_z \\ \{(\alpha_x, \bar{\alpha}) \mid \bar{\alpha} \leq \alpha_x \leq \bar{\alpha}_x\} & \text{if} \quad \bar{\alpha}_x \geq \bar{\alpha}_z \end{cases} \tag{6.10}$$

The purpose of using these sets is that we do not have to perform a computational expensive minimization of ϕ over the entire feasible set, [6] p. 5. By using the set $(\alpha_x, \alpha_z) \in A_1 \cup A_2$ a guaranteed reduction in ϕ can be achieved by solving two subproblems that are computationally cheap. Furthermore we set α_y to be an unrestricted step length parameter providing more freedom of movement.
By writing out $\phi(x + \alpha_x\Delta x, y + \alpha_y\Delta y, z + \alpha_z\Delta z)$ we obtain the following.

$$\begin{aligned}
&\phi(x + \alpha_x\Delta x, y + \alpha_y\Delta y, z + \alpha_z\Delta z) = \\
&\quad \|b - A(x + \alpha_x\Delta x)\|^2 + \| - A^T(y + \alpha_y\Delta y) - (z + \alpha_z\Delta z) + G(x + \alpha_x\Delta x) + g\|^2 + \\
&\quad (x + \alpha_x\Delta x)^T(z + \alpha_z\Delta z) = \\
&\|r_p - A\alpha_x\Delta x\|^2 + \|r_d + G\alpha_x\Delta x - A^T\alpha_y\Delta y - \alpha_z\Delta z\|^2 + \\
&\quad (x + \alpha_x\Delta x)^T(z + \alpha_z\Delta z)
\end{aligned}$$

We now define the vectors listed in (6.11).

$$r = \begin{bmatrix} r_p \\ r_d \end{bmatrix}, \quad s = \begin{bmatrix} -A \\ G \end{bmatrix}\Delta x, \quad t = \begin{bmatrix} 0 \\ -A^T \end{bmatrix}\Delta y, \quad u = \begin{bmatrix} 0 \\ -I \end{bmatrix}\Delta s \tag{6.11}$$

Using these definitions enables us to reduce the expression for $\phi(x + \alpha_x\Delta x, y + \alpha_y\Delta y, z + \alpha_z\Delta z)$ as shown in (6.12).

$$\phi(x+\alpha_x\Delta x, y+\alpha_y\Delta y, z+\alpha_z\Delta z) = \|r+\alpha_x s+\alpha_y t+\alpha_z u\|^2+(x+\alpha_x\Delta x)^T(z+\alpha_z\Delta z) \tag{6.12}$$

The global optimum of (6.12) can be computed by solving two 2-dimensional QP's with simple constraints, [6] p. 6. Solving the two 2-dimensional QP's we obtain 2 trial points and select the trial point that provides the smallest value of ϕ.
The first trial point is defined as the minimizer of ϕ over the set A_1 and is obtained by solving the QP problem (6.13) with $\alpha_z = \alpha_x$.

$$\min_{\alpha_x, \alpha_y \in \mathbb{R}} \quad q_1(\alpha_x, \alpha_y) = \frac{1}{2}[\alpha_x, \alpha_y]G_1[\alpha_x, \alpha_y]^T + g_1^T[\alpha_x, \alpha_y]^T \tag{6.13a}$$

$$\text{s.t.} \quad 0 \leq \alpha_x \leq \bar{\alpha} \tag{6.13b}$$

The matrix G_1 and the vector g_1 are listed in (6.14).

$$G_1 = \begin{bmatrix} (s+u)^T(s+u) + \Delta x^T \Delta z & (s+u)^T t \\ (s+u)^T t & t^T t \end{bmatrix} \tag{6.14a}$$

$$g_1 = \begin{bmatrix} r^T(s+u) + \frac{1}{2}(\Delta x^T z + x^T \Delta z) \\ r^T t \end{bmatrix} \tag{6.14b}$$

The second trial point is defined as the minimizer over the set A_2. Which 2-dimensional problem to solve depends on the values of $\bar{\alpha}_x$ and $\bar{\alpha}_z$.
If $\bar{\alpha}_x \leq \bar{\alpha}_z$ we solve the QP problem (6.15) with $\alpha_x = \bar{\alpha}_x$.

$$\min_{\alpha_y, \alpha_z \in \mathbb{R}} \quad q_2(\alpha_y, \alpha_z) = \frac{1}{2}[\alpha_y, \alpha_z] G_2 [\alpha_y, \alpha_z]^T + g_2^T [\alpha_y, \alpha_z]^T \tag{6.15a}$$

$$\text{s.t.} \quad \bar{\alpha} \leq \alpha_z \leq \bar{\alpha}_z \tag{6.15b}$$

The matrix G_2 and the vector g_2 are listed in (6.16).

$$G_2 = \begin{bmatrix} t^T t & t^T u \\ t^T u & u^T u \end{bmatrix} \tag{6.16a}$$

$$g_2 = \begin{bmatrix} r^T t + \bar{\alpha}_x s^T t \\ r^T u + \frac{1}{2} x^T \Delta z + \bar{\alpha}_x (s^T u + \frac{1}{2} \Delta x^T \Delta z) \end{bmatrix} \tag{6.16b}$$

If $\bar{\alpha}_x > \bar{\alpha}_z$ we solve the QP problem (6.17) with $\alpha_z = \bar{\alpha}_z$.

$$\min_{\alpha_x, \alpha_y \in \mathbb{R}} \quad q_3(\alpha_x, \alpha_y) = \frac{1}{2}[\alpha_x, \alpha_y] G_3 [\alpha_x, \alpha_y]^T + g_3^T [\alpha_x, \alpha_y]^T \tag{6.17a}$$

$$\text{s.t.} \quad \bar{\alpha} \leq \alpha_x \leq \bar{\alpha}_x \tag{6.17b}$$

The matrix G_3 and the vector g_3 are listed in (6.18).

$$G_3 = \begin{bmatrix} s^T s & s^T t \\ s^T t & t^T t \end{bmatrix} \tag{6.18a}$$

$$g_3 = \begin{bmatrix} r^T s + \frac{1}{2} \Delta x^T z + \bar{\alpha}_z (s^T u + \frac{1}{2} \Delta x^T \Delta z) \\ r^T t + \bar{\alpha}_z t^T u \end{bmatrix} \tag{6.18b}$$

The new step length strategy was implemented as a seperate MATLAB function that returns the step lengths α_x, α_y and α_z. `newStepLengths.m` can be seen in appendix B.3.
Before we present some results that compare the new step length strategy to the old step length strategy we will comment on some details in the implementation. In addition to using a dampening factor earlier than previously note that the value of the centering parameter σ can exceed 1 in some cases with the step length changes. The consequence of this is not clear, but the implemented interior-point method has been able to converge to the optimal solution even in cases where $\sigma > 1$. In the implementation we use $\beta = 0.95$.
To solve the 2-dimensional QP's we use the MATLAB function `quadprog` by defining lower and upper bounds on the variables present in the given 2-dimensional QP. In the event of a lower bound being equal to an upper bound (not handable by `quadprog`) we subtract a very small number from the lower bound to ensure that the QP is solved.

We test the implementation of the new step length strategy by using it in an implementation of the PC method. This enables us to compare the performance of this method with the performance of the same method using the old step length strategy. Specifically we are once again interested in using fewer iterations because this means computing fewer expensive matrix factorizations. We will also check if any savings in iterations translate into savings in computation time.
To also solve QP's with constraints of the form $A^T x \geq b$ or QP's with both inequality and equality constraints using the new step length strategy, we would have to reformulate the 2-dimensional QP's above because of the difference in formulation of optimality conditions for the different forms.
From the set of 38 test problems used in the previous chapter 4 are of the form (6.1). While 4 test problems is not a lot it will nevertheless provide us with an idea of how an interior-point method will perform using the new step length strategy.
In table 6.1 the results from using the PC method `pcQP_simplev2` with the old step length strategy are shown.

QP problem	Its	Time	Solution	`pcQP_simplev2.m`
`lotschd`	21	0.04	0.239841589+04	0.239841589+04
`qscsd1`	23	2.89	8.66666668e+00	8.66666667e+00
`qscsd6`	33	10.03	5.08082139e+01	5.08082139e+01
`qscsd8`	33	37.00	9.40763574e+02	9.40763574e+02

Table 6.1: *Results from testing PC method with old step length strategy.*

From table 6.1 we see that the implemented PC method using the old step length strategy determines solutions for the 4 problems with satisfactory precision. In

table 6.2 results from using the PC method with the new step length strategy are shown.

QP problem	Its	Time	Solution	`pcQP_simplev2new.m`
`lotschd`	18	1.36	0.239841589+04	0.239841589+04
`qscsd1`	19	4.46	8.66666668e+00	8.66666667e+00
`qscsd6`	29	15.57	5.08082139e+01	5.08082139e+01
`qscsd8`	29	53.96	9.40763574e+02	9.40763574e+02

Table 6.2: *Results from testing PC method with new step length strategy.*

In table 6.3 the two step length strategies are compared.

QP problem	Its PC old	Its PC new	Time PC old	Time PC new
`lotschd`	21	18	0.04	1.36
`qscsd1`	23	19	2.89	4.46
`qscsd6`	33	29	10.03	15.57
`qscsd8`	33	29	37.00	53.96

Table 6.3: *Comparison of step length strategies.*

From table 6.3 we see that we save 3-4 iterations for each of the 4 test problems by using the new step length strategy. In all 4 cases however we use significantly more computation time in reaching the solution compared to the old step length strategy. The reason for this is the extra computation cost needed in the new step length strategy to set up and solve the two 2-dimensional QP problems. This extra cost exceeds the savings we gain from using fewer iterations. In [6] it is argued that the extra work required to solve the 2 dimensional QP's is small and neglible when comparing the two step lengths strategies. No computation time results are provided however, only the number of iterations.
To make the new step length strategy efficient and usable in practice it is necessary to shave of all unnecessary computations in `newStepLengths.m`. For example inner products that are computed more than once and perhaps solving the 2-dimensional QP's in a more efficient way.
The results above show that the new step length strategy has the potential of being more efficient than the old step length strategy provided that it is possible to make the step length computation more efficient.

6.2 MMC method revisited

The following modifications to the MCC method are based on Colombo and Gondzio [8].

The background for the modifications stem from observations that the corrector step can occasionally produce a step length that is shorter than the step length obtained for the predictor step and that the corrector can be an order of magnitude larger than the predictor resulting in the composite direction being dominated by the corrector direction, [8] p. 10.
To hopefully avoid these problems we use multiple centrality correctors and weight the corrector steps with a parameter $\omega \in [0, 1]$ as shown in (6.19) where $(\Delta^p x, \Delta^p \lambda, \Delta^p s)$ and $(\Delta^m x, \Delta^m \lambda, \Delta^m s)$ are the predictor and modified corrector directions respectively.

$$(\Delta x, \Delta\lambda, \Delta s) = (\Delta^p x, \Delta^p \lambda, \Delta^p s) + \omega(\Delta^m x, \Delta^m \lambda, \Delta^m s) \tag{6.19}$$

By combining these two approaches we hope to make our method more efficient and better able to adapt to any inaccuracies that might occur as a result of an excessively influencial corrector step in individual iterations. Note that we also weight the corrector step $(\Delta^c x, \Delta^c \lambda, \Delta^c s)$, in the predictor-corrector step, as shown in (6.20) where $(\Delta^a x, \Delta^a \lambda, \Delta^a s)$ is the affine scaling direction.

$$(\Delta x, \Delta\lambda, \Delta s) = (\Delta^a x, \Delta^a \lambda, \Delta^a s) + \omega(\Delta^c x, \Delta^c \lambda, \Delta^c s) \tag{6.20}$$

In each iteration we choose ω such that the steplength α is maximized in the composite direction $(\Delta x, \Delta\lambda, \Delta s)$.
In practice we select the optimal value of ω by performing a very simple line-search in the interval $[\omega_{min}, \omega_{max}] = [\alpha, 1]$ as suggested in [8]. Specifically the line search is performed by selecting 9 uniformly distributed points in the interval $[\alpha, 1]$ and computing the resulting steplength α_ω for each of the points in the interval. The value $\hat{\omega}$ providing the largest steplength α_ω is then selected.
One more thing to note about this modified approach is that we use a slightly different way of computing the enlarged stepsize $\tilde{\alpha}$ as shown in (6.21) where $\delta_\alpha = 0.3$.

$$\tilde{\alpha} = \min(1.5\alpha + \delta_\alpha, 1) \tag{6.21}$$

Finally we also use a slightly different way of detecting when to stop correcting such that we stop correcting if $\alpha_w < 1.01\alpha$.
To test the modifications to the MCC method we implement a function `correctorWeights.m` to compute weights and test it by including it in a MCC method designed to solve inequality constrained QP's. `mccQP_simplev2.m` and `correctorWeights.m` can be seen in appendices B.2 and B.3 respectively. We are interested in comparing

the performance of the previously implemented MCC method `mccQP_simplev1.m` with the new modified version including weights.
From the test set of 38 problems used in the previous chapter we select 7 inequality constrained QP's. The results from solving these 7 problems with `mccQP_simplev1.m` are recapped in table 6.4.

QP problem	Its	Time	Solution	`mcc_simplev1.m`	K
`hs21`	51	0.06	-9.99599999e+01	-9.97357626e+01	2
`hs35`	6	0.04	1.11111111e-01	1.11111111e-01	2
`hs76`	7	0.05	-4.68181818e+00	-4.68181818e+00	2
`mosarqp1`	12	46.60	-0.952875441e+03	-0.952875443e+03	2
`mosarqp2`	11	3.68	-0.159748211e+04	-0.159748212e+04	2
`values`	17	0.32	-0.139662114e+01	-0.139662114e+01	2
`zecevic2`	10	0.09	-4.12500000e+00	-4.12500000e+00	2

Table 6.4: *Results from testing implemented MCC method.*

In table 6.5 the results from solving the 7 problems with the modified MCC method are shown.

QP problem	Its	Time	Solution	`mcc_simplev2.m`	K
`hs21`	51	0.11	-9.99599999e+01	-9.97359880e+01	2
`hs35`	6	0.05	1.11111111e-01	1.11111111e-01	2
`hs76`	7	0.06	-4.68181818e+00	-4.68181818e+00	2
`mosarqp1`	12	46.68	-0.952875441e+03	-0.952875443e+03	2
`mosarqp2`	11	3.73	-0.159748211e+04	-0.159748212e+04	2
`values`	11	0.22	-0.139662114e+01	-0.139640199e+01	2
`zecevic2`	10	0.07	-4.12500000e+00	-4.12500000e+00	2

Table 6.5: *Results from testing modified MCC method.*

The results from table 6.4 and 6.5 are compared in 6.6. The accuracy of the results from the two methods are almost similar in all cases.

QP problem	Its MCC	Its MCC new	Time MCC	Time MCC new
`hs21`	51	51	0.06	0.11
`hs35`	6	6	0.04	0.05
`hs76`	7	6	0.05	0.05
`mosarqp1`	12	12	46.60	46.68
`mosarqp2`	11	11	3.68	3.73
`values`	17	11	0.32	0.28
`zecevic2`	10	10	0.09	0.07

Table 6.6: *Comparison of the MCC and the modified MCC method.*

Table 6.6 shows that the modified MCC method uses as many or fewer iterations than the MCC method in all cases. There is very little difference in computation times however. This could be due to the possible absence of problems with the potential to produce corrector directions that dominate predictor directions. To test the modified MCC method more extensively it is necessary to use test problems where the problem of the corrector dominating the predictor occurs to see how much it will affect the final outcome. The results above however does not produce any noteworthy negative effects so there is no reason to not modify the MCC method to include weights based on the above observations.
It was unfortunately not possible to produce a working version of the modified MCC method for QP's with both inequality and equality constraints.
The modified MCC method presented in [8] is developed for linear programming problems and uses two step length parameters so the modified MCC method developed in this section is a bit simplified in the sense that we use only a single step length parameter. With the new step length strategy presented in the previous section there is the option of combining the modified MCC method with the new step length strategy.

6.3 Other possible modifications

In this section we list some modifications that might also improve the performance of the developed interior-point methods.

First there is the choice of the dampening factor η. In this presentation we have kept η fixed. If this value is too large the method will fail to converge, if it is too small convergence will be slow. If η could, in a computational inexpensive manner, be chosen adaptively for each iteration, it should improve the efficiency of each iteration. Nocedal and Wright [1] p. 409, suggests that $\eta_i \to 1$ for the ith iteration, when the iterates approach the solution to accelerate convergence.

Another topic we have not touched upon is the choice of starting points. A well chosen starting point for a given problem can greatly improve the convergence rate of the method. In most cases this requires knowledge of the problem to be solved. This sort of information has not been accessible for the test problems used in this presentation. There is however also the possibility of developing a general method to warm start the methods such that the method is capable of determining a suitable starting point that ensures faster convergence.

More refined stop criteria are in some cases needed for a method to succesfully solve a given problem. An investigation of more advanced stop criteria and their effect on the convergence of the developed methods might also result in better

computational performance.

Finally it is important to take the type of problem to be solved into consideration. If for example an interior-point method is needed for solving QP's in which the Hessian matrices share the same sparsity pattern or the constraints are formulated in the same way for all the QP's to be solved, then it would perhaps be a good idea to design and adapt the method to solve that particular type of problems, i.e. optimize the computations the interior-point method including the factorization computations to exploit prior knowledge of the QP's to be solved.

Chapter 7

Conclusion

In this thesis we have explored how to solve Quadratic Programs using Interior-Point Methods. In the process of investigating the computational performance of the interior-point methods the need for test problems arose so a QPS reader was developed to gain access to a test set of QP problems. Using various test problems we investigated and compared the performance of the two interior-point methods, the predictor-corrector (PC) method and the multiple centrality corrections (MCC) method. Finally further modifications where made to some versions of these methods seeking to improve their computational performance. The most important findings are summarized below.

- The computational performance of the PC method was in almost all cases very stable. It consistently computed solutions with satisfactory numerical precision using a small number of iterations and a fairly low amount of computation time. One of the major strengths of the PC method was that only one factorization had to be computed for each iteration.
- The purpose of the MCC method was to improve the computational performance of the PC method by reducing the number of iterations. Results suggested that the MCC method has the potential to be more efficient than the PC method for problems where the number of variables and constraints is large. Results also suggested that the MCC method is less stable than the PC method.

- In order for the new step length strategy to be efficient in practice the computational cost of computing the step lengths need to be reduced significantly such that the computational cost of computing the new step lengths does not exceed the gain in computational time from using fewer iterations.
- Results showed that the MCC method with weights used as many or fewer iterations than the standard version of the MCC method for a small extra computational cost. These observations suggested that it was profitable to include weights in the MCC method regardless of the problem to be solved.

In order for the interior-point methods presented in this thesis to be applicable in practice the implementations should be done in a low level progamming language, for example FORTRAN or C, to maximize the performance of the methods. If the interior-point methods are intended to be used for QP's that share specific properties then this should be taken into account in the design process. Adapting the methods to the specific problem will further enhance the computational performance of the methods.

Appendix A

Test problem dimensions

In table A.1 dimensions of the test problems used in this presentation are shown. The table is an excerpt of a table of similar nature in [4].

- m Number of constraints
- n Number of variables
- nz Number of nonzeros in constraint matrix
- qnz Number of off-diagonal entries in the lower triangular part of the Hessian

QP problem	m	n	nz	qnz
aug3dcqp	1000	3873	6546	0
aug3dqp	1000	3873	6546	0
cont-050	2401	2597	12005	0
cvxqp1_m	500	1000	1498	2984
cvxqp1_s	50	100	148	286
cvxqp2_m	250	1000	749	2984
cvxqp2_s	25	100	74	286
cvxqp3_m	750	1000	2247	2984
cvxqp3_s	75	100	222	286
dual1	1	85	85	3473
dual2	1	96	96	4412
dual3	1	111	111	5997
dual4	1	75	75	2724
gouldqp2	349	699	1047	348
gouldqp3	349	699	1047	697
hs21	1	2	2	0
hs35	1	3	3	2
hs53	3	5	7	2
hs76	3	4	10	2
lotschd	7	12	54	0
mosarqp1	700	2500	3422	45
mosarqp2	600	900	2930	45
qpcblend	74	83	491	0
qptest	2	2	4	1
qscorpio	388	358	1426	18
qscrs8	490	1169	3182	88
qscsd1	77	760	2388	691
qscsd6	147	1350	4316	1308
qscsd8	397	2750	8584	2370
qsctap1	300	480	1692	117
qsctap2	1090	1880	6714	636
qsctap3	1480	2480	8874	861
qshare2b	96	79	694	45
stcqp1	2052	4097	13338	22506
stcqp2	2052	4097	13338	22506
tame	1	2	2	1
values	1	202	202	3620
zecevic2	2	2	4	0

Table A.1: *Test problem dimensions.*

Appendix B

Program code

B.1 PC methods

Listing B.1: pcQP_simplev1.m

```
function [x_stop,z_stop,s_stop,k] = pcQP_simplev1(x,z,s,G,g,A,b)
%-------------------------------------------------------------
% pcQP_simplev1.m
%
% This function solves a QP problem of the form
%
% min 0.5x'Gx + g'x
% s.t. A'x >= b
%
% using the Predictor-Corrector (PC) method.
%
% Input:
%     x: starting point for the vector x            (n x 1 vector)
%     z: starting point for the vector x            (nA x 1 vector)
%     s: starting point for the slack-vector x      (nA x 1 vector)
%     G: Hessian                                    (n x n matrix)
%     g:                                            (n x 1 vector)
%     A: left hand side of inequality constraints   (n x nA  matrix)
%     b: right hand side of inequality constraints  (nA x 1 vector)
%     where nC and nA are the numbers of inequality and equality
%     constraints.
% Output:
%     x_stop: solution x
%     z_stop
%     s_stop
%     k: number of iterations used
%
```

```
% Thomas Reslow Krüth, s021898
%-----------------------------------------------------------------

eta = 0.95;
%residuals are computed
[mA,nA] = size(A);
e = ones(nA,1);
rL  = G*x + g - A*z;
rs  = s - A'*x + b;
rsz = s.*z;
mu = sum(z.*s)/nA;

%k: number of iterations, epsilon: tolerances
k = 0;
maxk = 200;
eps_L = 1e-16; eps_s = 1e-16; eps_mu = 1e-16;

while (k<=maxk && norm(rL)>=eps_L && norm(rs)>=eps_s && abs(mu)>=eps_mu)
    %Solve system with a Newton-like method/Factorizarion
    G_bar = G + A*(diag(z./s))*A';
    r_bar = A*((rsz-z.*rs)./s);
    g_bar = -(rL + r_bar);
    L = chol(G_bar,'lower');
    dx_a = L'\(L\g_bar);
    ds_a = -rs + A'*dx_a;
    dz_a = -(rsz+z.*ds_a)./s;

    %Compute alpha_aff
    alpha_a = 1;
    idx_z = find(dz_a<0);
    if (isempty(idx_z)==0)
        alpha_a = min(alpha_a,min(-z(idx_z)./dz_a(idx_z)));
    end
    idx_s = find(ds_a<0);
    if (isempty(idx_s)==0)
        alpha_a = min(alpha_a,min(-s(idx_s)./ds_a(idx_s)));
    end

    %Compute the affine duality gap
    mu_a = ((z+alpha_a*dz_a)'*(s+alpha_a*ds_a))/nA;

    %Compute the centering parameter
    sigma = (mu_a/mu)^3;

    %Solve system
    rsz = rsz + ds_a.*dz_a - sigma*mu*e;
    r_bar = A*((rsz-z.*rs)./s);
    g_bar = -(rL + r_bar);
    dx = L'\(L\g_bar);
    ds = -rs + A'*dx;
    dz = -(rsz+z.*ds)./s;

    %Compute alpha
    alpha = 1;
    idx_z = find(dz<0);
    if (isempty(idx_z)==0)
        alpha = min(alpha,min(-z(idx_z)./dz(idx_z)));
    end
    idx_s = find(ds<0);
    if (isempty(idx_s)==0)
        alpha = min(alpha,min(-s(idx_s)./ds(idx_s)));
    end

    %Update x,z,s
    x = x + eta*alpha*dx;
```

```
    z = z + eta*alpha*dz;
    s = s + eta*alpha*ds;

    k = k+1;

    %Update rhs and mu
    rL  = G*x + g - A*z;
    rs  = s - A'*x + b;
    rsz = s.*z;
    mu = sum(z.*s)/nA;
end

x_stop = x;
z_stop = z;
s_stop = s;
```

Listing B.2: pcQP_simplev2.m

```
function [x_stop,y_stop,z_stop,k] = pcQP_simplev2(x,y,z,G,g,A,b)
%-------------------------------------------------------------
% pcQP_simplev2.m
%
% This function solves a QP problem of the form
% min 0.5x'Gx + g'x
% s.t. Ax  = b
%       x >= 0
% using the Predictor-Corrector (PC) method and "old" step length
    strategy.
%
% Input:
%     x: starting point for the vector x            (n x 1 vector)
%     y: starting point for the vector x            (nA x 1 vector)
%     z: starting point for the vector x            (n x 1 vector)
%     G: Hessian                                    (n x n matrix)
%     g:                                            (n x 1 vector)
%     A: left hand side of equality constraints     (nA x n  matrix)
%     b: right hand side of equality constraints    (nA x 1 vector)
%     where nC and nA are the numbers of inequality and equality
%     constraints.
% Output:
%     x_stop: solution x
%     y_stop
%     s_stop
%     k: number of iterations used
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------

%residuals are computed
n = length(x);
[mA,nA] = size(A);
e = ones(n,1);
eta = 0.95;
rd  = G*x + g - A'*y - z;
rp  = b - A*x;
rc = -x.*z;
mu = sum(x.*z)/n;

%k: number of iterations, epsilon: tolerances
k = 0;
eps_L = 1e-16; eps_s = 1e-16; eps_mu = 1e-16;

while (k<=100 && norm(rd)>=eps_L && norm(rp)>=eps_s && abs(mu)>=eps_mu)
    %Solve system with a Newton-like method/Factorizarion
```

```
    lhs = [-G,A',eye(n);...
           A,zeros(mA,mA),zeros(mA,n);...
           diag(z),zeros(n,mA),diag(x)];
    rhs = [rd;rp;rc];
    dxyz_a = lhs\rhs;
    dx_a = dxyz_a(1:n);
    dy_a = dxyz_a(n+1:n+length(y));
    dz_a = dxyz_a(n+length(y)+1:end);

    %Compute alpha_aff
    idx_x = find(dx_a<0);
    if (isempty(idx_x)==0)
        alpha_xa = eta*min(1,min(-x(idx_x)./dx_a(idx_x)));
    end
    idx_z = find(dz_a<0);
    if (isempty(idx_z)==0)
        alpha_za = eta*min(1,min(-z(idx_z)./dz_a(idx_z)));
    end
    alpha_a = min(alpha_xa,alpha_za);

    %Compute the affine duality gap
    mu_a = ((x+alpha_a*dx_a)'*(z+alpha_a*dz_a))/n;

    %Compute the centering parameter
    sigma = (mu_a/mu)^3;

    %Solve system
    rc = rc + sigma*mu*e;
    rhs = [rd;rp;rc];
    dxyz = lhs\rhs;
    dx = dxyz(1:n);
    dy = dxyz(n+1:n+length(y));
    dz = dxyz(n+length(y)+1:end);

    %Compute alpha
    idx_z = find(dz<0);
    if (isempty(idx_z)==0)
        alpha_z = eta*min(1,min(-z(idx_z)./dz(idx_z)));
    end
    idx_x = find(dx<0);
    if (isempty(idx_x)==0)
        alpha_x = eta*min(1,min(-x(idx_x)./dx(idx_x)));
    end
    alpha = min(alpha_x,alpha_z);

    %Update x,z,s
    x = x + alpha*dx;
    y = y + alpha*dy;
    z = z + alpha*dz;

    k = k+1;

    %Update rhs
    rd  = G*x + g - A'*y - z;
    rp  = b - A*x;
    rc = -x.*z;
    mu = sum(x.*z)/n;
end
x_stop = x;
y_stop = y;
z_stop = z;
```

Listing B.3: pcQP_simplev2new.m

```
function [x_stop,y_stop,z_stop,k] = pcQP_simplev2new(x,y,z,G,g,A,b)
%-------------------------------------------------------------------
% pcQP_simplev2new.m
%
% This function solves a QP problem of the form
% min 0.5x'Gx + g'x
% s.t. Ax  = b
%       x >= 0
% using the Predictor-Corrector (PC) method and new step length strategy
    .
%
% Input:
%     x: starting point for the vector x            (n x 1 vector)
%     y: starting point for the vector x            (nA x 1 vector)
%     z: starting point for the vector x            (n x 1 vector)
%     G: Hessian                                    (n x n matrix)
%     g:                                            (n x 1 vector)
%     A: left hand side of equality constraints     (nA x n  matrix)
%     b: right hand side of equality constraints    (nA x 1 vector)
%     where nC and nA are the numbers of inequality and equality
%     constraints.
% Output:
%     x_stop: solution x
%     y_stop
%     s_stop
%     k: number of iterations used
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------------

%residuals are computed
n = length(x);
[mA,nA] = size(A);
e = ones(n,1);
eta = 0.95;
rd  = G*x + g - A'*y - z;
rp  = b - A*x;
rc = -x.*z;
mu = sum(x.*z)/n;

%k: number of iterations, epsilon: tolerances
k = 0;
maxk = 200;
eps_L = 1e-16; eps_s = 1e-16; eps_mu = 1e-16;

while (k<=maxk && norm(rd)>=eps_L && norm(rp)>=eps_s && abs(mu)>=eps_mu)
    %Solve system with a Newton-like method/Factorizarion
    lhs = [-G,A',eye(n);...
           A,zeros(mA,mA),zeros(mA,n);...
           diag(z),zeros(n,mA),diag(x)];
    rhs = [rd;rp;rc];
    dxyz_a = lhs\rhs;
    dx_a = dxyz_a(1:n);
    dy_a = dxyz_a(n+1:n+length(y));
    dz_a = dxyz_a(n+length(y)+1:end);

    %Compute alpha_aff
    idx_x = find(dx_a<0);
    if (isempty(idx_x)==0)
        alpha_xa = eta*min(1,min(-x(idx_x)./dx_a(idx_x)));
    end
    idx_z = find(dz_a<0);
    if (isempty(idx_z)==0)
        alpha_za = eta*min(1,min(-z(idx_z)./dz_a(idx_z)));
    end
```

```
    alpha_a = min(alpha_xa,alpha_za);
    [ax_a,ay_a,az_a] = newStepLengths(alpha_xa,alpha_za,...
                                alpha_a,rd,rp,A,G,dx_a,dy_a,dz_a,x,z);

    %Compute the affine duality gap
    mu_a = ((x+ax_a*dx_a)'*(z+az_a*dz_a))/n;

    %Compute the centering parameter
    sigma = (mu_a/mu)^3;

    %Solve system
    rc = rc + sigma*mu*e;
    rhs = [rd;rp;rc];
    dxyz = lhs\rhs;
    dx = dxyz(1:n);
    dy = dxyz(n+1:n+length(y));
    dz = dxyz(n+length(y)+1:end);

    %Compute alpha
    idx_z = find(dz<0);
    if (isempty(idx_z)==0)
        alpha_z = eta*min(1,min(-z(idx_z)./dz(idx_z)));
    end
    idx_x = find(dx<0);
    if (isempty(idx_x)==0)
        alpha_x = eta*min(1,min(-x(idx_x)./dx(idx_x)));
    end
    alpha = min(alpha_x,alpha_z);
    [ax,ay,az] = newStepLengths(alpha_x,alpha_z,...
                                alpha,rd,rp,A,G,dx,dy,dz,x,z);
    %Update x,z,s
    x = x + ax*dx;
    y = y + ay*dy;
    z = z + az*dz;

    k = k+1;

    %Update rhs
    rd  = G*x + g - A'*y - z;
    rp  = b - A*x;
    rc = -x.*z;
    mu = sum(x.*z)/n;
end
x_stop = x;
y_stop = y;
z_stop = z;
```

Listing B.4: pcQP_gen.m

```
function [x_stop,y_stop,z_stop,s_stop,k] = pcQP_gen(x,y,z,s,G,g,C,d,A,b)
%-------------------------------------------------------------
% pcQP_gen.m
%
% This function solves a QP problem of the form
% min 0.5x'Gx + g'x
% s.t. A'x  = b
%      C'x >= d
% using the Predictor-Corrector (PC) method.
%
% Input:
%     x: starting point for the vector x            (n x 1 vector)
%     y: starting point for the vector x            (nA x 1 vector)
%     z: starting point for the vector x            (nC x 1 vector)
%     s: starting point for the slack-vector x      (nC x 1 vector)
```

```
%     G: Hessian                                          (n x n matrix)
%     g:                                                  (n x 1 vector)
%     C: left hand side of inequality constraints         (n x nC matrix)
%     d: right hand side of inequality constraints        (nC x 1 vector)
%     A: left hand side of equality constraints           (n x nA  matrix)
%     b: right hand side of equality constraints          (nA x 1 vector)
%     where nC and nA are the numbers of inequality and equality
%     constraints.
% Output:
%     x_stop: solution x
%     y_stop
%     z_stop
%     s_stop
%     k: number of iterations used
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------

%dampening factor eta
eta = 0.95;
%residuals are computed
[mA,nA] = size(A);
[mC,nC] = size(C);
e = ones(nC,1);
rL  = G*x + g - A*y - C*z;
rA = -A'*x + b;
rC = -C'*x + s + d;
rsz = s.*z;
mu = sum(z.*s)/nC;

%k: number of iterations, epsilon: tolerances
k = 0;
maxk = 200;
eps_L = 1e-10; eps_A = 1e-10; eps_C = 1e-10; eps_mu = 1e-10;

while (k<=maxk && norm(rL)>=eps_L && norm(rA)>=eps_A && norm(rC)>=eps_C
    && abs(mu)>=eps_mu)
    %Solve system with a Newton-like method/Factorizarion
    lhs = [G,-A,-C;-A',spalloc(nA,nA,0),spalloc(nA,nC,0);-C',spalloc(nC,
        nA,0),sparse(-diag(s./z))];
    [L,D,P] = ldl(lhs);
    rhs = [-rL;-rA;-rC+rsz./z];
    dxyz_a = P*(L'\(D\(L\(P'*rhs))));
    dx_a = dxyz_a(1:length(x));
    dy_a = dxyz_a(length(x)+1:length(x)+length(y));
    dz_a = dxyz_a(length(x)+length(y)+1:length(x)+length(y)+length(z));
    ds_a = -((rsz+s.*dz_a)./z);

    %Compute alpha_aff
    alpha_a = 1;
    idx_z = find(dz_a<0);
    if (isempty(idx_z)==0)
        alpha_a = min(alpha_a,min(-z(idx_z)./dz_a(idx_z)));
    end
    idx_s = find(ds_a<0);
    if (isempty(idx_s)==0)
        alpha_a = min(alpha_a,min(-s(idx_s)./ds_a(idx_s)));
    end

    %Compute the affine duality gap
    mu_a = ((z+alpha_a*dz_a)'*(s+alpha_a*ds_a))/nC;

    %Compute the centering parameter
    sigma = (mu_a/mu)^3;
```

```
    %Solve system
    rsz = rsz + ds_a.*dz_a - sigma*mu*e;
    rhs = [-rL;-rA;-rC+rsz./z];
    dxyz = P*(L'\(D\(L\(P'*rhs))));
    dx = dxyz(1:length(x));
    dy = dxyz(length(x)+1:length(x)+length(y));
    dz = dxyz(length(x)+length(y)+1:length(x)+length(y)+length(z));
    ds = -((rsz+s.*dz)./z);

    %Compute alpha
    alpha = 1;
    idx_z = find(dz<0);
    if (isempty(idx_z)==0)
        alpha = min(alpha,min(-z(idx_z)./dz(idx_z)));
    end
    idx_s = find(ds<0);
    if (isempty(idx_s)==0)
        alpha = min(alpha,min(-s(idx_s)./ds(idx_s)));
    end

    %Update x,z,s
    x = x + eta*alpha*dx;
    y = y + eta*alpha*dy;
    z = z + eta*alpha*dz;
    s = s + eta*alpha*ds;

    k = k+1;

    %Update rhs
    rL  = G*x + g - A*y - C*z;
    rA = -A'*x + b;
    rC = -C'*x + s + d;
    rsz = s.*z;
    mu = sum(z.*s)/nC;
end

%Output
x_stop = x;
y_stop = y;
z_stop = z;
s_stop = s;
```

B.2 MCC methods

Listing B.5: mccQP_simplev1.m

```
function [x_stop,z_stop,s_stop,k] = mccQP_simplev1(x,z,s,G,g,A,b)
%-------------------------------------------------------------
% mccQP_simplev1.m
%
% This function solves a QP problem of the form
%
% min 0.5x'Gx + g'x
% s.t. A'x >= b
%
% using the Multiple Centrality Corrections (MCC) method.
%
% Input:
%     x: starting point for the vector x           (n x 1 vector)
```

```
%     z: starting point for the vector x                (nA x 1 vector)
%     s: starting point for the slack-vector x          (nA x 1 vector)
%     G: Hessian                                        (n x n matrix)
%     g:                                                (n x 1 vector)
%     A: left hand side of equality constraints         (n x nA  matrix)
%     b: right hand side of equality constraints        (nA x 1 vector)
%     where nC and nA are the numbers of inequality and equality
%     constraints.
% Output:
%     x_stop: solution x
%     z_stop
%     s_stop
%     k: number of iterations used
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------

B_min = 0.1; B_max = 10; d_alpha = 0.3; gamma = 0.1;
%residuals are computed
[mA,nA] = size(A);
e = ones(nA,1);
rL  = G*x + g - A*z;
rs  = s - A'*x + b;
rsz = s.*z;
mu = sum(z.*s)/nA;

%k: number of iterations, epsilon: tolerances
k = 0;
maxk = 200;
eps_L = 1e-16; eps_s = 1e-16; eps_mu = 1e-16;

while (k<=maxk && norm(rL)>=eps_L && norm(rs)>=eps_s && abs(mu)>=eps_mu)
    %Solve system with a Newton-like method/Factorizarion
    G_bar = G + A*(diag(z./s))*A';
    r_bar = A*((rsz-z.*rs)./s);
    g_bar = -(rL + r_bar);
    L = chol(G_bar,'lower');
    dx_a = L'\(L\g_bar);
    ds_a = -rs + A'*dx_a;
    dz_a = -(rsz+z.*ds_a)./s;

    %Compute alpha_aff
    alpha_a = 1;
    idx_z = find(dz_a<0);
    if (isempty(idx_z)==0)
        alpha_a = min(alpha_a,min(-z(idx_z)./dz_a(idx_z)));
    end
    idx_s = find(ds_a<0);
    if (isempty(idx_s)==0)
        alpha_a = min(alpha_a,min(-s(idx_s)./ds_a(idx_s)));
    end

    %Compute the affine duality gap
    mu_a = ((z+alpha_a*dz_a)'*(s+alpha_a*ds_a))/nA;

    %Compute the centering parameter
    sigma = (mu_a/mu)^3;
    mu_t = mu*sigma;

    %Solve system
    rsz = ds_a.*dz_a - sigma*mu*e;
    r_bar = A*(rsz./s);
    g_bar = -r_bar;
    dx_c = L'\(L\g_bar);
    ds_c = A'*dx_c;
```

```
dz_c = -(rsz+z.*ds_c)./s;

dx_p = dx_a + dx_c;
dz_p = dz_a + dz_c;
ds_p = ds_a + ds_c;

%Compute alpha_p
alpha_p = 1;
idx_z = find(dz_p<0);
if (isempty(idx_z)==0)
    alpha_p = min(alpha_p,min(-z(idx_z)./dz_p(idx_z)));
end
idx_s = find(ds_p<0);
if (isempty(idx_s)==0)
    alpha_p = min(alpha_p,min(-s(idx_s)./ds_p(idx_s)));
end

%Modified centering directions
alpha_h = min(1.5*alpha_p+d_alpha,1);

%Number of corrections
K=2;

%Corrections
j = 1;
while(j<=K)
    %Compute trial point
    z_t = z + alpha_h*dz_p;
    s_t = s + alpha_h*ds_p;
    %Define target
    v_thilde = s_t.*z_t;
    v_t = v_thilde;
    for i=1:length(v_t)
        if(v_t(i)<B_min*mu_t)
            v_t(i) = B_min*mu_t;
        elseif(v_t(i)>B_max*mu_t)
            v_t(i) = B_max*mu_t;
        end
    end
    %Compute corrector
    rsz = (v_t - v_thilde);
    for i=1:length(rsz)
        if(rsz(i)<(-B_max*mu_t))
            rsz(i) = -B_max*mu_t;
        end
    end
    r_bar = A*(rsz./s);
    g_bar = -r_bar;
    dx_m = L'\(L\g_bar);
    ds_m = A'*dx_m;
    dz_m = -rsz./s-diag(z./s)*ds_m;
    %Composite direction
    dx = dx_p + dx_m;
    ds = ds_p + ds_m;
    dz = dz_p + dz_m;

    %Compute alpha
    alpha = 1;
    idx_z = find(dz<0);
    if (isempty(idx_z)==0)
        alpha = min(alpha,min(-z(idx_z)./dz(idx_z)));
    end
    idx_s = find(ds<0);
    if (isempty(idx_s)==0)
        alpha = min(alpha,min(-s(idx_s)./ds(idx_s)));
```

```
        end

        %Test for improvement
        if(alpha >= alpha_p + gamma*d_alpha)
            j=j+1;
            dx_p = dx;
            ds_p = ds;
            dz_p = dz;
            alpha_p = alpha;
            alpha_h = min(1.5*alpha_p+d_alpha,1);
        else
            dx = dx_p;
            ds = ds_p;
            dz = dz_p;
            j=10;
        end
    end

    %Update x,z,s
    eta = 0.99995;
    x = x + eta*alpha*dx;
    z = z + eta*alpha*dz;
    s = s + eta*alpha*ds;

    k = k+1;

    %Update rhs
    rL  = G*x + g - A*z;
    rs  = s - A'*x + b;
    rsz = s.*z;
    mu = sum(z.*s)/nA;
end

x_stop = x;
z_stop = z;
s_stop = s;
```

Listing B.6: mccQP_simplev2.m

```
function [x_stop,z_stop,s_stop,k] = mccQP_simplev2(x,z,s,G,g,A,b)
%-------------------------------------------------------------
% mccQP_simplev2.m
%
% This function solves a QP problem of the form
%
% min 0.5x'Gx + g'x
% s.t. A'x >= b
%
% using the modified Multiple Centrality Corrections (MCC) method.
% Modified MCC: applies weights to corrector directions
% Input:
%     x: starting point for the vector x            (n x 1 vector)
%     z: starting point for the vector x            (nA x 1 vector)
%     s: starting point for the slack-vector x      (nA x 1 vector)
%     G: Hessian                                    (n x n matrix)
%     g:                                            (n x 1 vector)
%     A: left hand side of equality constraints     (n x nA  matrix)
%     b: right hand side of equality constraints    (nA x 1 vector)
%     where nC and nA are the numbers of inequality and equality
%     constraints.
% Output:
%     x_stop: solution x
%     z_stop
%     s_stop
```

```
%      k: number of iterations used
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------------

B_min = 0.1; B_max = 10; d_alpha = 0.3;
%residuals are computed
[mA,nA] = size(A);
e = ones(nA,1);
rL  = G*x + g - A*z;
rs  = s - A'*x + b;
rsz = s.*z;
mu = sum(z.*s)/nA;

%k: number of iterations, epsilon: tolerances
k = 0;
maxk = 200;
eps_L = 1e-16; eps_s = 1e-16; eps_mu = 1e-16;

while (k<=maxk && norm(rL)>=eps_L && norm(rs)>=eps_s && abs(mu)>=eps_mu)
    %Solve system with a Newton-like method/Factorizarion
    G_bar = G + A*(diag(z./s))*A';
    r_bar = A*((rsz-z.*rs)./s);
    g_bar = -(rL + r_bar);
    L = chol(G_bar,'lower');
    dx_a = L'\(L\g_bar);
    ds_a = -rs + A'*dx_a;
    dz_a = -(rsz+z.*ds_a)./s;

    %Compute alpha_aff
    alpha_a = 1;
    idx_z = find(dz_a<0);
    if (isempty(idx_z)==0)
        alpha_a = min(alpha_a,min(-z(idx_z)./dz_a(idx_z)));
    end
    idx_s = find(ds_a<0);
    if (isempty(idx_s)==0)
        alpha_a = min(alpha_a,min(-s(idx_s)./ds_a(idx_s)));
    end

    %Compute the affine duality gap
    mu_a = ((z+alpha_a*dz_a)'*(s+alpha_a*ds_a))/nA;

    %Compute the centering parameter
    sigma = (mu_a/mu)^3;
    mu_t = mu*sigma;

    %Solve system
    rsz = ds_a.*dz_a - sigma*mu*e;
    r_bar = A*(rsz./s);
    g_bar = -r_bar;
    dx_c = L'\(L\g_bar);
    ds_c = A'*dx_c;
    dz_c = -(rsz+z.*ds_c)./s;

    dz_p = dz_a + dz_c;
    ds_p = ds_a + ds_c;

    %Compute alpha_p
    alpha_p = 1;
    idx_z = find(dz_p<0);
    if (isempty(idx_z)==0)
        alpha_p = min(alpha_p,min(-z(idx_z)./dz_p(idx_z)));
    end
    idx_s = find(ds_p<0);
```

```
if (isempty(idx_s)==0)
    alpha_p = min(alpha_p,min(-s(idx_s)./ds_p(idx_s)));
end

[w_hat,alpha_w] = correctorWeight(alpha_p,dz_a,ds_a,dz_c,ds_c,z,s);

dx_p = dx_a + w_hat*dx_c;
dz_p = dz_a + w_hat*dz_c;
ds_p = ds_a + w_hat*ds_c;

%Modified centering directions
alpha_h = min(1.5*alpha_w+d_alpha,1);

%Number of corrections
K=2;

%Corrections
j = 1;
while(j<=K)
    %Compute trial point
    z_t = z + alpha_h*dz_p;
    s_t = s + alpha_h*ds_p;
    %Define target
    v_thilde = s_t.*z_t;
    v_t = v_thilde;
    for i=1:length(v_t)
        if(v_t(i)<B_min*mu_t)
            v_t(i) = B_min*mu_t;
        elseif(v_t(i)>B_max*mu_t)
            v_t(i) = B_max*mu_t;
        end
    end
    %Compute corrector
    rsz = (v_t - v_thilde);
    for i=1:length(rsz)
        if(rsz(i)<(-B_max*mu_t))
            rsz(i) = -B_max*mu_t;
        end
    end
    r_bar = A*(rsz./s);
    g_bar = -r_bar;
    dx_m = L'\(L\g_bar);
    ds_m = A'*dx_m;
    dz_m = -rsz./s-diag(z./s)*ds_m;
    %Composite direction
    ds = ds_p + ds_m;
    dz = dz_p + dz_m;

    %Compute alpha
    alpha = 1;
    idx_z = find(dz<0);
    if (isempty(idx_z)==0)
        alpha = min(alpha,min(-z(idx_z)./dz(idx_z)));
    end
    idx_s = find(ds<0);
    if (isempty(idx_s)==0)
        alpha = min(alpha,min(-s(idx_s)./ds(idx_s)));
    end

    [w_hat,alpha_w] = correctorWeight(alpha,dz_p,ds_p,dz_m,ds_m,z,s)
        ;

    dx = dx_p + w_hat*dx_m;
    dz = dz_p + w_hat*dz_m;
    ds = ds_p + w_hat*ds_m;
```

```
        %Test for improvement
        if(alpha >= 1.01*alpha_w)
            j=j+1;
            dx_p = dx;
            ds_p = ds;
            dz_p = dz;
            alpha_p = alpha_w;
            alpha_h = min(1.5*alpha_p+d_alpha,1);
        else
            dx = dx_p;
            ds = ds_p;
            dz = dz_p;
            j=10;
        end
    end

    %Update x,z,s
    eta = 0.99995;
    x = x + eta*alpha_w*dx;
    z = z + eta*alpha_w*dz;
    s = s + eta*alpha_w*ds;

    k = k+1;

    %Update rhs
    rL  = G*x + g - A*z;
    rs  = s - A'*x + b;
    rsz = s.*z;
    mu = sum(z.*s)/nA;
end

x_stop = x;
z_stop = z;
s_stop = s;
```

Listing B.7: mccQP_gen.m

```
function [x_stop,y_stop,z_stop,s_stop,k] = mccQP_gen(x,y,z,s,G,g,C,d,A,b
    )
%-------------------------------------------------------------
% mccQP_gen.m
%
% This function solves a QP problem of the form
% min 0.5x'Gx + g'x
% s.t. A'x  = b
%      C'x >= d
% using the Multiple Centrality Corrections (MCC) method.
%
% Input:
%     x: starting point for the vector x            (n x 1 vector)
%     y: starting point for the vector x            (nA x 1 vector)
%     z: starting point for the vector x            (nC x 1 vector)
%     s: starting point for the slack-vector x      (nC x 1 vector)
%     G: Hessian                                    (n x n matrix)
%     g:                                            (n x 1 vector)
%     C: left hand side of inequality constraints   (n x nC matrix)
%     d: right hand side of inequality constraints  (nC x 1 vector)
%     A: left hand side of equality constraints     (n x nA  matrix)
%     b: right hand side of equality constraints    (nA x 1 vector)
%     where nC and nA are the numbers of inequality and equality
%     constraints.
% Output:
%     x_stop: solution x
```

```
%       y_stop
%       z_stop
%       s_stop
%       k: number of iterations used
%
% Thomas Reslow Krüth, s021898
%-----------------------------------------------------------------

B_min = 0.1; B_max = 10; d_alpha = 0.1; gamma = 0.1;
eta = 0.99995;
%residuals are computed
[mA,nA] = size(A);
[mC,nC] = size(C);
e = ones(nC,1);
rL  = G*x + g - A*y - C*z;
rA = -A'*x + b;
rC = -C'*x + s + d;
rsz = s.*z;
mu = sum(z.*s)/nC;

%k: number of iterations, epsilon: tolerances
k = 0;
maxk = 200;
eps_L = 1e-10; eps_A = 1e-10; eps_C = 1e-10; eps_mu = 1e-10;

while (k<=maxk && norm(rL)>=eps_L && norm(rA)>=eps_A && norm(rC)>=eps_C
    && abs(mu)>=eps_mu)
    %Solve system with a Newton-like method/Factorizarion
    lhs = [G,-A,-C;-A',spalloc(nA,nA,0),spalloc(nA,nC,0);-C',spalloc(nC,
        nA,0),sparse(-diag(s./z))];
    [L,D,P] = ldl(lhs);
    rhs = [-rL;-rA;-rC+rsz./z];
    dxyz_a = P*(L'\(D\(L\(P'*rhs))));
    dz_a = dxyz_a(length(x)+length(y)+1:length(x)+length(y)+length(z));
    ds_a = -((rsz+s.*dz_a)./z);

    %Compute alpha_aff
    alpha_a = 1;
    idx_z = find(dz_a<0);
    if (isempty(idx_z)==0)
        alpha_a = min(alpha_a,min(-z(idx_z)./dz_a(idx_z)));
    end
    idx_s = find(ds_a<0);
    if (isempty(idx_s)==0)
        alpha_a = min(alpha_a,min(-s(idx_s)./ds_a(idx_s)));
    end

    %Compute the affine duality gap
    mu_a = ((z+alpha_a*dz_a)'*(s+alpha_a*ds_a))/nC;

    %Compute the centering parameter
    sigma = (mu_a/mu)^3;
    mu_t = sigma*mu;

    rsz = rsz + ds_a.*dz_a - mu_t*e;
    rhs = [-rL;-rA;-rC+rsz./z];
    dxyz_p = P*(L'\(D\(L\(P'*rhs))));
    dx_p = dxyz_p(1:length(x));
    dy_p = dxyz_p(length(x)+1:length(x)+length(y));
    dz_p = dxyz_p(length(x)+length(y)+1:length(x)+length(y)+length(z));
    ds_p = -((rsz+s.*dz_p)./z);

    %Compute alpha
    alpha_p = 1;
    idx_z = find(dz_p<0);
```

```
if (isempty(idx_z)==0)
    alpha_p = min(alpha_p,min(-z(idx_z)./dz_p(idx_z)));
end
idx_s = find(ds_p<0);
if (isempty(idx_s)==0)
    alpha_p = min(alpha_p,min(-s(idx_s)./ds_p(idx_s)));
end

%Modified centering directions
alpha_h = min(alpha_p+d_alpha,1);

%Maximum number of corrections
K=2;

%Corrections
j = 1;
while(j<=K)
    %Compute trial point
    z_t = z + alpha_h*dz_p;
    s_t = s + alpha_h*ds_p;
    %Define target
    v_thilde = s_t.*z_t;
    v_tmp = v_thilde;
    for i=1:length(v_tmp)
        if(v_tmp(i)<B_min*mu_t)
            v_tmp(i) = B_min*mu_t;
        elseif(v_tmp(i)>B_max*mu_t)
            v_tmp(i) = B_max*mu_t;
        end
    end
    v_t = v_tmp;
    %Compute corrector
    rsz = -(v_t - v_thilde);
    for i=1:length(rsz)
        if(rsz(i)<(-B_max*mu_t))
            rsz(i) = -B_max*mu_t;
        end
    end

    rhs = [zeros(length(rL),1);zeros(length(rA),1);rsz./z];
    dxyz_m = P*(L'\(D\(L\(P'*rhs))));
    dx_m = dxyz_m(1:length(x));
    dy_m = dxyz_m(length(x)+1:length(x)+length(y));
    dz_m = dxyz_m(length(x)+length(y)+1:length(x)+length(y)+length(z
        ));
    ds_m = -((rsz+s.*dz_m)./z);
    %Composite direction
    dx = dx_p + dx_m;
    ds = ds_p + ds_m;
    dy = dy_p + dy_m;
    dz = dz_p + dz_m;

    %Compute alpha
    alpha = 1;
    idx_z = find(dz<0);
    if (isempty(idx_z)==0)
        alpha = min(alpha,min(-z(idx_z)./dz(idx_z)));
    end
    idx_s = find(ds<0);
    if (isempty(idx_s)==0)
        alpha = min(alpha,min(-s(idx_s)./ds(idx_s)));
    end

    %Test for improvement
    if(alpha >= alpha_p + gamma*d_alpha)
```

```
            j=j+1;
            dx_p = dx;
            ds_p = ds;
            dz_p = dz;
            dy_p = dy;
            alpha_p = alpha;
            alpha_h = min(alpha_p+d_alpha,1);
        else
            dx = dx_p;
            ds = ds_p;
            dz = dz_p;
            dy = dy_p;
            j=10;
        end
    end %inner while-loop

    %Update x,y,z,s
    x = x + eta*alpha*dx;
    y = y + eta*alpha*dy;
    z = z + eta*alpha*dz;
    s = s + eta*alpha*ds;

    k = k+1;

    %Update rhs
    rL  = G*x + g - A*y - C*z;
    rA = -A'*x + b;
    rC = -C'*x + s + d;
    rsz = s.*z;
    mu = sum(z.*s)/nC;
end

%Output
x_stop = x;
y_stop = y;
z_stop = z;
s_stop = s;
```

B.3 Modification functions

Listing B.8: newStepLengths.m

```
function [ax,ay,az] = newStepLengths(axbar,azbar,abar,rd,rp,A,G,dx,dy,dz
    ,x,z)
%-------------------------------------------------------------
% newStepLengths.m
%
% This function solves two 2-dimensional QP's to obtain the step length
% pamaters ax, ay and az for QP problems of the form
%
% min 0.5x'Gx + g'x
% s.t. Ax  = b
%       x >= 0
%
% Input:
%     axbar: step length for x
%     azbar: step length for z
%     abar: min(axbar,azbar)
%     rd: dual residual vector
```

```
%     rp: primal residual vector
%     A: left hand side of equality constraints     (nA x n  matrix)
%     G: Hessian                                    (n x n matrix)
%     dx: search direction for x
%     dy: search direction for y
%     dz: search direction for z
%     x: current iterate x
%     z: current iterate z
%
% Output:
%     ax: step length for x
%     ay: step length for y
%     az: step length for z
%
% Thomas Reslow Krüth, s021898
%-----------------------------------------------------------------
r = [rp;rd];
s = [-A;G]*dx;
t = [zeros(length(dy),length(dy));-A']*dy;
u = [zeros(length(dy),length(dx));-eye(length(dz),length(dz))]*dz;

%Compute first trial point (ax_a,ay_a,az_a)
%-----------------First qp prob: min q1-------------------
Q1 = [(s+u)'*(s+u)+dx'*dz,(s+u)'*t;...
      (s+u)'*t,t'*t];
c1 = [r'*(s+u)+0.5*(dx'*z+x'*dz);...
      r'*t];
l1 = [0;-inf];
u1 = [abar;inf];
x1 = quadprog(Q1,c1,[],[],[],[],l1,u1);
ax_a = x1(1);
ay_a = x1(2);
az_a = ax_a;
%-----------------------------------------------------------------
%Compute second trial point (ax_b,ay_b,az_b)
if(axbar<=azbar)
    %-----------------Second qp prob: min q2-----------
    Q2 = [t'*t,t'*u;...
          t'*u,u'*u];
    c2 = [r'*t+axbar*s'*t;...
          r'*u+0.5*x'*dz+axbar*(s'*u+0.5*dx'*dz)];
    if(abar==azbar)
        abar = abar - 1e-12;
    end
    l2 = [-inf;abar];
    u2 = [inf;azbar];
    x2 = quadprog(Q2,c2,[],[],[],[],l2,u2);
    ax_b = axbar;
    ay_b = x2(1);
    az_b = x2(2);
    %-----------------------------------------------------
else
    %-----------------Third qp prob: min q3--------
    Q3 = [s'*s,s'*t;...
          s'*t,t'*t];
    c3 = [r'*s+0.5*dx'*z+azbar*(s'*u+0.5*dx'*dz);...
          r'*t+azbar*t'*u];
    if(abar==axbar)
          abar = abar - 1e-12;
    end
    l3 = [abar;-inf];
    u3 = [axbar;inf];
    x3 = quadprog(Q3,c3,[],[],[],[],l3,u3);
    ax_b = x3(1);
    ay_b = x3(2);
```

```
    az_b = azbar;
    %-------------------------------------------------------------
end
%Choose trial point with smallest merit function value
phi_a = norm(r + ax_a*s + ay_a*t + az_a*u)^2+(x+ax_a*dx)'*(z+az_a*dz);
phi_b = norm(r + ax_b*s + ay_b*t + az_b*u)^2+(x+ax_b*dx)'*(z+az_b*dz);
if(phi_a<=phi_b)
    ax = ax_a;
    ay = ay_a;
    az = az_a;
else
    ax = ax_b;
    ay = ay_b;
    az = az_b;
end
```

Listing B.9: correctorWeight.m

```
function [w_hat,alpha_w] = correctorWeight(alpha,dz_p,ds_p,dz_c,ds_c,z,s
    )
%-------------------------------------------------------------
% correctorWeight.m
%
% This function determines a weight w_hat for a corrector direction by
% performing a simple line search.
%
% Input:
%     alpha: step length
%     dz_p: predictor search direction for z
%     ds_p: predictor search direction for s
%     dz_c: corrector search direction for z
%     ds_c: corrector search direction for z
%     z: current iterate z
%     s: current iterate s
%
% Output:
%     w_hat: weight for the corrector direction
%     alpha_w: new step length
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------
w_min = alpha;
w_max = 1;
w_hat = 1;
alpha_w = alpha;

step_l = (w_max-w_min)/10;
w_vec = w_min:step_l:w_max;

for i=1:length(w_vec)
    dz = dz_p + w_vec(i)*dz_c;
    ds = ds_p + w_vec(i)*ds_c;
    alpha_p = 1;
    idx_z = find(dz<0);
    if (isempty(idx_z)==0)
        alpha_p = min(alpha_p,min(-z(idx_z)./dz(idx_z)));
    end
    idx_s = find(ds<0);
    if (isempty(idx_s)==0)
        alpha_p = min(alpha_p,min(-s(idx_s)./ds(idx_s)));
    end
    if(alpha_p>alpha)
        alpha_w = alpha_p;
        w_hat = w_vec(i);
```

```
    end
end
```

B.4 QPS reader and converter

Listing B.10: QPSreader.m

```
function qpProblem = QPSreader(filename)
%-------------------------------------------------------------
% QPSreader.m
%
% This function reads a QPS file and gives the following output.
%
%    Name: Name of the objective function
%
%    A_raw: Matrix in which rows represent either an equality constraint
     or
%    an inequality constraint, i.e.
%                        A_raw(i,:)*x >= rhs_raw(i) OR
%                        A_raw(i,:)*x <= rhs_raw(i) OR
%                        A_raw(i,:)*x  = rhs_raw(i)
%
%    rhs_raw: Vector containing right hand side values corresponding to
%    each row in A_raw as explained above
%
%    constraintType: Vector containing single-letter strings each
%    describing the constrainttype of each row in A_raw, i.e.
%                        constraintType(i) = 'N' (objective function) OR
%                        constraintType(i) = 'E' (=)                  OR
%                        constraintType(i) = 'L' (<=)                 OR
%                        constraintType(i) = 'G' (>=)
%
%    lo_vec: Vector containing the lower bounds on x
%    up_vec: Vector containing the upper bounds on x such that
%                        lo_vec <= x <= up_bound
%            Note that the value infinity (Inf) might occur in lo_vec or
%            up_vec.
%
%    ranges_raw: Vector containing the values r = u - h where u and h
     are
%    defined as h <= constraint <= u. For information on how to compute
     u
%    and h given r and the vector rhs_raw see
%    http://lpsolve.sourceforge.net/5.5/mps-format.htm.
%
%    idx_rangesraw: Vector containing the indexes of the constraints in
%    RANGES such that the type of each constraint can be identified and
     the
%    corresponding values from rhs_raw can be extracted.
%
%    G: Hessian matrix of the problem
%
%    g_vec: Vector g from the objective function f(x) = 0.5x'Gx + g'x +
     g0
%
%    g0: Scalar g0 from the objective function f(x) = 0.5x'Gx + g'x + g0
%
%    bv_li_ui_sc: Scalar equal to 0 or 1. If equal to 1 then the problem
     in
```

```
%    the qps file contains bounds of the type BV, LI, UI or SC
%                       BV:   binary variable    x = 0 or 1
%                       LI:   integer variable   b <= x (< +inf)
%                       UI:   integer variable   (0 <=) x <= b
%                       SC:   semi-cont variable x = 0 or l <= x <= b
%                             l is the lower bound on the variable
%                             If none set then defaults to 1
%    If equal to 0 the problem does not contain any of the above types
    of
%    bounds.
%
% Thomas Reslow Krüth, s021898
%------------------------------------------------------------------

if(exist(filename,'file')~=2)
    disp('File does not exist');
    qpProblem = [];
    return
end

qpD = ReadQPSdimensions(filename);
m = qpD.m;
n = qpD.n;
nz = qpD.nz;
qnz = qpD.qnz;

fid=fopen(filename,'r');

% Assume first line is NAME
strLine = fgetl(fid);
qpName = strLine(15:end);

% Read ROWS
strLine = fgetl(fid);
qpsKeyword = 0;
i=0;
j=1;
h_rows = java.util.Hashtable;
strLine = fgetl(fid);
while (~qpsKeyword)
    if(qpsKeyword==0)
        i=i+1;
        cN = strLine(5:end);
        if(length(cN)==8)
            cN = cN;
        elseif(length(cN)==7)
            cN = [cN,' '];
        elseif(length(cN)==6)
            cN = [cN,'  '];
        elseif(length(cN)==5)
            cN = [cN,'   '];
        elseif(length(cN)==4)
            cN = [cN,'    '];
        elseif(length(cN)==3)
            cN = [cN,'     '];
        elseif(length(cN)==2)
            cN = [cN,'      '];
        elseif(length(cN)==1)
            cN = [cN,'       '];
        end
    end
    if(strcmp(strLine(2:3),' N')==1 || strcmp(strLine(2:3),'N ')==1)
        constraintType(i,1:2) = strLine(2:3);
        constraintName(i,1:8) = cN;
        objfuncN = cN;
```

```
    else
        constraintType(i,1:2) = strLine(2:3);
        constraintName(i,1:8) = cN;
        h_rows.put(cN,j);
        j=j+1;
    end
    strLine = fgetl(fid);
    qpsKeyword = strcmp(strLine(1:6),'COLUMN');
end
[mcN,ncN] = size(constraintName);

% Read COLUMNS
A_raw = spalloc(m,n,nz);
g_vec = zeros(n,1);
qpsKeyword = 0;
j=0;
h_cols = java.util.Hashtable;
str_old = 'C------0';
strLine = fgetl(fid);
while(~qpsKeyword)
    str_new = strLine(5:12);
    if(strcmp(str_old,str_new)==0)
        j=j+1;
        h_cols.put(strLine(5:12),j);
    end
    %Case: C------1 OBJ.FUNC double R------2 double
    if(length(strLine)>40 && strcmp(strLine(15:22),objfuncN)==1)
        g_vec(j) = str2double(strLine(25:36));
        k = h_rows.get(strLine(40:47));
        A_raw(k,j) = str2double(strLine(50:end));
        %Case: C------1 R------1 double OBJ.FUNC double
    elseif(length(strLine)>40 && strcmp(strLine(40:47),objfuncN)==1)
        g_vec(j) = str2double(strLine(50:end));
        k = h_rows.get(strLine(15:22));
        A_raw(k,j) = str2double(strLine(25:36));
        %Case: C------1 R------1 double R------2 double
    elseif(length(strLine)>40)
        k = h_rows.get(strLine(15:22));
        l = h_rows.get(strLine(40:47));
        A_raw(k,j) = str2double(strLine(25:36));
        A_raw(l,j) = str2double(strLine(50:end));
        %Case: C------1 OBJ.FUNC double
    elseif(length(strLine)<39 && strcmp(strLine(15:22),objfuncN)==1)
        g_vec(j) = str2double(strLine(25:end));
        %Case: C------1 R------1 double
    elseif(length(strLine)<39)
        k = h_rows.get(strLine(15:22));
        A_raw(k,j) = str2double(strLine(25:end));
    end
    str_old = strLine(5:12);
    strLine = fgetl(fid);
    qpsKeyword = strcmp(strLine(1:3),'RHS');
end

% Read RHS
rhs_raw = zeros(mcN-1,1);
g0 = 0;
strLine = fgetl(fid);
qpsKeyword1 = strcmp(strLine(1:6),'RANGES');
qpsKeyword2 = strcmp(strLine(1:6),'BOUNDS');
qpsKeyword3 = strcmp(strLine(1:6),'QUADOB');
while(~qpsKeyword1 && ~qpsKeyword2 && ~qpsKeyword3)
    %Case: RHS OBJ.FUNC double
    if(length(strLine)<39 && strcmp(strLine(15:22),objfuncN)==1)
        g0 = -str2double(strLine(25:end));
```

```
        %Case: RHS R------1 double
    elseif(length(strLine)<39)
        k = h_rows.get(strLine(15:22));
        rhs_raw(k) = str2double(strLine(25:end));
        %Case: RHS R------1 double R------2 double
    elseif(length(strLine)>40)
        k = h_rows.get(strLine(15:22));
        l = h_rows.get(strLine(40:47));
        rhs_raw(k) = str2double(strLine(25:36));
        rhs_raw(l) = str2double(strLine(50:end));
    end
    strLine = fgetl(fid);
    qpsKeyword1 = strcmp(strLine(1:6),'RANGES');
    qpsKeyword2 = strcmp(strLine(1:6),'BOUNDS');
    qpsKeyword3 = strcmp(strLine(1:6),'QUADOB');
end

if(qpsKeyword1) % RANGES and BOUNDS present
    % Read RANGES
    strLine = fgetl(fid);
    qpsKeyword = strcmp(strLine(1:6),'BOUNDS');
    if(qpsKeyword)
        ranges_raw = [];
        idx_rangesraw = [];
    end
    j=1;
    while(~qpsKeyword)
        if(length(strLine)<39)
            k = h_rows.get(strLine(15:22));
            idx_rangesraw(j) = k-1;
            ranges_raw(k-1) = str2double(strLine(25:end));
            j=j+1;
        elseif(length(strLine)>39)
            k = h_rows.get(strLine(15:22));
            l = h_rows.get(strLine(40:47));
            idx_rangesraw(j)=k-1;
            idx_rangesraw(j+1)=l-1;
            rhs_raw(j) = str2double(strLine(25:36));
            rhs_raw(j+1) = str2double(strLine(50:end));
            j=j+2;
        end
        strLine = fgetl(fid);
        qpsKeyword = strcmp(strLine(1:6),'BOUNDS');
    end
    % Read BOUNDS
    str_old = 'C------0';
    strLine = fgetl(fid);
    qpsKeyword = strcmp(strLine(1:7),'QUADOBJ');
    lo_vec = zeros(n,1);
    up_vec = inf*ones(n,1);
    bv_li_ui_sc = 0;
    i = 0;
    while(~qpsKeyword)
        str_new = strLine(15:22);
        if(strcmp(str_old,str_new)==0)
            i=i+1;
        end
        j = h_cols.get(str_new);
        if(strcmp(strLine(2:3),'LO')==1)
            lo_vec(j) = str2double(strLine(25:end));
            up_vec(j) = inf;
        elseif(strcmp(strLine(2:3),'UP')==1)
            if(up_vec(i)==inf)
                up_vec(j) = str2double(strLine(25:end));
            else
```

```
                lo_vec(j) = 0;
                up_vec(j) = str2double(strLine(25:end));
            end
        elseif(strcmp(strLine(2:3),'FX')==1)
            lo_vec(j) = str2double(strLine(25:end));
            up_vec(j) = str2double(strLine(25:end));
        elseif(strcmp(strLine(2:3),'FR')==1)
            lo_vec(j) = -inf;
            up_vec(j) = inf;
        elseif(strcmp(strLine(2:3),'MI')==1)
            lo_vec(j) = -inf;
            up_vec(j) = 0;
        elseif(strcmp(strLine(2:3),'PL')==1)
            lo_vec(j) = 0;
            up_vec(j) = inf;
        elseif(strcmp(strLine(2:3),'BV')==1)
            bv_li_ui_sc = 1;
        elseif(strcmp(strLine(2:3),'LI')==1)
            bv_li_ui_sc = 1;
        elseif(strcmp(strLine(2:3),'UI')==1)
            bv_li_ui_sc = 1;
        elseif(strcmp(strLine(2:3),'SC')==1)
            bv_li_ui_sc = 1;
        end
        str_old = strLine(15:22);
        strLine = fgetl(fid);
        qpsKeyword = strcmp(strLine(1:7),'QUADOBJ');
    end

    % Read QUADOBJ
    Q = spalloc(n,n,n+qnz);
    strLine = fgetl(fid);
    qpsKeyword = strcmp(strLine(1:6),'ENDATA');
    while(~qpsKeyword)
        if(length(strLine)<39)
            j = h_cols.get(strLine(5:12));
            i = h_cols.get(strLine(15:22));
            Q(i,j) = str2double(strLine(25:end));
        elseif(length(strLine)>40)
            j = h_cols.get(strLine(5:12));
            i1 = h_cols.get(strLine(15:22));
            i2 = h_cols.get(strLine(40:47));
            Q(i1,j) = str2double(strLine(25:36));
            Q(i2,j) = str2double(strLine(50:end));
        end
        strLine = fgetl(fid);
        qpsKeyword = strcmp(strLine(1:6),'ENDATA');
    end
    Q_tmp = Q'-diag(diag(Q));
    G = Q + Q_tmp;


elseif(qpsKeyword2) % BOUNDS present
    ranges_raw = [];
    idx_rangesraw = [];
    % Read BOUNDS
    str_old = 'C------0';
    strLine = fgetl(fid);
    qpsKeyword = strcmp(strLine(1:7),'QUADOBJ');
    lo_vec = zeros(n,1);
    up_vec = inf*ones(n,1);
    bv_li_ui_sc = 0;
    i = 0;
    while(~qpsKeyword)
        str_new = strLine(15:22);
```

```
        if(strcmp(str_old,str_new)==0)
            i=i+1;
        end
        j = h_cols.get(str_new);
        if(strcmp(strLine(2:3),'LO')==1)
            lo_vec(j) = str2double(strLine(25:end));
            up_vec(j) = inf;
        elseif(strcmp(strLine(2:3),'UP')==1)
            if(up_vec(i)==inf)
                up_vec(j) = str2double(strLine(25:end));
            else
                lo_vec(j) = 0;
                up_vec(j) = str2double(strLine(25:end));
            end
        elseif(strcmp(strLine(2:3),'FX')==1)
            lo_vec(j) = str2double(strLine(25:end));
            up_vec(j) = str2double(strLine(25:end));
        elseif(strcmp(strLine(2:3),'FR')==1)
            lo_vec(j) = -inf;
            up_vec(j) = inf;
        elseif(strcmp(strLine(2:3),'MI')==1)
            lo_vec(j) = -inf;
            up_vec(j) = 0;
        elseif(strcmp(strLine(2:3),'PL')==1)
            lo_vec(j) = 0;
            up_vec(j) = inf;
        elseif(strcmp(strLine(2:3),'BV')==1)
            bv_li_ui_sc = 1;
        elseif(strcmp(strLine(2:3),'LI')==1)
            bv_li_ui_sc = 1;
        elseif(strcmp(strLine(2:3),'UI')==1)
            bv_li_ui_sc = 1;
        elseif(strcmp(strLine(2:3),'SC')==1)
            bv_li_ui_sc = 1;
        end
        str_old = strLine(15:22);
        strLine = fgetl(fid);
        qpsKeyword = strcmp(strLine(1:7),'QUADOBJ');
    end

    % Read QUADOBJ
    Q = spalloc(n,n,n+qnz);
    strLine = fgetl(fid);
    qpsKeyword = strcmp(strLine(1:6),'ENDATA');
    while(~qpsKeyword)
        if(length(strLine)<39)
            j = h_cols.get(strLine(5:12));
            i = h_cols.get(strLine(15:22));
            Q(i,j) = str2double(strLine(25:end));
        elseif(length(strLine)>40)
            j = h_cols.get(strLine(5:12));
            i1 = h_cols.get(strLine(15:22));
            i2 = h_cols.get(strLine(40:47));
            Q(i1,j) = str2double(strLine(25:36));
            Q(i2,j) = str2double(strLine(50:end));
        end
        strLine = fgetl(fid);
        qpsKeyword = strcmp(strLine(1:6),'ENDATA');
    end
    Q_tmp = Q'-diag(diag(Q));
    G = Q + Q_tmp;

elseif(qpsKeyword3) % RANGES and BOUNDS not present
    ranges_raw = [];
    idx_rangesraw = [];
```

```
    lo_vec = zeros(n,1);
    up_vec = inf*ones(n,1);
    bv_li_ui_sc = 0;
    % Read QUADOBJ
    Q = spalloc(n,n,n+qnz);
    strLine = fgetl(fid);
    qpsKeyword = strcmp(strLine(1:6),'ENDATA');
    while(~qpsKeyword)
        if(length(strLine)<39)
            j = h_cols.get(strLine(5:12));
            i = h_cols.get(strLine(15:22));
            Q(i,j) = str2double(strLine(25:end));
        elseif(length(strLine)>40)
            j = h_cols.get(strLine(5:12));
            i1 = h_cols.get(strLine(15:22));
            i2 = h_cols.get(strLine(40:47));
            Q(i1,j) = str2double(strLine(25:36));
            Q(i2,j) = str2double(strLine(50:end));
        end
        strLine = fgetl(fid);
        qpsKeyword = strcmp(strLine(1:6),'ENDATA');
    end
    Q_tmp = Q'-diag(diag(Q));
    G = Q + Q_tmp;
end

fclose(fid);

qpProblem = struct(...
                'Name', qpName,...
                'A_raw', A_raw,...
                'rhs_raw', rhs_raw,...
                'constraintType', constraintType,...
                'lo_vec', lo_vec,...
                'up_vec', up_vec,...
                'ranges_raw', ranges_raw,...
                'idx_rangesraw', idx_rangesraw,...
                'G', sparse(G),...
                'g_vec', g_vec,...
                'g0', g0,...
                'bv_li_ui_sc', bv_li_ui_sc,...
                'm', m,...
                'n', n,...
                'nz', nz);
%-------------------------------------------------------------------



function qpDim = ReadQPSdimensions(filename)
%-------------------------------------------------------------------
% ReadQPSdimensions.m
%
% This function scans a QPS file for the sizes of the QP problem and
    gives
% the following output.
%
%    m: number of constraints of the form
%                        A(i,:)*x >= rhs(i) OR
%                        A(i,:)*x <= rhs(i) OR
%                        A(i,:)*x  = rhs(i)
%
%    n: number of variables
%
%    nz: the number of nonzeros in A
%
```

```
%    qnz: the number of off-diagonal entries in the lower triangular
     part
%    of Q
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------------

fid=fopen(filename,'r');

% Read m
strLine = fgetl(fid);
qpName = strLine(15:end);
strLine = fgetl(fid);
qpsKeyword = 0;
i=0;
while (~qpsKeyword)
    strLine = fgetl(fid);
    qpsKeyword = strcmp(strLine(1:6),'COLUMN');
    if(qpsKeyword==0)
        i=i+1;
    end
end
m = i-1;

fclose(fid);
fid=fopen(filename,'r');
strLine = fgetl(fid);
qpName = strLine(15:end);
strLine = fgetl(fid);
qpsKeyword = 0;
constraintName = zeros(m+1,8);
i=0;
while (~qpsKeyword)
    strLine = fgetl(fid);
    qpsKeyword = strcmp(strLine(1:6),'COLUMN');
    if(qpsKeyword==0)
        i=i+1;
        cN = strLine(5:end);
        if(length(cN)==8)
            cN = cN;
        elseif(length(cN)==7)
            cN = [cN,' '];
        elseif(length(cN)==6)
            cN = [cN,'  '];
        elseif(length(cN)==5)
            cN = [cN,'   '];
        elseif(length(cN)==4)
            cN = [cN,'    '];
        elseif(length(cN)==3)
            cN = [cN,'     '];
        elseif(length(cN)==2)
            cN = [cN,'      '];
        elseif(length(cN)==1)
            cN = [cN,'       '];
        end
    end
    constraintName(i,1:8) = cN;
end

% Read n and nz
qpsKeyword = 0;
n=0;
nz=0;
str_old = 'C------0';
strLine = fgetl(fid);
```

```
while(~qpsKeyword)
    str_new = strLine(5:12);
    if(strcmp(str_old,str_new)==0)
        n=n+1;
    end
    %Case: C------1 OBJ.FUNC double R------2 double
    if(length(strLine)>40 && strcmp(strLine(15:22),constraintName(1,1:8)
        )==1)
        nz = nz + 1;
    %Case: C------1 R------1 double R------2 double
    elseif(length(strLine)>40)
        nz = nz + 2;
    %Case: C------1 OBJ.FUNC double
    elseif(length(strLine)<39 && strcmp(strLine(15:22),constraintName
        (1,1:8))==1)
        nz = nz;
    %Case: C------1 R------1 double
    elseif(length(strLine)<39)
        nz = nz + 1;
    end
    str_old = strLine(5:12);
    strLine = fgetl(fid);
    qpsKeyword = strcmp(strLine(1:3),'RHS');
end

% Read qnz
strLine = fgetl(fid);
qpsKeyword = strcmp(strLine(1:6),'QUADOB');
while(~qpsKeyword)
    strLine = fgetl(fid);
    qpsKeyword = strcmp(strLine(1:6),'QUADOB');
end

strLine = fgetl(fid);
qpsKeyword = strcmp(strLine(1:6),'ENDATA');
qnz=0;
while(~qpsKeyword)
    strLine = fgetl(fid);
    if(length(strLine)<39)
        qnz = qnz + 1;
    elseif(length(strLine)>40)
        qnz = qnz + 2;
    end
    qpsKeyword = strcmp(strLine(1:6),'ENDATA');
end
%qnz = qnz - n;

fclose(fid);

qpDim = struct(...
                'm', m,...
                'n', n,...
                'nz', nz,...
                'qnz', qnz);
```

Listing B.11: convertFormat.m

```
function qpStandard = convertFormat(A_raw,rhs_raw,ctype,lo_vec,up_vec
    ,...
                                    ranges_raw,idx_ranges,n,nz)
%-------------------------------------------------------------------
% convertFormat.m
%
% This function converts output from QPSreader.m to the form
```

```
%
%                           min 0.5x'Gx + g'x
%                           s.t A'x  = b
%                               C'x >= d
% or
%                           min 0.5x'Gx + g'x
%                           s.t C'x >= d
%
% Input:
%     A_raw: constraint matrix
%     rhs_raw: right hand side for constraint matrix
%     ctype: vector with constraint types
%     lo_vec: lower bounds on x
%     up_vec: upper bounds on x
%     ranges_raw: ranges-entries
%     idx_ranges: indexes for ranges
%     n: number of variables
%     nz: number of nonzeros in A
%
% Output:
%     A: matrix containing equality constraints
%     b: right hand side of equality constraints
%     C: matrix containing inequality constraints
%     d: right hand side of inequality constraints
%     z0: starting point for z
%     s0: starting point for s
%     y0: starting point for y
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------

% To do: ranges conversion

numEBounds = 0;
numGBounds = 0;
numLBounds = 0;
numNBounds = 0;

% Form A, b and possibly parts of C and d
for i=1:length(ctype)
    cT = ctype(i,1:2);
    if(strcmp(cT,' E')==1 || strcmp(cT,'E ')==1)
        numEBounds = numEBounds + 1;
    elseif(strcmp(cT,' G')==1 || strcmp(cT,'G ')==1)
        numGBounds = numGBounds + 1;
    elseif(strcmp(cT,' L')==1 || strcmp(cT,'L ')==1)
        numLBounds = numLBounds + 1;
    elseif(strcmp(cT,' N')==1 || strcmp(cT,'N ')==1)
        numNBounds = numNBounds + 1;
    end
end

A = spalloc(numEBounds,n,nz);
b = zeros(1,numEBounds);
C_G = spalloc(numGBounds,n,nz);
d_G = zeros(1,numGBounds);
C_L = spalloc(numLBounds,n,nz);
d_L = zeros(1,numLBounds);
numEBounds = 0;
numGBounds = 0;
numLBounds = 0;
numNBounds = 0;
for i=2:length(ctype)
    cT = ctype(i,1:2);
    if(strcmp(cT,' E')==1 || strcmp(cT,'E ')==1)
```

```
        numEBounds = numEBounds + 1;
        A(numEBounds,:) = A_raw(i-1,:);
        b(numEBounds) = rhs_raw(i-1);
    elseif(strcmp(cT,' G')==1 || strcmp(cT,'G ')==1)
        numGBounds = numGBounds + 1;
        C_G(numGBounds,:) = A_raw(i-1,:);
        d_G(numGBounds) = rhs_raw(i-1);
    elseif(strcmp(cT,' L')==1 || strcmp(cT,'L ')==1)
        numLBounds = numLBounds + 1;
        C_L(numLBounds,:) = -A_raw(i-1,:);
        d_L(numLBounds) = -rhs_raw(i-1);
    elseif(strcmp(cT,' N')==1 || strcmp(cT,'N ')==1)
        numNBounds = numNBounds + 1;
    end
end

if(isempty(b))
    y = [];
else
    [mA,nA] = size(A);
    y = ones(mA,1);
end
C_col = [C_G;C_L];
d_col = [d_G,d_L]';

% Form C
% remove Inf entries
idx_lotmp = find(lo_vec==-Inf);
idx_uptmp = find(up_vec==Inf);
if(isempty(idx_lotmp))
    idx_lotmp = [];
end
if(isempty(idx_uptmp))
    idx_uptmp = [];
end
lo_vec(idx_lotmp) = [];
up_vec(idx_uptmp) = [];

if(isempty(lo_vec) && isempty(up_vec) && isempty(d_col))
    C = [];
    d = [];
    z = [];
    s = [];
elseif(isempty(lo_vec) && isempty(up_vec) && ~isempty(d_col))
    C = C_col;
    d = d_col;
    [mC,nC] = size(C);
    z = ones(mC,1);
    s = ones(mC,1);
elseif(~isempty(lo_vec) && isempty(up_vec))
    C_tmp = speye(length(lo_vec),n);
    d_tmp = lo_vec;
    if(~isempty(d_col))
        C = [C_col;C_tmp];
        d = [d_col;d_tmp];
    else
        C = C_tmp;
        d = d_tmp;
    end
    [mC,nC] = size(C);
    z = ones(mC,1);
    s = ones(mC,1);
elseif(~isempty(lo_vec) && ~isempty(up_vec))
    C_tmp = [speye(length(lo_vec),n);-speye(length(up_vec),n)];
    d_tmp = [lo_vec;-up_vec];
```

```
    if(~isempty(d_col))
        C = [C_col;C_tmp];
        d = [d_col;d_tmp];
    else
        C = C_tmp;
        d = d_tmp;
    end
    [mC,nC] = size(C);
    z = ones(mC,1);
    s = ones(mC,1);
end

qpStandard = struct(...
                'A', A',...
                'b', b',...
                'C', C',...
                'd', d,...
                'z0', z,...
                's0', s,...
                'y0', y);
```

B.5 Script files

Listing B.12: test1.m

```
%-------------------------------------------------------------
% test1.m
%
% This script generates results used in chapter 2.
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------
clear all;
close all;
clc;

n = 10;
G = eye(n,n);
seed = 100;
randn('state',seed);
g = randn(n,1);
A = [eye(n,n) -eye(n,n)];
b = -10*ones(2*n,1);
x = zeros(n,1);
z = ones(2*n,1);
s = ones(2*n,1);

[x_stop,z_stop,s_stop,k] = pcQP_simplev1(x,z,s,G,g,A,b);

x_stop + g
k
```

Listing B.13: test2.m

```
%-------------------------------------------------------------
% test2.m
%
% This script generates results used in chapter 5.
```

```
%
% Thomas Reslow Krüth, s021898
%------------------------------------------------------------------
clear all;
close all;
clc;

n = 10;
G = eye(n,n);
seed = 100;
randn('state',seed);
g = randn(n,1);
A = [eye(n,n) -eye(n,n)];
b = -10*ones(2*n,1);
x = zeros(n,1);
z = ones(2*n,1);
s = ones(2*n,1);

[x_stop,z_stop,s_stop,k] = mccQP_simplev1(x,z,s,G,g,A,b);

x_stop + g
k
```

Listing B.14: plotnvsiter.m

```
%------------------------------------------------------------------
% plotnvsiter.m
%
% This script generates a plot used to compare the number of iterations
% used by the PC method and the MCC method.
%
% Thomas Reslow Krüth, s021898
%------------------------------------------------------------------
clear all;
close all;
clc;

n_max = 1000;
n_plot = (10:20:n_max);
k_vec1 = zeros(1,length(n_plot));
seed=1000;
randn('state',seed);
i=0;
for n=10:20:n_max
    i=i+1;
    G = eye(n,n);
    g = randn(n,1);
    A = [eye(n,n) -eye(n,n)];
    b = -10*ones(2*n,1);
    x = zeros(n,1);
    z = ones(2*n,1);
    s = ones(2*n,1);
    [x_stop,z_stop,s_stop,k] = pcQP_simplev1(x,z,s,G,g,A,b);
    k_vec1(i) = k;
end

figure
plot(n_plot,k_vec1,'o-b','LineWidth',2);
xlabel('number of variables n');
ylabel('number of iterations');
title('Number of iterations as a function of n');
hold on

k_vec2 = zeros(1,length(n_plot));
```

```
i=0;
for n=10:20:n_max
    i=i+1;
    G = eye(n,n);
    g = randn(n,1);
    A = [eye(n,n) -eye(n,n)];
    b = -10*ones(2*n,1);
    x = zeros(n,1);
    z = ones(2*n,1);
    s = ones(2*n,1);
    [x_stop,z_stop,s_stop,k] = mccQP_simplev1(x,z,s,G,g,A,b);
    k_vec2(i) = k;
end

plot(n_plot,k_vec2,'x-k','LineWidth',2);
legend('pc','mcc');
axis([0 1000 5 20]);
grid on
hold off
```

Listing B.15: plotnvstime.m

```
%-------------------------------------------------------------
% plotnvstime.m
%
% This script generates plots used to compare the computation time
% used by the PC method and the MCC method.
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------
clear all;
close all;
clc;

n_max = 1000;
n_plot = (10:20:n_max);
time_n1 = zeros(1,length(n_plot));
seed=1000;
randn('state',seed);
i=0;
for n=10:20:n_max
    i=i+1;
    G = eye(n,n);
    g = randn(n,1);
    A = [eye(n,n) -eye(n,n)];
    b = -10*ones(2*n,1);
    x = zeros(n,1);
    z = ones(2*n,1);
    s = ones(2*n,1);
    tic
    [x_stop,z_stop,s_stop,k] = pcQP_simplev1(x,z,s,G,g,A,b);
    time = toc;
    time_n1(i) = time;
end
figure
plot(n_plot,time_n1,'o-b', 'LineWidth', 2);
xlabel('number of variables n');
ylabel('time (seconds)');
title('Time as a function of n');
hold on

time_n2 = zeros(1,length(n_plot));
i=0;
for n=10:20:n_max
```

```
    i=i+1;
    G = eye(n,n);
    g = randn(n,1);
    A = [eye(n,n) -eye(n,n)];
    b = -10*ones(2*n,1);
    x = zeros(n,1);
    z = ones(2*n,1);
    s = ones(2*n,1);
    tic
    [x_stop,z_stop,s_stop,k] = mccQP_simplev1(x,z,s,G,g,A,b);
    time = toc;
    time_n2(i) = time;
end
plot(n_plot,time_n2,'x-k', 'LineWidth', 2);
legend('pc','mcc');
grid on
hold off

sav = time_n1-time_n2;
figure
plot(n_plot,sav,'b','LineWidth',2);
xlabel('number of variables n');
ylabel('savings (seconds)');
grid on
title('Savings as a function of n');
```

Listing B.16: plotsigmavsmu.m

```
%-------------------------------------------------------------
% plotsigmavsmu.m
%
% This script generates a plot used in chapter 2 to visualize the
    centering
% parameter.
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------

clear all;
close all;
clc;

psi = 3;
mu_a = (0.1:0.1:5);
mu = 5;

sigma = (mu_a./mu).^psi;
plot(mu_a,sigma,'x-b','LineWidth',2)
grid on;
xlabel('muaff','FontSize',11);
ylabel('sigma','FontSize',11);
title('sigma=(muafff/mu)^3, mu=5','FontSize',11);
```

Listing B.17: introExample.m

```
%-------------------------------------------------------------
% introExample.m
%
% This script generates the plot used in the Introduction.
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------
clear all;
```

```
close all;
clc;

n = 2;
G = [8,2;2,2];
g = [2;3];
A = [1,-1,-1;-1,-1,0];
b = [0;-4;-3];

x0 = zeros(n,1);
z0 = ones(3,1);
s0 = ones(3,1);

[x_stop,z_stop,s_stop,k] = pcQP_simplev1(x0,z0,s0,G,g,A,b);

%Plot of problem
x1 = (-4:0.01:4);
x2 = (-4:0.01:4);

[X1,X2] = meshgrid(x1,x2);
F = 2*X1+3*X2+4*X1.^2+2*X1.*X2+X2.^2;

v = [0:2:10 10:10:100 100:20:200];
[c,h] = contour(X1,X2,F,v);
colorbar;

x2c1 = x1;
x2c2 = -x1+4;

hold on
grid on
fill([x1 x1(end) x1(1)],[x2c1 4 4],[0.7 0.7 0.7],'facealpha',0.5);
fill([x1 4 x1(end)],[x2c2 4 4],[0.7 0.7 0.7],'facealpha',0.5);
fill([3 x1(end) x1(end) 3],[x2(1) x2(1) x2(end) x2(end)],[0.7 0.7 0.7],'
    facealpha',0.5);
plot(x_stop(1),x_stop(2),'x','LineWidth',2);
xlabel('x_1','FontSize',11);
ylabel('x_2','FontSize',11);
title('Contourplot of testproblem and constraints','FontSize',11);
hold off
```

Listing B.18: testNewSteps.m

```
%-------------------------------------------------------------
% testNewSteps.m
%
% This script tests the new step length strategy.
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------
clear all;
clc;
%----------------------------
% test_prob = 'lotschd.qps';
% test_prob = 'qscsd1.qps';
% test_prob = 'qscsd6.qps';
% test_prob = 'qscsd8.qps';
%----------------------------

tic
qpProb = reader(test_prob);
readertime = toc;

tic
```

```
qpSta = convertFormat(qpProb.A_raw,qpProb.rhs_raw,qpProb.constraintType
    ,...
                          qpProb.lo_vec,qpProb.up_vec,...
                          qpProb.ranges_raw,qpProb.idx_rangesraw,...
                          qpProb.n,qpProb.nz);
converttime = toc;

[mA,nA] = size(qpSta.A);
x0 = 0.5*ones(qpProb.n,1);
y0 = ones(nA,1);
z0 = ones(qpProb.n,1);

tic
[x_stop,y_stop,z_stop,k] = pcQP_simplev8(x0,y0,z0,...
    qpProb.G,qpProb.g_vec,qpSta.A',qpSta.b);
cputime = toc
k
f = 0.5*x_stop'*qpProb.G*x_stop + qpProb.g_vec'*x_stop + qpProb.g0
```

Listing B.19: demo_v1.m

```
%-------------------------------------------------------------
% demo_v1.m
%
% This scriptfile demonstrates the use of
% QPSreader.m
% convertFormat.m
% and
% the implemented interior-point methods.
% 38 test problems are available. Run script by uncommenting 1 of the
    test
% problems.
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------

clear all;
clc;

%-----------------------------
% test_prob = 'aug3dcqp.qps';
% test_prob = 'aug3dqp.qps';
% test_prob = 'cont-050.qps';
% test_prob = 'cvxqp1_m.qps';
% test_prob = 'cvxqp1_s.qps';
% test_prob = 'cvxqp2_m.qps';
% test_prob = 'cvxqp2_s.qps';
% test_prob = 'cvxqp3_m.qps';
% test_prob = 'cvxqp3_s.qps';
% test_prob = 'dual1.qps';
% test_prob = 'dual2.qps';
% test_prob = 'dual3.qps';
% test_prob = 'dual4.qps';
% test_prob = 'gouldqp2.qps';
% test_prob = 'gouldqp3.qps';
% test_prob = 'hs21.qps';
% test_prob = 'hs35.qps';
% test_prob = 'hs53.qps';
% test_prob = 'hs76.qps';
% test_prob = 'lotschd.qps';
% test_prob = 'mosarqp1.qps';
% test_prob = 'mosarqp2.qps';
% test_prob = 'qpcblend.qps';
% test_prob = 'qptest.qps';
```

```
% test_prob = 'qscorpio.qps';
% test_prob = 'qscrs8.qps';
% test_prob = 'qscsd1.qps';
% test_prob = 'qscsd6.qps';
% test_prob = 'qscsd8.qps';
% test_prob = 'qsctap1.qps';
% test_prob = 'qsctap2.qps';
% test_prob = 'qsctap3.qps';
% test_prob = 'qshare2b.qps';
% test_prob = 'stcqp1.qps';
% test_prob = 'stcqp2.qps';
% test_prob = 'tame.qps';
% test_prob = 'zecevic2.qps';
% test_prob = 'values.qps';
%--------------------------

tic
qpProb = QPSreader(test_prob);
readertime = toc;

tic
qpStand = convertFormat(qpProb.A_raw,qpProb.rhs_raw,qpProb.
    constraintType,...
                            qpProb.lo_vec,qpProb.up_vec,...
                            qpProb.ranges_raw,qpProb.idx_rangesraw,...
                            qpProb.n,qpProb.nz);
converttime = toc;

%Starting guess x
x = 0.5*ones(qpProb.n,1);

if(isempty(qpStand.A))
    tic
    [x_stop,z_stop,s_stop,k] = mccQP_simplev1(x,qpStand.z0,qpStand.s0
        ,...
        qpProb.G,qpProb.g_vec,qpStand.C,qpStand.d);
    cputime = toc;
else
    tic
    [x_stop,y_stop,z_stop,s_stop,k] = pcQP_gen(x,qpStand.y0,qpStand.z0,
        qpStand.s0,...
        qpProb.G,qpProb.g_vec,qpStand.C,qpStand.d,qpStand.A,qpStand.b);
    cputime = toc;
end
k
cputime
f = 0.5*x_stop'*qpProb.G*x_stop + qpProb.g_vec'*x_stop + qpProb.g0
```

Listing B.20: demo_v2.m

```
%-------------------------------------------------------------
% demo_v2.m
%
% This scriptfile demonstrates the use of
% QPSreader.m
% convertFormat.m
% and
% the implemented interior-point methods.
%
% Thomas Reslow Krüth, s021898
%-------------------------------------------------------------
clear all;
close all;
clc;
```

```
%-----------------------------
test_prob = 'qptest.qps';
%-----------------------------

tic
qpProb = QPSreader(test_prob);
readertime = toc;

tic
qpStand = convertFormat(qpProb.A_raw,qpProb.rhs_raw,...
                        qpProb.constraintType,...
                        qpProb.lo_vec,qpProb.up_vec,...
                        qpProb.ranges_raw,qpProb.idx_rangesraw,...
                        qpProb.n,qpProb.nz);
converttime = toc;

%Starting guess x
x = 0.5*ones(qpProb.n,1);

if(isempty(qpStand.A))
    [x_stoppc,z_stoppc,s_stoppc,kpc,x_itspc,a_itspc] =...
        pcQP_simplev11(x,qpStand.z0,qpStand.s0,...
        qpProb.G,qpProb.g_vec,qpStand.C,qpStand.d);
    [x_stop,z_stop,s_stop,k,x_its,a_its] =...
        mccQP_simplev11(x,qpStand.z0,qpStand.s0,...
        qpProb.G,qpProb.g_vec,qpStand.C,qpStand.d);
else
    tic
    [x_stop,y_stop,z_stop,s_stop,k] = mccQP_genv1(x,qpStand.y0,...
        qpStand.z0,qpStand.s0,...
        qpProb.G,qpProb.g_vec,qpStand.C,qpStand.d,qpStand.A,qpStand.b);
    cputime = toc;
end
f = 0.5*x_stop'*qpProb.G*x_stop + qpProb.g_vec'*x_stop + qpProb.g0;

x1 = (0:0.005:1.8);
x2 = (0:0.005:1.8);

[X1,X2] = meshgrid(x1,x2);
F = 4 + 1.5*X1 - 2*X2 + 0.5*(8*X1.^2+2*X1.*X2+2*X2.*X1+10*X2.^2);

v = [0:2:10 10:10:100 100:20:200];
[c,h] = contour(X1,X2,F,v);
colorbar;

x2c1 = 2-2*x1;
x2c2 = 0.5*x1+3;
hold on
grid on
fill([x1 x1(end) x1(1)],[x2c1 4 4],[0.7 0.7 0.7],'facealpha',0.5);
plot(x_itspc(:,1),x_itspc(:,2),'x-','LineWidth',1.5);
plot(x_its(:,1),x_its(:,2),'x-r','LineWidth',1.5);
legend('f','c1','PC','MCC');
xlabel('x','FontSize',11);
ylabel('y','FontSize',11);
title('Contourplot of testproblem and constraints','FontSize',11);
hold off
```

Bibliography

[1] Jorge Nocedal and Stephen Wright (2000)
Numerical Optimization Second Edition

[2] Stephen Wright (1997)
Primal-Dual Interior-Point Methods

[3] John B. Jørgensen (November 2006) Quadratic Optimization, Primal-Dual Interior-Point Algorithm
Notes for course 02611 Optimization Algorithms and Data-Fitting

[4] Istvan Maros and Csaba Meszaros, Departmental Technical Report DOC 97/6, Department of Computing, Imperial College, London, U.K. (1 July, 1997)
A Repository of Convex Quadratic Programming Problems

[5] http://lpsolve.sourceforge.net/5.5/mps-format.htm
MPS file format

[6] Frank Curtis and Jorge Nocedal (December 20, 2005)
Steplength Selection in Interior-Point Methods for Quadratic Programming

[7] Jacek Gondzio, Computational Optimization and Applications 6, 137-156 (1996)
Multiple Centrality Corrections in a Primal-Dual Method for Linear Programming

[8] Marco Colombo and Jacek Gondzio (MS-2005-001)
Further Development of Multiple Centrality Correctors for Interior Point Methods