Numerical Algorithms for Sequential Quadratic Optimization

Esben Lundsager Hansen s022022
Carsten Völcker s961572

Kongens Lyngby 2007
IMM-M.Sc-2007-70

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

Abstract

This thesis investigates numerical algorithms for sequential quadratic programming (SQP). SQP algorithms are used for solving nonlinear programs, i.e. mathematical optimization problems with nonlinear constraints.

SQP solves the nonlinear constrained program by solving a sequence of associating quadratic programs (QP's). A QP is a constrained optimization problem in which the objective function is quadratic and the constraints are linear. The QP is solved by use of the primal active set method or the dual active set method. The primal active set method solves a convex QP where the Hessian matrix is positive semi definite. The dual active set method requires the QP to be strictly convex, which means that the Hessian matrix must be positive definite. The active set methods solve an inequality constrained QP by solving a sequence of corresponding equality constrained QP's.

The equality constrained QP is solved by solving an indefinite symmetric linear system of equations, the so-called Karush-Kuhn-Tucker (KKT) system. When solving the KKT system, the range space procedure or the null space procedure is used. These procedures use Cholesky and QR factorizations. The range space procedure requires the Hessian matrix to be positive definite, while the null space procedure only requires it to be positive semi-definite.

By use of Givens rotations, complete factorization is avoided at each iteration of the active set methods. The constraints are divided into bounded variables and general constraints. If a bound becomes active the bounded variable is fixed, otherwise it is free. This is exploited for further optimization of the factorizations.

The algorithms has been implemented in Matlab and tested on strictly convex QP's of sizes up to 1800 variables and 7200 constraints. The testcase is the quadruple tank process, described in appendix A.

Main Findings of this Thesis

When the number of active constraints reaches a certain amount compared to the number of variables, the null space procedure should be used. The range space procedure is only prefereble, when the number of active constraints is very small compared to the number of variables.

The update procedures of the factorizations give significant improvement in computational speed.

Whenever the Hessian matrix of the QP is positive definite the dual active set method is prefereble. The calculation of a starting point is implicit in the method and furthermore convergence is guaranteed.

When the Hessian matrix is positive semi definite, the primal active set can be used. For this matter an LP solver should be implemented, which computes a starting point and an active set that makes the reduced Hessian matrix positive definite. This LP solver has not been implemented, as it is out of the range of this thesis.

Dansk Resumé

Dette Projekt omhandler numeriske algoritmer til sekventiel kvadratisk programmering (SQP). SQP benyttes til at løse ikke-lineære programmer, dvs. matematiske optimeringsproblemer med ikke-linære begrænsninger.

SQP løser det ikke-lineært begrænsede program ved at løse en sekvens af tilhørende kvadratiske programmer (QP'er). Et QP er et begrænset optimeringsproblem, hvor objektfunktionen er kvadratisk og begrænsningerne er lineære. Et QP løses ved at bruge primal aktiv set metoden eller dual aktiv set metoden. Primal aktiv set metoden løser et konvekst QP, hvor Hessian matricen er positiv semi definit. Dual aktiv set metoden kræver et strengt konvekst QP, dvs. at Hessian matricen skal være positiv definit. Aktiv set metoderne løser et ulighedsbegrænset QP ved at løse en sekvens af tilhørende lighedsbegænsede QP'er.

Løsningen til det lighedsbegrænsede QP findes ved at løse et indefinit symmetrisk lineært ligningssystem, det såkaldte Karush-Kuhn-Tucker (KKT) system. Til at løse KKT systemet benyttes range space proceduren eller null space proceduren, som bruger Cholesky og QR faktoriseringer. Range space proceduren kræver, at Hessian matricen er positiv definit. Null space proceduren kræver kun, at den er positiv semi definit.

Ved brug af Givens rotationer ungås fuld faktorisering for hver iteration i aktiv set metoderne. Begrænsningerne deles op i begrænsede variable og egentlige begrænsninger beskrevet ved funktionsudtryk. Begrænsede variable betyder, at en andel af variablene er fikserede, mens resten er frie pr. iteration. Dette udnyttes til yderligere optimering af faktoriseringerne mellem hver iteration.

Algoritmerne er implementeret i Matlab og testet på strengt konvekse QP'er

bestående af op til 1800 variable og 7200 begrænsninger. Testeksemplerne er genereret udfra det firdobbelte tank system, som er beskrevet i appendix A.

Hovedresultater

Når antallet af aktive begrænsninger når en vis mængde i forhold til antallet af variable, bør null space proceduren benyttes. Range space proceduren bør kun benyttes, når antallet af aktive begrænsninger er lille i forhold til antallet af variable.

Når fuld faktorisering undgås ved at benytte opdateringer, er der betydelige beregningsmæssige besparelser.

Hvis Hessian matricen af et QP er positiv definit, bør dual aktiv set metoden benyttes. Her foregår beregningerne af startpunkt implicit i metoden, og desuden er konvergens garanteret.

Hvis Hessian matricen er positiv semi definit, kan primal aktiv set metoden benyttes. Men her skal der benyttes en LP-løser til at beregne et startpunkt og et tilhørende aktivt set, som medfører at den reducerede Hessian matrix bliver positiv definit. Denne LP-løser er ikke blevet implementeret, da den ligger udenfor området af dette projekt.

Contents

Chapter 1

Introduction

In optimal control there is a high demand for real-time solutions. Dynamic systems are more or less sensitive to outer influences, and therefore require fast and reliable adjustment of the control parameters.

A dynamic system in equilibrium can experience disturbances explicitly, e.g. sudden changes in the environment in which the system is embedded or online changes to the demands of the desired outcome of the system. Implicit disturbances have also to be taken care of in real-time, e.g. changes of the input needed to run the system. In all cases, fast and reliable optimal control is essential in lowering the running cost of a dynamic system.

Usually the solution of a dynamic process must be kept within certain limits. In order to generate a feasible solution to the process, these limits have to be taken into account. If a process like this can be modeled as a constrained optimization problem, model predictive control can be used in finding a feasible solution, if it exists.

Model predictive control with nonlinear models can be performed using sequential quadratic programming (SQP). Model predictive control with linear models may be conducted using quadratic programming (QP). A variety of different numerical methods exist for both SQP and QP. Some of these methods comprise the subject of this project.

The main challenge in SQP is to solve the QP, and therefore methods for solving QP's constitute a major part of this work. In itself, QP has a variety of applications, e.g. Portfolio Optimization by Markowitz, found in Nocedal and Wright [14], solving constraint least squares problems and in Huber regression Li and Swetits [1]. A QP consists of a quadratic objective function, which we want to minimize subject to a set of linear constraints. A QP is stated as

$$\begin{aligned} \min_{\boldsymbol{x} \in \mathbb{R}^n} \quad & \frac{1}{2}\boldsymbol{x}^T \boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u} \\ & \boldsymbol{b_l} \leq \boldsymbol{A}^T \boldsymbol{x} \leq \boldsymbol{b_u}, \end{aligned}$$

and this program is solved by solving a set of equality constrained QP's

$$\begin{aligned} \min_{\boldsymbol{x} \in \mathbb{R}^n} \quad & \frac{1}{2}\boldsymbol{x}^T \boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T \boldsymbol{x} \\ \text{s.t.} \quad & \bar{\boldsymbol{A}}^T \boldsymbol{x} = \bar{\boldsymbol{b}}. \end{aligned}$$

The methods we describe are the primal active set method and the dual active set method. Within these methods the Karush-Kuhn-Tucker (KKT) system[1]

$$\begin{pmatrix} \boldsymbol{G} & -\bar{\boldsymbol{A}} \\ -\bar{\boldsymbol{A}}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\lambda} \end{pmatrix} = -\begin{pmatrix} \boldsymbol{g} \\ \bar{\boldsymbol{b}} \end{pmatrix}$$

is solved using the range space procedure or the null space procedure. These methods in themselves fulfill the demand of reliability, while the demand of efficiency is obtained by refinement of these methods.

1.1 Research Objective

We will investigate the primal and dual active set methods for solving QP's. Thus we will discuss the range and the null space procedures together with different refinements for gaining efficiency and reliability. The methods and

[1] This is the KKT system of the primal program, the KKT system of the dual program is found in (4.69) at page 63.

procedures for solving QP's will be implemented and tested in order to determine the best suited combination in terms of efficiency for solving different types of problems. The problems can be divided into two categories, those with a low number of active constraints in relation to the number of variables, and problems where the number of active constraints is high in relation to the number of variables. Finally we will discuss and implement the SQP method to find out how our QP solver performs in this setting.

1.2 Thesis Structure

The thesis is divided into five main areas: Equality constrained quadratic programming, updating of matrix factorizations, active set methods, test and refinements and nonlinear programming.

Equality Constrained Quadratic Programming

In this chapter we present two methods for solving equality constrained QP's, namely the range space procedure and the null space procedure. The methods are implemented and tested, and their relative benefits, and drawbacks are investigated.

Updating of Matrix Factorizations

Both the null space and the range space procedure use matrix factorizations in solving the equality constrained QP. Whenever the constraint matrix is changed by either appending or removing a constraint, the matrix factorizations can be updated using Givens rotations. By avoiding complete re-factorization, computational savings are achieved. This is the subject of this chapter and methods for updating the QR and the Cholesky factorizations are presented.

Active Set Methods

Inequality constrained QP's can be solved using active set methods. These methods find a solution by solving a sequence of equality constrained QP's, where the difference between two consecutive iterations is a single appended or removed constraint. In this chapter we present the primal active set method and the dual active set method.

Test and Refinements

In this chapter we test how the presented methods perform in practice, when combined in different ways. We also implement some refinements, and their impact on computational speed and stability are likewise tested.

Nonlinear Programming

SQP is an efficient method of nonlinear constrained optimization. The basic idea is Newton's method, where each step is generated as an inequality constrained QP. Implementation, discussion and testing of SQP are the topics of this chapter.

Chapter 2

Equality Constrained Quadratic Programming

In this section we present various algorithms for solving convex[1] equality constrained QP's. The problem to be solved is

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

$$\text{s.t.} \qquad \boldsymbol{A}^T \boldsymbol{x} = \boldsymbol{b}, \tag{2.1b}$$

where $\boldsymbol{G} \in \mathbb{R}^{n \times n}$ is the Hessian matrix of the objective function f. The Hessian matrix must be symmetric and positive semi definite[2]. $\boldsymbol{A} \in \mathbb{R}^{n \times m}$ is the constraint matrix (coefficient matrix of the constraints), where n is the number of variables and m is the number of constraints. $\boldsymbol{A}$ has full column rank, that is the constraints are linearly independent. The right hand side of the constraints is $\boldsymbol{b} \in \mathbb{R}^m$ and $\boldsymbol{g} \in \mathbb{R}^n$ denotes the coefficients of the linear term of the objective function.

[1] The range space procedure presented in section 2.1 requires a strictly convex QP.

[2] The range space procedure presented in section 2.1 requires $\boldsymbol{G}$ to be positive definite.

From the Lagrangian function

$$L(\boldsymbol{x}, \boldsymbol{\lambda}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x} - \boldsymbol{\lambda}^T(\boldsymbol{A}^T\boldsymbol{x} - \boldsymbol{b}), \tag{2.2}$$

which is differentiated according to $\boldsymbol{x}$ and the Lagrange multipliers $\boldsymbol{\lambda}$

$$\nabla_{\boldsymbol{x}} L(\boldsymbol{x}, \boldsymbol{\lambda}) = \boldsymbol{G}\boldsymbol{x} + \boldsymbol{g} - \boldsymbol{A}\boldsymbol{\lambda} \tag{2.3a}$$

$$\nabla_{\boldsymbol{\lambda}} L(\boldsymbol{x}, \boldsymbol{\lambda}) = -\boldsymbol{A}^T\boldsymbol{x} + \boldsymbol{b}, \tag{2.3b}$$

the problem can be formulated as the Karush-Kuhn-Tucker (KKT) system

$$\begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\lambda} \end{pmatrix} = -\begin{pmatrix} \boldsymbol{g} \\ \boldsymbol{b} \end{pmatrix}. \tag{2.4}$$

The KKT system is basically a set of linear equations, and therefore general solvers for linear systems could be used, e.g. Gaussian elimination. In order to solve a KKT system as fast and reliable as possible, we want to use Cholesky and QR factorizations. But according to Gould in Nocedal and Wright [14] the KKT matrix is indefinite, and therefore it is not possible to solve it by use of either of the two factorizations. In this chapter, we present two procedures for solving the KKT system by dividing it into subproblems, on which it is possible to use these factorizations. Namely the range space procedure and the null space procedure. We also investigate their individual benefits and drawbacks.

2.1 Range Space Procedure

The range space procedure based on Nocedal and Wright [14] and Gill *et al.* [2] solves the KKT system (2.4), corresponding to the convex equality constrained QP (2.1). The Hessian matrix $\boldsymbol{G} \in \mathbb{R}^{n \times n}$ must be symmetric and positive definite, because the procedure uses the inverted Hessian matrix $\boldsymbol{G}^{-1}$. The KKT system

$$\begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\lambda} \end{pmatrix} = - \begin{pmatrix} \boldsymbol{g} \\ \boldsymbol{b} \end{pmatrix} \tag{2.5}$$

can be interpreted as two equations

$$\boldsymbol{G}\boldsymbol{x} - \boldsymbol{A}\boldsymbol{\lambda} = -\boldsymbol{g} \tag{2.6a}$$

$$\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}. \tag{2.6b}$$

Isolating $\boldsymbol{x}$ in (2.6a) gives

$$\boldsymbol{x} = \boldsymbol{G}^{-1}\boldsymbol{A}\boldsymbol{\lambda} - \boldsymbol{G}^{-1}\boldsymbol{g}, \tag{2.7}$$

and substituting (2.7) into (2.6b) gives us one equation with one unknown $\boldsymbol{\lambda}$

$$\boldsymbol{A}^T(\boldsymbol{G}^{-1}\boldsymbol{A}\boldsymbol{\lambda} - \boldsymbol{G}^{-1}\boldsymbol{g}) = \boldsymbol{b}, \tag{2.8}$$

which is equivalent to

$$\boldsymbol{A}^T\boldsymbol{G}^{-1}\boldsymbol{A}\boldsymbol{\lambda} = \boldsymbol{A}^T\boldsymbol{G}^{-1}\boldsymbol{g} + \boldsymbol{b}. \tag{2.9}$$

From the Cholesky factorization of $\boldsymbol{G}$ we get $\boldsymbol{G} = \boldsymbol{L}\boldsymbol{L}^T$ and $\boldsymbol{G}^{-1} = (\boldsymbol{L}^T)^{-1}\boldsymbol{L}^{-1} = (\boldsymbol{L}^{-1})^T\boldsymbol{L}^{-1}$. This is inserted in (2.9)

$$\boldsymbol{A}^T(\boldsymbol{L}^{-1})^T\boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{\lambda} = \boldsymbol{A}^T(\boldsymbol{L}^{-1})^T\boldsymbol{L}^{-1}\boldsymbol{g} + \boldsymbol{b} \tag{2.10}$$

so

$$(\boldsymbol{L}^{-1}\boldsymbol{A})^T\boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{\lambda} = (\boldsymbol{L}^{-1}\boldsymbol{A})^T\boldsymbol{L}^{-1}\boldsymbol{g} + \boldsymbol{b}. \tag{2.11}$$

From simplifying (2.11), by defining $\boldsymbol{K} = \boldsymbol{L}^{-1}\boldsymbol{A}$ and $\boldsymbol{w} = \boldsymbol{L}^{-1}\boldsymbol{g}$, where $\boldsymbol{K}$ can be found as the solution to $\boldsymbol{L}\boldsymbol{K} = \boldsymbol{A}$, and $\boldsymbol{w}$ as the solution to $\boldsymbol{L}\boldsymbol{w} = \boldsymbol{g}$, we get

$$\boldsymbol{K}^T\boldsymbol{K}\boldsymbol{\lambda} = \boldsymbol{K}^T\boldsymbol{w} + \boldsymbol{b}. \tag{2.12}$$

By now $\boldsymbol{K}$, $\boldsymbol{w}$ and $\boldsymbol{b}$ are known, and by computing $\boldsymbol{z} = \boldsymbol{K}^T\boldsymbol{w}+\boldsymbol{b}$ and $\boldsymbol{H} = \boldsymbol{K}^T\boldsymbol{K}$ we reformulate (2.12) into

$$\boldsymbol{H}\boldsymbol{\lambda} = \boldsymbol{z}. \tag{2.13}$$

The matrix $\boldsymbol{G}$ is positive definite and the matrix $\boldsymbol{A}$ has full column rank, so $\boldsymbol{H}$ is also positive definite. This makes it possible to Cholesky factorize $\boldsymbol{H} = \boldsymbol{M}\boldsymbol{M}^T$, and by backward and forward substitution $\boldsymbol{\lambda}$ is found from

$$\boldsymbol{M}\boldsymbol{M}^{\boldsymbol{T}}\boldsymbol{\lambda} = \boldsymbol{z}. \tag{2.14}$$

Substituting $\boldsymbol{M}^{\boldsymbol{T}}\boldsymbol{\lambda}$ with $\boldsymbol{q}$ gives

$$\boldsymbol{M}\boldsymbol{q} = \boldsymbol{z}, \tag{2.15}$$

and by forward substitution $\boldsymbol{q}$ is found. Now $\boldsymbol{\lambda}$ is found by backward substitution in

$$\boldsymbol{M}^{\boldsymbol{T}}\boldsymbol{\lambda} = \boldsymbol{q}. \tag{2.16}$$

We now know $\boldsymbol{\lambda}$ and from (2.6a) we find $\boldsymbol{x}$ as follows

$$\boldsymbol{G}\boldsymbol{x} = \boldsymbol{A}\boldsymbol{\lambda} - \boldsymbol{g} \tag{2.17}$$

gives us

$$\boldsymbol{L}\boldsymbol{L}^T\boldsymbol{x} = \boldsymbol{A}\boldsymbol{\lambda} - \boldsymbol{g}, \tag{2.18}$$

and

$$\boldsymbol{L}^T\boldsymbol{x} = \boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{\lambda} - \boldsymbol{L}^{-1}\boldsymbol{g}, \tag{2.19}$$

which is equivalent to

$$\boldsymbol{L}^T\boldsymbol{x} = \boldsymbol{K}\boldsymbol{\lambda} - \boldsymbol{w}. \tag{2.20}$$

As $\boldsymbol{K}, \boldsymbol{\lambda}$ and $\boldsymbol{w}$ are now known, $\boldsymbol{r}$ is computed as $\boldsymbol{r} = \boldsymbol{K}\boldsymbol{\lambda} - \boldsymbol{w}$, and by backward substitution $\boldsymbol{x}$ is found in

$$\boldsymbol{L}^T\boldsymbol{x} = \boldsymbol{r}. \tag{2.21}$$

The range space procedure requires $\boldsymbol{G}$ to be positive definite as $\boldsymbol{G}^{-1}$ is needed. It is obvious, that the procedure is most efficient, when $\boldsymbol{G}^{-1}$ is easily computed. In other words, when it is well-conditioned and even better, if $\boldsymbol{G}$ is a diagonal-matrix or can be computed a priori. Another bottleneck of the procedure is the factorization of the matrix $\boldsymbol{A}^T\boldsymbol{G}^{-1}\boldsymbol{A} \in \mathbb{R}^{m\times m}$. The smaller this matrix is, the easier the factorization gets. This means, that the procedure is most effecient, when the number of constraints is small compared to the number of variables.

Algorithm 2.1.1 summarizes how the calculations in the range space procedure are carried out.

Algorithm 2.1.1: Range Space Procedure.
Note: The algorithm requires $\boldsymbol{G}$ to be positive definite and $\boldsymbol{A}$ to have full column rank.

Cholesky factorize $\boldsymbol{G} = \boldsymbol{L}\boldsymbol{L}^T$
Compute $\boldsymbol{K}$ by solving $\boldsymbol{L}\boldsymbol{K} = \boldsymbol{A}$
Compute $\boldsymbol{w}$ by solving $\boldsymbol{L}\boldsymbol{w} = \boldsymbol{g}$
Compute $\boldsymbol{H} = \boldsymbol{K}^T\boldsymbol{K}$
Compute $\boldsymbol{z} = \boldsymbol{K}^T\boldsymbol{w} + \boldsymbol{b}$
Cholesky factorize $\boldsymbol{H} = \boldsymbol{M}\boldsymbol{M}^T$
Compute $\boldsymbol{q}$ by solving $\boldsymbol{M}\boldsymbol{q} = \boldsymbol{z}$
Compute $\boldsymbol{\lambda}$ by solving $\boldsymbol{M}^T\boldsymbol{\lambda} = \boldsymbol{q}$
Compute $\boldsymbol{r} = \boldsymbol{K}\boldsymbol{\lambda} - \boldsymbol{w}$
Compute $\boldsymbol{x}$ by solving $\boldsymbol{L}^T\boldsymbol{x} = \boldsymbol{r}$

2.2 Null Space Procedure

The null space procedure based on Nocedal and Wright [14] and Gill *et al.* [3] solves the KKT system (2.4) using the null space of $\boldsymbol{A} \in \mathbb{R}^{n\times m}$. This procedure does not need $\boldsymbol{G} \in \mathbb{R}^{n\times n}$ to be positive definite but only positive semi definite. This means, that it is not restricted to strictly convex quadratic programs. The KKT system to be solved is

$$\begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\lambda} \end{pmatrix} = - \begin{pmatrix} \boldsymbol{g} \\ \boldsymbol{b} \end{pmatrix}, \tag{2.22}$$

where $\boldsymbol{A}$ has full column rank. We compute the null space using the QR factorization of $\boldsymbol{A}$

$$\boldsymbol{A} = \boldsymbol{Q}\begin{pmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{pmatrix} = (\boldsymbol{Y}\ \boldsymbol{Z})\begin{pmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{pmatrix}, \tag{2.23}$$

where $\boldsymbol{Z} \in \mathbb{R}^{n\times(n-m)}$ is the null space and $\boldsymbol{Y} \in \mathbb{R}^{n\times m}$ is the range space. $(\boldsymbol{Y}\ \boldsymbol{Z}) \in \mathbb{R}^{n\times n}$ is orthogonal and $\boldsymbol{R} \in \mathbb{R}^{m\times m}$ is upper triangular.

By defining $\boldsymbol{x} = \boldsymbol{Q}\boldsymbol{p}$ we write

$$\boldsymbol{x} = \boldsymbol{Q}\boldsymbol{p} = (\boldsymbol{Y}\ \boldsymbol{Z})\boldsymbol{p} = (\boldsymbol{Y}\ \boldsymbol{Z})\begin{pmatrix} \boldsymbol{p_y} \\ \boldsymbol{p_z} \end{pmatrix} = \boldsymbol{Y}\boldsymbol{p_y} + \boldsymbol{Z}\boldsymbol{p_z}. \tag{2.24}$$

Using this formulation, we can reformulate $(\boldsymbol{x}\ \boldsymbol{\lambda})^T$ in (2.22) as

$$\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \boldsymbol{Y} & \boldsymbol{Z} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \boldsymbol{p_y} \\ \boldsymbol{p_z} \\ \boldsymbol{\lambda} \end{pmatrix}, \tag{2.25}$$

and because $(\boldsymbol{Y}\ \boldsymbol{Z})$ is orthogonal we also have

$$\begin{pmatrix} \boldsymbol{Y} & \boldsymbol{Z} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{pmatrix}^T \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \boldsymbol{p_y} \\ \boldsymbol{p_z} \\ \boldsymbol{\lambda} \end{pmatrix}. \tag{2.26}$$

Now we will use (2.25) and (2.26) to express the KKT system in a more detailed form, in which it becomes clear what part corresponds to the null space. Inserting (2.25) and (2.26) in (2.22) gives

$$\begin{pmatrix} \boldsymbol{Y} & \boldsymbol{Z} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{pmatrix}^T \begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{Y} & \boldsymbol{Z} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \boldsymbol{p_y} \\ \boldsymbol{p_z} \\ \boldsymbol{\lambda} \end{pmatrix} = -\begin{pmatrix} \boldsymbol{Y} & \boldsymbol{Z} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{pmatrix}^T \begin{pmatrix} \boldsymbol{g} \\ \boldsymbol{b} \end{pmatrix}, \tag{2.27}$$

which is equivalent to

$$\begin{pmatrix} \boldsymbol{Y}^T\boldsymbol{GY} & \boldsymbol{Y}^T\boldsymbol{GZ} & -(\boldsymbol{A}^T\boldsymbol{Y})^T \\ \boldsymbol{Z}^T\boldsymbol{GY} & \boldsymbol{Z}^T\boldsymbol{GZ} & -(\boldsymbol{A}^T\boldsymbol{Z})^T \\ -\boldsymbol{A}^T\boldsymbol{Y} & -\boldsymbol{A}^T\boldsymbol{Z} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{p_y} \\ \boldsymbol{p_z} \\ \boldsymbol{\lambda} \end{pmatrix} = -\begin{pmatrix} \boldsymbol{Y}^T\boldsymbol{g} \\ \boldsymbol{Z}^T\boldsymbol{g} \\ \boldsymbol{b} \end{pmatrix}. \tag{2.28}$$

By definition $\boldsymbol{A}^T\boldsymbol{Z} = \boldsymbol{0}$, which simplifies (2.28) to

$$\begin{pmatrix} \boldsymbol{Y}^T\boldsymbol{GY} & \boldsymbol{Y}^T\boldsymbol{GZ} & -(\boldsymbol{A}^T\boldsymbol{Y})^T \\ \boldsymbol{Z}^T\boldsymbol{GY} & \boldsymbol{Z}^T\boldsymbol{GZ} & \boldsymbol{0} \\ -\boldsymbol{A}^T\boldsymbol{Y} & \boldsymbol{0} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{p_y} \\ \boldsymbol{p_z} \\ \boldsymbol{\lambda} \end{pmatrix} = -\begin{pmatrix} \boldsymbol{Y}^T\boldsymbol{g} \\ \boldsymbol{Z}^T\boldsymbol{g} \\ \boldsymbol{b} \end{pmatrix}. \tag{2.29}$$

This system can be solved using backward substitution, but to do this, we need the following statement based on (2.23)

$$\boldsymbol{A} = \boldsymbol{YR} \tag{2.30a}$$

$$\boldsymbol{A}^T = (\boldsymbol{YR})^T \tag{2.30b}$$

$$\boldsymbol{A}^T\boldsymbol{Y} = (\boldsymbol{YR})^T\boldsymbol{Y} \tag{2.30c}$$

$$\boldsymbol{A}^T\boldsymbol{Y} = \boldsymbol{R}^T\boldsymbol{Y}^T\boldsymbol{Y} \tag{2.30d}$$

$$\boldsymbol{A}^T\boldsymbol{Y} = \boldsymbol{R}^T. \tag{2.30e}$$

and therefore the last block row from (2.29)

$$-\boldsymbol{A}^T\boldsymbol{Y}\boldsymbol{p_y} = -\boldsymbol{b} \tag{2.31}$$

is equivalent to

$$\boldsymbol{R}^T \boldsymbol{p_y} = \boldsymbol{b}. \tag{2.32}$$

As $\boldsymbol{R}$ is upper triangular this equation has a unique solution. When we have computed $\boldsymbol{p_y}$ we can solve the middle block row in (2.29)

$$\boldsymbol{Z}^T \boldsymbol{G} \boldsymbol{Y} \boldsymbol{p_y} + \boldsymbol{Z}^T \boldsymbol{G} \boldsymbol{Z} \boldsymbol{p_z} = -\boldsymbol{Z}^T \boldsymbol{g}, \tag{2.33}$$

The only unknown is $\boldsymbol{p_z}$, which we find by solving

$$(\boldsymbol{Z}^T \boldsymbol{G} \boldsymbol{Z}) \boldsymbol{p_z} = -\boldsymbol{Z}^T (\boldsymbol{G} \boldsymbol{Y} \boldsymbol{p_y} + \boldsymbol{g}). \tag{2.34}$$

The reduced Hessian matrix $(\boldsymbol{Z}^T \boldsymbol{G} \boldsymbol{Z}) \in \mathbb{R}^{(n-m)\times(n-m)}$ is positive definite and therefore the solution to (2.34) is unique. We find it by use of the Cholesky factorization $(\boldsymbol{Z}^T \boldsymbol{G} \boldsymbol{Z}) = \boldsymbol{L}\boldsymbol{L}^T$. Now, having computed both $\boldsymbol{p_y}$ and $\boldsymbol{p_z}$, we find $\boldsymbol{\lambda}$ from the first block row in (2.29)

$$\boldsymbol{Y}^T \boldsymbol{G} \boldsymbol{Y} \boldsymbol{p_y} + \boldsymbol{Y}^T \boldsymbol{G} \boldsymbol{Z} \boldsymbol{p_z} - (\boldsymbol{A}^T \boldsymbol{Y})^T \boldsymbol{\lambda} = -\boldsymbol{Y}^T \boldsymbol{g}, \tag{2.35}$$

which is equivalent to

$$(\boldsymbol{A}^T \boldsymbol{Y})^T \boldsymbol{\lambda} = \boldsymbol{Y}^T \boldsymbol{G} (\boldsymbol{Y} \boldsymbol{p_y} + \boldsymbol{Z} \boldsymbol{p_z}) + \boldsymbol{Y}^T \boldsymbol{g}. \tag{2.36}$$

Using (2.24) $\boldsymbol{x} = \boldsymbol{Y}\boldsymbol{p_y} + \boldsymbol{Z}\boldsymbol{p_z}$ and (2.30) $\boldsymbol{A}^T \boldsymbol{Y} = \boldsymbol{R}^T$ this can be reformulated into

$$\boldsymbol{R}\boldsymbol{\lambda} = \boldsymbol{Y}^T (\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}) \tag{2.37}$$

and because $\boldsymbol{R}$ is upper triangular, this equation also has a unique solution, which is found by backward substitution.

This is the most efficient procedure, when the degree of freedom $n - m$ is small, i.e. when the number of constraints is large compared to the number of variables. The reduced Hessian matrix $\boldsymbol{Z}^T \boldsymbol{G} \boldsymbol{Z} \in \mathbb{R}^{(n-m)\times(n-m)}$ grows smaller, when m approaches n, and is thereby inexpensive to factorize. The most expensive part of the computations is the QR factorization of $\boldsymbol{A}$. While the null space $\boldsymbol{Z}$ can be found in a number of different ways, we have chosen to use QR factorization because it makes $\boldsymbol{Y}$ and $\boldsymbol{Z}$ orthogonal. In this way, we preserve numerical stability, because the conditioning of the reduced Hessian matrix $\boldsymbol{Z}^T\boldsymbol{G}\boldsymbol{Z}$ is at least as good as the conditioning of $\boldsymbol{G}$.

Algorithm 2.2.1 summarizes how the calculations in the null space procedure are carried out.

Algorithm 2.2.1: Null Space Procedure.
Note: The algorithm requires $\boldsymbol{G}$ to be positive semi definite and $\boldsymbol{A}$ to have full column rank.

QR factorize $\boldsymbol{A} = (\boldsymbol{Y}\ \boldsymbol{Z})\begin{pmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{pmatrix}$
Cholesky factorize $\boldsymbol{Z}^T\boldsymbol{G}\boldsymbol{Z} = \boldsymbol{L}\boldsymbol{L}^T$
Compute $\boldsymbol{p}_y$ by solving $\boldsymbol{R}^T\boldsymbol{p}_y = \boldsymbol{b}$
Compute $\boldsymbol{g}_z = -\boldsymbol{Z}^T(\boldsymbol{G}\boldsymbol{Y}\boldsymbol{p}_y + \boldsymbol{g})$
Compute $\boldsymbol{r}$ by solving $\boldsymbol{L}\boldsymbol{r} = \boldsymbol{g}_z$
Compute $\boldsymbol{p}_z$ by solving $\boldsymbol{L}^T\boldsymbol{p}_z = \boldsymbol{r}$
Compute $\boldsymbol{x} = \boldsymbol{Y}\boldsymbol{p}_y + \boldsymbol{Z}\boldsymbol{p}_z$
Compute $\boldsymbol{\lambda}$ by solving $\boldsymbol{R}\boldsymbol{\lambda} = \boldsymbol{Y}^T(\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g})$

2.3 Computational Cost of the Range and the Null Space Procedures

In this section we want to find out, how the range space and the null space procedures perform individually. We also consider whether it is worthwhile to shift between the two procedures dynamically.

2.3.1 Computational Cost of the Range Space Procedure

In the range space procedure there are three dominating computations in relation to time consumption. The computation of $\boldsymbol{K} \in \mathbb{R}^{n\times m}$, computation of and Cholesky factorization of $\boldsymbol{H} \in \mathbb{R}^{m\times m}$.

Since $\boldsymbol{L} \in \mathbb{R}^{n\times n}$ is lower triangular, solving $\boldsymbol{LK} = \boldsymbol{A}$ with respect to $\boldsymbol{K}$ is done by simple forward substitution. The amount of work involved in forward substitution is n^2 per column, according to L. Eldén, L. Wittmeyer-Koch and H.B. Nielsen [18]. Since $\boldsymbol{K}$ contains m columns, the total cost for computing $\boldsymbol{K}$ is n^2m.

We define $\boldsymbol{K}_T = \boldsymbol{K}^T \in \mathbb{R}^{m\times n}$. Making the inner product of two vectors of length n requires $2n$ operations. Since $\boldsymbol{K}_T$ consists of m rows and as mentioned above $\boldsymbol{K}$ contains m columns, then the computational workload involved in the matrix multiplication $\boldsymbol{H} = \boldsymbol{K}_T\boldsymbol{K}$ is $2nm^2$.

The size of $\boldsymbol{H}$ is $m \times m$, so the computational cost of the Cholesky factorization is roughly $\frac{1}{3}m^3$, according to L. Eldén, L. Wittmeyer-Koch and H.B. Nielsen [18].

Thus, we can estimate the total computational cost of the range space procedure as

$$\tfrac{1}{3}m^3 + 2nm^2 + n^2m \tag{2.38}$$

and since $0 \le m \le n$, the total computational workload will roughly be in the range

$$0 \le \tfrac{1}{3}m^3 + 2nm^2 + n^2m \le \tfrac{10}{3}n^3. \tag{2.39}$$

Here we also see, that the range space procedure gets slower, as the number of constraints compared to the number of variables increases.

Figure 2.1 shows the theoretical computational speed of the range space procedure. As stated above, it is obvious that the method gets slower as the number of constraints increases in comparison to the number of variables.

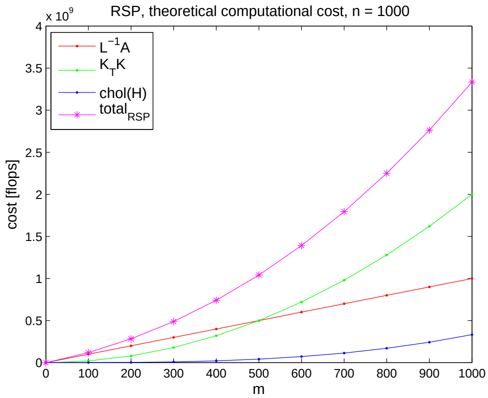


Figure 2.1: Theoretical computational speed for the range procedure.

2.3.2 Computational Cost of the Null Space Procedure

The time consumption of the null space procedure is dominated by two computations. The QR factorization of the constraint matrix $\boldsymbol{A}$ and computation of the reduced Hessian matrix $\boldsymbol{Z}^T\boldsymbol{G}\boldsymbol{Z} \in \mathbb{R}^{(n-m)\times(n-m)}$.

With $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ as our point of departure, the computational work of the QR factorization is in the region $2m^2(n - \frac{1}{3}m)$, see L. Eldén, L. Wittmeyer-Koch and H.B. Nielsen [18]. The QR factorization of $\boldsymbol{A}$ is

$$\boldsymbol{A} = \boldsymbol{Q}\begin{pmatrix}\boldsymbol{R}\\ \boldsymbol{0}\end{pmatrix} = \begin{pmatrix}\boldsymbol{Y} & \boldsymbol{Z}\end{pmatrix}\begin{pmatrix}\boldsymbol{R}\\ \boldsymbol{0}\end{pmatrix}, \tag{2.40}$$

where $\boldsymbol{Y} \in \mathbb{R}^{n\times m}$, $\boldsymbol{Z} \in \mathbb{R}^{n\times(n-m)}$, $\boldsymbol{R} \in \mathbb{R}^{m\times m}$ and $\boldsymbol{0} \in \mathbb{R}^{(n-m)\times m}$.

We now want to find the amount of work involved in computing the reduced Hessian matrix $\boldsymbol{Z}^T\boldsymbol{G}\boldsymbol{Z}$. We define $\boldsymbol{Z}_T = \boldsymbol{Z}^T \in \mathbb{R}^{(n-m)\times n}$. The computational workload of making the inner product of two vectors in $\mathbb{R}^n$ is $2n$. Since $\boldsymbol{Z}_T$

contains $n - m$ rows and G consists of n columns, the computational cost of the matrix product $\boldsymbol{Z}_T\boldsymbol{G}$ is $2n(n-m)n$. Because $(\boldsymbol{Z}_T\boldsymbol{G}) \in \mathbb{R}^{(n-m)\times n}$ and $\boldsymbol{Z}$ consists of $n - m$ columns, the amount of work involved in the matrix product $(\boldsymbol{Z}_T\boldsymbol{G})\boldsymbol{Z}$ is $2n(n-m)(n-m)$. Therefore the computational cost of making the reduced Hessian matrix is

$$2n(n-m)n + 2n(n-m)(n-m) = 2n(n-m)(2n-m). \tag{2.41}$$

So the total computational cost of the null space procedure is roughly

$$2m^2(n - \tfrac{1}{3}m) + 2n(n-m)(2n-m) \tag{2.42}$$

and since $0 \le m \le n$, the total computational workload is estimated to be in the range of

$$\tfrac{4}{3}n^3 \le 2m^2(n - \tfrac{1}{3}m) + 2n(n-m)(2n-m) \le 4n^3. \tag{2.43}$$

Therefore the null space procedure accelerates, as the number of constraints compared to the number of variables increases. Figure 2.2 illustrates this.

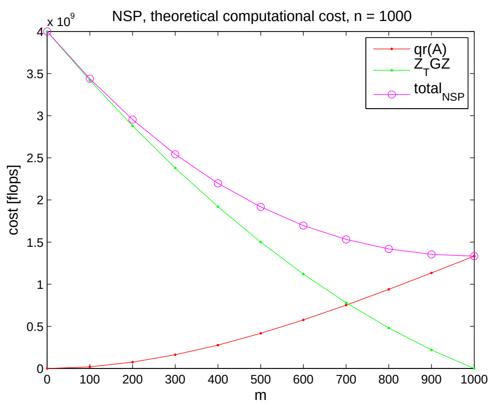


Figure 2.2: Theoretical computational costs for the null space procedure.

2.3.3 Comparing Computational Costs

To take advantage of the individual differences in computational speeds, we want to find out at what ratio between the number of constraints related to the number of variables, the null space procedure gets faster than the range space procedure. This is done by comparing the computational costs of both procedures, hereby finding the point, at which they run equally fast. With respect to m we solve the polynomial

$$\begin{aligned}\tfrac{1}{3}m^3 + 2nm^2 + n^2m - 2m^2(n - \tfrac{1}{3}m) - 2n(n-m)(2n-m) = \\ m^3 - 2nm^2 + 7n^2m - 4n^3 = 0, \end{aligned} \tag{2.44}$$

where by we find the relation to be $m \simeq 0.65n$.

In figure 2.3 we have $n = 1000$ and $0 \le m \le n$, so the ratio, i.e. the point at which one should shift from using the range space to using the null space procedure is of course to be found at $m \simeq 650$.

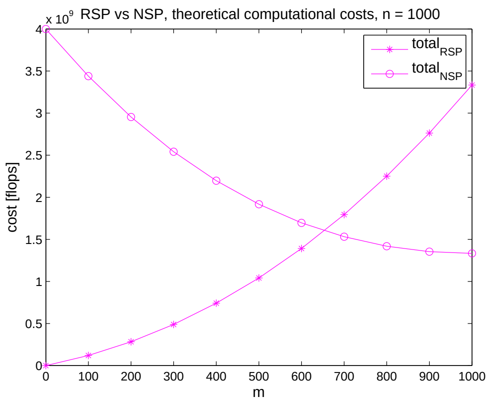


Figure 2.3: Total estimated theoretical computational costs for the range space and the null space procedures.

We made some testruns of the the two procedures by setting up a KKT system consisting of identity matrices. The theoretical computational costs are based on full matrices, and we know, that MATLAB treats identity matrices like full matrices. So by means of this simple KKT system we are able to compare the theoretical behavior of the respective procedures with the real behavior of our implementation. The test setup consists of $n = 1000$ variables and $0 \le m \le n$

constraints, as illustrated in figures 2.4 and 2.5. The black curves represent all computations carried out by the two procedures. It is clear, that they run parallel to the magenta representing the dominating computations. This verifies our comparison between the theoretical computational workloads with our test setup.

In figure 2.4 we test the range space procedure, and the behavior is just as expected, when compared to the theory represented in figure 2.1.

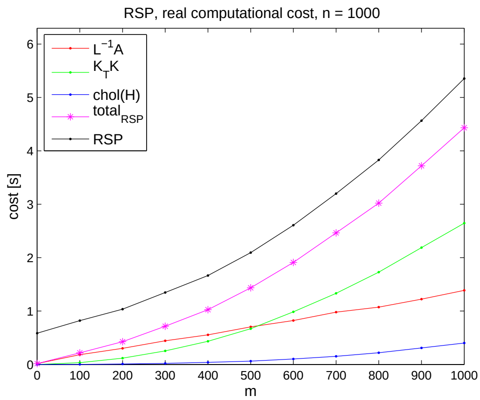


Figure 2.4: Real computational cost for the range space procedure.

In figure 2.5 the null space procedure is tested. The computation of the reduced Hessian matrix $\boldsymbol{Z}_T\boldsymbol{G}\boldsymbol{Z}$ behaves as expected. The behavior of the QR factorization of $\boldsymbol{A}$ however is not as expected, compared to figure 2.2. When $0 \leq m \lesssim 100$, the computational time is too small to be properly measured. The QR factorization still behaves some what unexpectedly, when $100 \lesssim m \leq 1000$. Hence the advantage of shifting from the range space to the null space procedure decreases. In other words, the ratio between the number of constraints related to the number of variables, where the null space procedure gets faster than the range space procedure, is larger than expected. Therefore using the null space procedure might actually prove to be a disadvantage. This is clearly in figure 2.6, where the dominating computations for the range space and the null space procedures are compared to each other. From this plot it is clear, that the ratio indicating when to shift procedure is much bigger in practice than in theory. The cause of this could be an inappropriate implementation of the QR factorization in MATLAB, architecture of the processing unit, memory access etc.. Perhaps implementation of the QR factorization in a low level language could prove different.

It must be mentioned at this point, that the difference in unit on the abscissa in all figures in this section does not influence the shape of the curves between theory and testruns, because the only difference between them is the constant ratio time/flop.

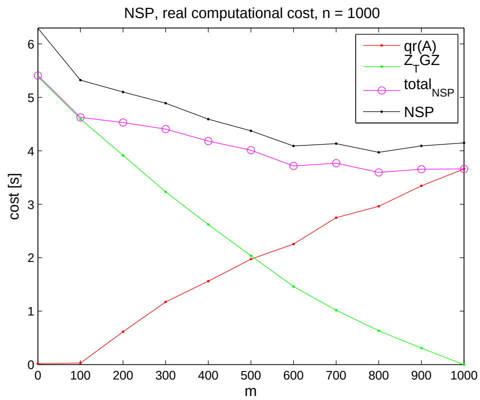


Figure 2.5: Real computational cost for the null space procedure.

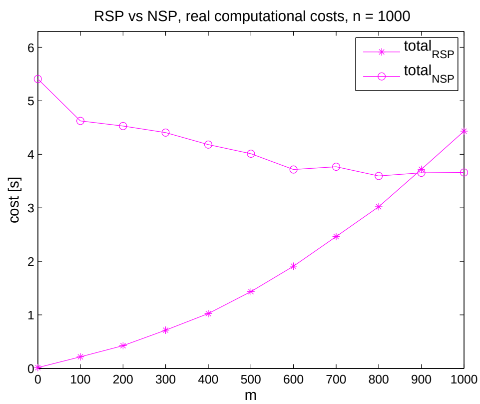


Figure 2.6: Real computational costs for the range space and the null space procedures.

Chapter 3

Updating Procedures for Matrix Factorization

Matrix factorization is used, when solving an equality constrained QP. The factorization of a square matrix of size $n \times n$ has computational complexity $O(n^3)$.

As we will describe in chapter 4, the solution of an inequality constrained QP is found by solving a sequence of equality constrained QP's. The difference between two consecutive equality constrained QP's in this sequence is one single appended or removed constraint. This is the procedure of the active set methods.

Because of this property, the factorization of the matrices can be done more efficiently than complete refactorization. This is done by an updating procedure where the current factorization of the matrices in the sequence, is partly used to factorize the matrices for the next iteration. The computational complexity of this updating procedure is $O(n^2)$ and therefore a factor n faster than a complete factorization. This is important in particular for large-scale problems. The updating procedure discussed in the following is based on Givens rotations and Givens reflections.

3.1 Givens rotations and Givens reflections

Givens rotations and reflections are methods for introducing zeros in a vector. This is achieved by rotating or reflecting the coordinate system according to an angle or a line. This angle or line is defined so that one of the coordinates of a given point $\boldsymbol{p}$ becomes zero. As both methods serve the same purpose, we have chosen to use only Givens rotations which we will explain in the following. This theory is based on the work of Golub and Van Loan [4] and Wilkinson [5]. A Givens rotation is graphically illustrated in figure 3.1. As illustrated we can rotate the coordinate system so the coordinates of $\boldsymbol{p}$ in the rotated system actually become $(x', 0)^T$.

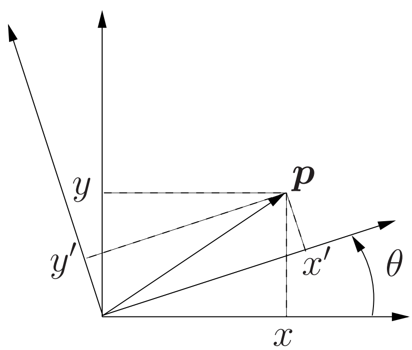


Figure 3.1: A Givens rotation rotates the coordinate system according to an angle θ.

The Givens rotation matrix $\hat{\boldsymbol{Q}} \in \mathbb{R}^{2\times 2}$ is defined as

$$\hat{\boldsymbol{Q}} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}, \quad c = \frac{x}{\sqrt{x^2+y^2}} = cos(\theta), \quad s = \frac{y}{\sqrt{x^2+y^2}} = sin(\theta), \quad (3.1)$$

and $(x, y)^T$, $x \neq 0 \;\wedge\; y \neq 0$, is the vector $\boldsymbol{p}$ in which we want to introduce a zero

$$
\begin{aligned}
\hat{\boldsymbol{Q}}\boldsymbol{p} &= \begin{pmatrix} \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} \\ -\frac{y}{\sqrt{x^2+y^2}} & \frac{x}{\sqrt{x^2+y^2}} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \\
&= \begin{pmatrix} \frac{x^2+y^2}{\sqrt{x^2+y^2}} \\ \frac{-xy+yx}{\sqrt{x^2+y^2}} \end{pmatrix} \\
&= \begin{pmatrix} \sqrt{x^2+y^2} \\ 0 \end{pmatrix}.
\end{aligned} \tag{3.2}
$$

If we want to introduce zeros in a vector $\boldsymbol{v} \in \mathbb{R}^n$, the corresponding rotation matrix is constructed by the identity matrix $\boldsymbol{I} \in \mathbb{R}^{n\times n}$, c and s. The matrix introduces one zero, modifies one element m and leaves the rest of the vector untouched

$$
\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & c & s \\ 0 & 0 & 0 & -s & c \end{pmatrix} \begin{pmatrix} x \\ \vdots \\ x \\ x \\ x \end{pmatrix} = \begin{pmatrix} x \\ \vdots \\ x \\ m \\ 0 \end{pmatrix}. \tag{3.3}
$$

Any Givens operation introduces only one zero at a time, but if we want to introduce more zeros, a sequence of Givens operations $\tilde{\boldsymbol{Q}} \in \mathbb{R}^{n\times n}$ can be constructed

$$
\tilde{\boldsymbol{Q}} = \hat{\boldsymbol{Q}}_{1,2}\hat{\boldsymbol{Q}}_{2,3}\cdots\hat{\boldsymbol{Q}}_{n-2,n-1}\hat{\boldsymbol{Q}}_{n-1,n}, \tag{3.4}
$$

which yields

$$
\tilde{\boldsymbol{Q}}\boldsymbol{v} = \tilde{\boldsymbol{Q}} \begin{pmatrix} x \\ x \\ \vdots \\ x \end{pmatrix} = \begin{pmatrix} \gamma \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \gamma = \pm||\boldsymbol{v}||_2. \tag{3.5}
$$

For example when $\boldsymbol{v} \in \mathbb{R}^4$ the process is as follows

$$\begin{pmatrix} x \\ x \\ x \\ x \end{pmatrix} \xrightarrow{\hat{\boldsymbol{Q}}_{3,4}} \begin{pmatrix} x \\ x \\ m \\ 0 \end{pmatrix} \xrightarrow{\hat{\boldsymbol{Q}}_{2,3}} \begin{pmatrix} x \\ m \\ 0 \\ 0 \end{pmatrix} \xrightarrow{\hat{\boldsymbol{Q}}_{1,2}} \begin{pmatrix} m \\ 0 \\ 0 \\ 0 \end{pmatrix}, \tag{3.6}$$

where m is the modified element.

3.2 Updating the QR Factorization

When a constraint is appended to, or removed from, the active set, updating the factorization is done using Givens rotations. This section is based on the work of Dennis and Schnabel [6], Gill *et al.* [7] and Golub and Van Loan [4] and describes how the updating procedure is carried out.

Appending a Constraint

Before the new column is appended, we take a closer look at the constraint matrix $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ and its QR-factorization $\boldsymbol{Q} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{R} \in \mathbb{R}^{m\times m}$. The constraint matrix $\boldsymbol{A}$ has full column rank, and can be written as

$$\boldsymbol{A} = \begin{pmatrix}\boldsymbol{a}_1 \dots \boldsymbol{a}_m\end{pmatrix}, \tag{3.7}$$

where $\boldsymbol{a}_i$ is the i^{th} column of the constraint matrix. The QR-factorization of $\boldsymbol{A}$ is

$$\boldsymbol{A} = \boldsymbol{Q}\begin{pmatrix}\boldsymbol{R}\\ \boldsymbol{0}\end{pmatrix}. \tag{3.8}$$

As $\boldsymbol{Q}$ is orthogonal we have $\boldsymbol{Q}^{-1} = \boldsymbol{Q}^T$, so

$$\boldsymbol{Q}^T\boldsymbol{A} = \begin{pmatrix}\boldsymbol{R}\\ \boldsymbol{0}\end{pmatrix}. \tag{3.9}$$

Inserting (3.7) in (3.9) gives

$$\boldsymbol{Q}^T\boldsymbol{A} = \boldsymbol{Q}^T\begin{pmatrix}\boldsymbol{a}_1 \dots \boldsymbol{a}_m\end{pmatrix}. \tag{3.10}$$

Expression (3.9) can be written as

$$\boldsymbol{Q}^T\boldsymbol{A} = \begin{pmatrix}\boldsymbol{R}\\ \hline \boldsymbol{0}\end{pmatrix} = \left(\begin{array}{ccc} x_{(1,1)} & \cdots & x_{(1,m)} \\ & \cdots & \cdots \\ & & x_{(m,m)} \\ \hline & \boldsymbol{0} & \end{array}\right). \tag{3.11}$$

Now we append the new column $\bar{\boldsymbol{a}} \in \mathbb{R}^n$ to the constraint matrix $\boldsymbol{A} \in \mathbb{R}^{n\times m}$, which becomes $\bar{\boldsymbol{A}} \in \mathbb{R}^{n\times m+1}$. To optimize the efficiency of the updating procedure, the new column is appended at index $m+1$. The new constraint matrix $\bar{\boldsymbol{A}}$ is

$$\bar{\boldsymbol{A}} = \big((\boldsymbol{a}_1 \dots \boldsymbol{a}_m) \quad \bar{\boldsymbol{a}}\big) \, . \tag{3.12}$$

Replacing $\boldsymbol{A}$ in (3.10) with $\bar{\boldsymbol{A}}$ gives

$$\boldsymbol{Q}^T \bar{\boldsymbol{A}} = \boldsymbol{Q}^T \, \big((\boldsymbol{a}_1 \dots \boldsymbol{a}_m) \quad \bar{\boldsymbol{a}}\big) \, , \tag{3.13}$$

which is equivalent to

$$\boldsymbol{Q}^T \bar{\boldsymbol{A}} = \big(\boldsymbol{Q}^T (\boldsymbol{a}_1 \dots \boldsymbol{a}_m) \quad \boldsymbol{Q}^T \bar{\boldsymbol{a}}\big) \, . \tag{3.14}$$

Thus from (3.10), (3.11) and (3.14) we have

$$\boldsymbol{Q}^T \bar{\boldsymbol{A}} = \left(\begin{array}{c|c} \boldsymbol{R} & \boldsymbol{v} \\ \hline \boldsymbol{0} & \boldsymbol{w} \end{array} \right), \quad \begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{w} \end{pmatrix} = \boldsymbol{Q}^T \bar{\boldsymbol{a}}, \tag{3.15}$$

where $\boldsymbol{v} \in \mathbb{R}^m$ and $\boldsymbol{w} \in \mathbb{R}^{n-m}$. This can be expressed as

$$\boldsymbol{Q}^T \bar{\boldsymbol{A}} = \left(\begin{array}{ccc|c} x_{(1,1)} & \dots & x_{(1,m)} & \\ & \dots & \dots & \boldsymbol{v} \\ & & x_{(m,m)} & \\ \hline & \boldsymbol{0} & & \boldsymbol{w} \end{array} \right). \tag{3.16}$$

Unless only the first element is different from zero in vector $\boldsymbol{w}$, the triangular structure is violated by appending $\bar{\boldsymbol{a}}$. By using Givens rotations, zeros are introduced in the vector $(\boldsymbol{v}, \boldsymbol{w})^T$ with a view to making the matrix upper triangular again. As a Givens operation only introduces one zero at a time, a sequence $\tilde{\boldsymbol{Q}} \in \mathbb{R}^{n\times n}$ of $n-m+1$ Givens rotations is used

$$\tilde{\boldsymbol{Q}} = \hat{\boldsymbol{Q}}_{(m+1,m+2)} \hat{\boldsymbol{Q}}_{(m+2,m+3)} \cdots \hat{\boldsymbol{Q}}_{(n-1,n)}, \tag{3.17}$$

where $\hat{\boldsymbol{Q}}_{(i+2,i+3)}$ defines the Givens rotation matrix that introduces one zero at index $i+3$ and modifies the element at index $i+2$. It is clear from (3.16) that the smallest amount of Givens rotations are needed, when $\bar{\boldsymbol{a}}$ is appended at index $m+1$. This sequence is constructed so that

$$\tilde{\boldsymbol{Q}} \left(\frac{\boldsymbol{v}}{\boldsymbol{w}} \right) = \begin{pmatrix} \boldsymbol{v} \\ \hline \gamma \\ 0 \\ \vdots \\ 0 \end{pmatrix}. \tag{3.18}$$

Now that the sequence $\tilde{\boldsymbol{Q}}$ has been constructed, when we multiply it with (3.16) we get

$$\begin{aligned} \tilde{\boldsymbol{Q}}\boldsymbol{Q}^T\bar{\boldsymbol{A}} &= \tilde{\boldsymbol{Q}} \left(\begin{array}{c|c} \boldsymbol{R} & \boldsymbol{v} \\ \hline \boldsymbol{0} & \boldsymbol{w} \end{array} \right) \\ &= \left(\begin{array}{c|c} \boldsymbol{R} & \boldsymbol{v} \\ \hline \boldsymbol{0} & \gamma \\ \hline \boldsymbol{0} & \boldsymbol{0} \end{array} \right) \\ &= \left(\begin{array}{ccc|c} x_{(1,1)} & \dots & x_{(1,m)} & \\ & \dots & \dots & \boldsymbol{v} \\ & & x_{(m,m)} & \\ \hline & \boldsymbol{0} & & \gamma \\ \hline & \boldsymbol{0} & & \boldsymbol{0} \end{array} \right) \\ &= \begin{pmatrix} \bar{\boldsymbol{R}} \\ \boldsymbol{0} \end{pmatrix}. \end{aligned} \tag{3.19}$$

This indicates, that the triangular shape is regained and the Givens operations only affects the elements in $\boldsymbol{w}$.

The QR-factorization of the new constraint matrix is

$$\bar{\boldsymbol{A}} = \bar{\boldsymbol{Q}} \begin{pmatrix} \bar{\boldsymbol{R}} \\ \boldsymbol{0} \end{pmatrix}. \tag{3.20}$$

Now that $\bar{\boldsymbol{R}}$ has been found, we only need to find $\bar{\boldsymbol{Q}}$ to complete the updating procedure. From (3.19) we have

$$\tilde{\boldsymbol{Q}}\boldsymbol{Q}^T\bar{\boldsymbol{A}} = \begin{pmatrix} \bar{\boldsymbol{R}} \\ \boldsymbol{0} \end{pmatrix}, \tag{3.21}$$

and because both $\tilde{\boldsymbol{Q}}$ and $\boldsymbol{Q}^T$ are orthogonal, this can be reformulated as

$$\bar{\boldsymbol{A}} = \boldsymbol{Q}\tilde{\boldsymbol{Q}}^T\bar{\boldsymbol{R}}. \tag{3.22}$$

From this expression it is seen that $\bar{\boldsymbol{Q}}$ is

$$\bar{\boldsymbol{Q}} = \boldsymbol{Q}\tilde{\boldsymbol{Q}}^T. \tag{3.23}$$

The updating procedure, when appending one constraint to the constraint matrix is summarized in algorithm 3.2.1.

Algorithm 3.2.1: Updating the QR-Factorization, when appending a column.
Note: Having $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ and its QR factorization, where $\boldsymbol{Q} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{R} \in \mathbb{R}^{m\times m}$. Appending a column $\bar{\boldsymbol{a}}$ to matrix $\boldsymbol{A}$ at index $m+1$ gives a new matrix $\bar{\boldsymbol{A}}$. The new factorization is $\bar{\boldsymbol{A}} = \bar{\boldsymbol{Q}}\bar{\boldsymbol{R}}$.

Compute $\bar{\boldsymbol{A}} = (\boldsymbol{A}, \bar{\boldsymbol{a}})$
Compute $\begin{pmatrix} \boldsymbol{v} \\ \hline \boldsymbol{w} \end{pmatrix} = \boldsymbol{Q}^T\bar{\boldsymbol{a}}$, where $\boldsymbol{v} \in \mathbb{R}^m$ and $\boldsymbol{w} \in \mathbb{R}^{n-m}$.
Compute the Givens rotation matrix $\tilde{\boldsymbol{Q}}$ such that: $\tilde{\boldsymbol{Q}}\begin{pmatrix} \boldsymbol{v} \\ \hline \boldsymbol{w} \end{pmatrix} = \begin{pmatrix} \boldsymbol{v} \\ \hline \gamma \\ \hline \boldsymbol{0} \end{pmatrix}$,
where $\gamma \in \mathbb{R}$.
Compute $\bar{\boldsymbol{R}} = \tilde{\boldsymbol{Q}}\boldsymbol{Q}^T\bar{\boldsymbol{A}}$
Compute $\bar{\boldsymbol{Q}} = \boldsymbol{Q}\tilde{\boldsymbol{Q}}^T$

Removing a Constraint

To begin with we take a close look at the situation before the column is removed. $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ and its QR-factorization $\boldsymbol{Q} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{R} \in \mathbb{R}^{m\times m}$ have the following relationship

$$\boldsymbol{A} = \big((\boldsymbol{a}_1 \dots \boldsymbol{a}_{i-1}) \quad \boldsymbol{a}_i \quad (\boldsymbol{a}_{i+1} \dots \boldsymbol{a}_m)\big)\,, \tag{3.24}$$

where $(\boldsymbol{a}_1 \dots \boldsymbol{a}_{i-1})$ are the first $i-1$ columns, $\boldsymbol{a}_i$ is the i^{th} column and $(\boldsymbol{a}_{i+1} \dots \boldsymbol{a}_m)$ are the last $m-i$ columns. The QR-factorization of $\boldsymbol{A}$ is

$$\boldsymbol{A} = \boldsymbol{Q}\begin{pmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{pmatrix}, \tag{3.25}$$

and as $\boldsymbol{Q}$ is orthogonal we have

$$\boldsymbol{Q}^T\boldsymbol{A} = \begin{pmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{pmatrix}. \tag{3.26}$$

From (3.24) and (3.26) we thus have

$$\boldsymbol{Q}^T\boldsymbol{A} = \boldsymbol{Q}^T\left((\boldsymbol{a}_1 \dots \boldsymbol{a}_{i-1}) \quad \boldsymbol{a}_i \quad (\boldsymbol{a}_{i+1} \dots \boldsymbol{a}_m)\right), \tag{3.27}$$

which is equivalent to

$$\boldsymbol{Q}^T\boldsymbol{A} = \left(\boldsymbol{Q}^T(\boldsymbol{a}_1 \dots \boldsymbol{a}_{i-1}) \quad \boldsymbol{Q}^T\boldsymbol{a}_i \quad \boldsymbol{Q}^T(\boldsymbol{a}_{i+1} \dots \boldsymbol{a}_m)\right). \tag{3.28}$$

Using expression (3.26) and (3.28) gives

$$
\begin{aligned}
\boldsymbol{Q}^T\boldsymbol{A} &= \left(\begin{array}{c|c|c}
\boldsymbol{R}_{11} & \boldsymbol{R}_{12} & \boldsymbol{R}_{13} \\ \hline
\mathbf{0} & \boldsymbol{R}_{22} & \boldsymbol{R}_{23} \\ \hline
\mathbf{0} & \mathbf{0} & \boldsymbol{R}_{33} \\ \hline
\mathbf{0} & \mathbf{0} & \mathbf{0}
\end{array}\right) \\
&= \left(\begin{array}{ccc|c|ccc}
x_{(1,1)} & \cdots & x_{(1,i-1)} & x_{(1,i)} & x_{(1,i+1)} & \cdots & x_{(1,m)} \\
 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
 & & x_{(i-1,i-1)} & \cdots & \cdots & \cdots & \cdots \\ \hline
\multicolumn{3}{c|}{\mathbf{0}} & x_{(i,i)} & \cdots & \cdots & \cdots \\ \hline
 & & & & x_{(i+1,i+1)} & \cdots & \cdots \\
\multicolumn{3}{c|}{\mathbf{0}} & \mathbf{0} & & & \cdots \\
 & & & & & & x_{(m,m)} \\ \hline
\multicolumn{3}{c|}{\mathbf{0}} & \mathbf{0} & \multicolumn{3}{c}{\mathbf{0}}
\end{array}\right).
\end{aligned}
\tag{3.29}
$$

Removing the column of index i changes the constraint matrix $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ to

$$
\bar{\boldsymbol{A}} = \big((\boldsymbol{a}_1 \ldots \boldsymbol{a}_{i-1}) \quad (\boldsymbol{a}_{i+1} \ldots \boldsymbol{a}_m)\big)\,, \tag{3.30}
$$

where $\bar{\boldsymbol{A}} \in \mathbb{R}^{n\times(m-1)}$. Replacing $\boldsymbol{A}$ with $\bar{\boldsymbol{A}}$ in (3.27) gives

$$
\boldsymbol{Q}^T\bar{\boldsymbol{A}} = \boldsymbol{Q}^T\big((\boldsymbol{a}_1 \ldots \boldsymbol{a}_{i-1}) \quad (\boldsymbol{a}_{i+1} \ldots \boldsymbol{a}_m)\big)\,, \tag{3.31}
$$

which is equivalent to

$$
\boldsymbol{Q}^T\bar{\boldsymbol{A}} = \big(\boldsymbol{Q}^T(\boldsymbol{a}_1 \ldots \boldsymbol{a}_{i-1}) \quad \boldsymbol{Q}^T(\boldsymbol{a}_{i+1} \ldots \boldsymbol{a}_m)\big)\,. \tag{3.32}
$$

Together expression (3.28), (3.29) and (3.32) indicate that

$$
\boldsymbol{Q}^T\bar{\boldsymbol{A}} = \left(\begin{array}{c|c}
\boldsymbol{R}_{11} & \boldsymbol{R}_{13} \\ \hline
\mathbf{0} & \boldsymbol{R}_{23} \\ \hline
\mathbf{0} & \boldsymbol{R}_{33} \\ \hline
\mathbf{0} & \mathbf{0}
\end{array}\right) = \left(\begin{array}{ccc|ccc}
x & \cdots & x & x & \cdots & x \\
 & \cdots & \cdots & \cdots & \cdots & \cdots \\
 & & x & x & \cdots & x \\ \hline
\multicolumn{3}{c|}{\mathbf{0}} & x & \cdots & x \\ \hline
 & & & x & \cdots & x \\
\multicolumn{3}{c|}{\mathbf{0}} & & \cdots & \cdots \\
 & & & & & x \\ \hline
\multicolumn{3}{c|}{\mathbf{0}} & \multicolumn{3}{c}{\mathbf{0}}
\end{array}\right). \tag{3.33}
$$

The triangular structure is obviously violated and in order to regain it, Givens rotations are used. It is only the upper Hessenberg matrix $(\boldsymbol{R}_{23}, \boldsymbol{R}_{33})^T$, that needs to be made triangular. This is done using a sequence of $m - i$ Givens rotation matrices $\tilde{\boldsymbol{Q}} \in \mathbb{R}^{n \times n}$

$$\tilde{\boldsymbol{Q}} \begin{pmatrix} \boldsymbol{R}_{13} \\ \hline \boldsymbol{R}_{23} \\ \hline \boldsymbol{R}_{33} \\ \hline \boldsymbol{0} \end{pmatrix} = \tilde{\boldsymbol{Q}} \left(\begin{array}{ccc} x & \dots & x \\ \dots & \dots & \dots \\ x & \dots & x \\ \hline x & \dots & x \\ \hline x & \dots & x \\ & \dots & \dots \\ & & x \\ \hline & \boldsymbol{0} & \end{array} \right) = \left(\begin{array}{cccc} x & \dots & \dots & x \\ \dots & \dots & \dots & \dots \\ x & \dots & \dots & x \\ \hline m & \dots & \dots & m \\ \hline & m & \dots & m \\ & & \dots & \dots \\ & & & m \\ & & & 0 \\ \hline & & \boldsymbol{0} & \end{array} \right). \tag{3.34}$$

This means, that the triangular matrix $\bar{\boldsymbol{R}}$ is found from the product of $\tilde{\boldsymbol{Q}}$ and $\boldsymbol{Q}^T \bar{\boldsymbol{A}}$, so that we get

$$\begin{pmatrix} \bar{\boldsymbol{R}} \\ \boldsymbol{0} \end{pmatrix} = \tilde{\boldsymbol{Q}} \boldsymbol{Q}^T \bar{\boldsymbol{A}} = \left(\begin{array}{ccc|cccc} x & \dots & x & x & \dots & \dots & x \\ & \dots & \dots & \dots & \dots & \dots & \dots \\ & & x & x & \dots & \dots & x \\ \hline & \boldsymbol{0} & & m & \dots & \dots & m \\ \hline & & & & m & \dots & m \\ & \boldsymbol{0} & & & & \dots & \dots \\ & & & & & & m \\ \hline & \boldsymbol{0} & & & & \boldsymbol{0} & \end{array} \right). \tag{3.35}$$

Now that we have found the upper triangular matrix $\bar{\boldsymbol{R}}$ of the new factorization $\bar{\boldsymbol{A}} = \bar{\boldsymbol{Q}} \bar{\boldsymbol{R}}$, we only need to find the orthogonal matrix $\bar{\boldsymbol{Q}}$. As $\tilde{\boldsymbol{Q}}$ and $\boldsymbol{Q}^T$ are orthogonal (3.35) can be reformulated as

$$\bar{\boldsymbol{A}} = \boldsymbol{Q} \tilde{\boldsymbol{Q}}^T \bar{\boldsymbol{R}}, \tag{3.36}$$

which means that

$$\bar{\boldsymbol{Q}} = \boldsymbol{Q} \tilde{\boldsymbol{Q}}^T. \tag{3.37}$$

The updating procedure, when removing a constraint from the constraint matrix is summarized in algorithm 3.2.2.

Algorithm 3.2.2: Updating the QR-Factorization, when removing a column.
Note: Having $\boldsymbol{A} \in \mathbb{R}^{n \times m}$ and its QR factorization, where $\boldsymbol{Q} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{R} \in \mathbb{R}^{m \times m}$. Removing a column $\boldsymbol{c}$ from matrix $\boldsymbol{A}$ gives a new matrix $\bar{\boldsymbol{A}}$. The new factorization is $\bar{\boldsymbol{A}} = \bar{\boldsymbol{Q}}\bar{\boldsymbol{R}}$.

Compute $\bar{\boldsymbol{A}}$ by removing $\boldsymbol{c}$ from $\boldsymbol{A}$
Compute $\boldsymbol{P} = \boldsymbol{Q}^T \bar{\boldsymbol{A}}$
Compute the Givens rotation matrix $\tilde{\boldsymbol{Q}}$ such that: $\tilde{\boldsymbol{Q}}\boldsymbol{P}$ is upper triangular
Compute $\bar{\boldsymbol{R}} = \tilde{\boldsymbol{Q}}\boldsymbol{P}$
Compute $\bar{\boldsymbol{Q}} = \boldsymbol{Q}\tilde{\boldsymbol{Q}}^T$

3.3 Updating the Cholesky factorization

The matrix $\boldsymbol{A}^T\boldsymbol{G}^{-1}\boldsymbol{A} = \boldsymbol{H} \in \mathbb{R}^{m\times m}$, derived through (2.9) on page 7 and (2.13) on page 8, is both symmetric and positive definite. Therefore it has the Cholesky factorization $\boldsymbol{H} = \boldsymbol{L}\boldsymbol{L}^T$, where $\boldsymbol{L} \in \mathbb{R}^{m\times m}$ is lower triangular. This section is based on the work of Dennis and Schnabel [6], Gill *et al.* [7] and Golub and Van Loan [4], and presents the updating procedure of the Cholesky factorization to be employed, when appending or removing a constraint from constraint matrix $\boldsymbol{A}$.

Appending a Constraint

When a constraint is appended to constraint matrix $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ at column $m+1$, the matrix $\boldsymbol{H} \in \mathbb{R}^{m\times m}$ becomes

$$\bar{\boldsymbol{H}} = \begin{pmatrix} \boldsymbol{H} & \boldsymbol{q} \\ \boldsymbol{q}^T & r \end{pmatrix}, \tag{3.38}$$

where $\bar{\boldsymbol{H}} \in \mathbb{R}^{(m+1)\times(m+1)}$, $\boldsymbol{q} \in \mathbb{R}^m$ and $r \in \mathbb{R}$. The new Cholesky factorization is

$$\bar{\boldsymbol{H}} = \bar{\boldsymbol{L}}\bar{\boldsymbol{L}}^T, \quad \bar{\boldsymbol{L}} = \begin{pmatrix} \tilde{\boldsymbol{L}} & \boldsymbol{0} \\ \boldsymbol{s}^T & t \end{pmatrix}, \tag{3.39}$$

where $\bar{\boldsymbol{L}} \in \mathbb{R}^{(m+1)\times(m+1)}$, $\tilde{\boldsymbol{L}} \in \mathbb{R}^{m\times m}$, $\boldsymbol{s} \in \mathbb{R}^m$ and $t \in \mathbb{R}$. Together (3.38) and (3.39) give

$$\begin{aligned} \bar{\boldsymbol{H}} &= \begin{pmatrix} \boldsymbol{H} & \boldsymbol{q} \\ \boldsymbol{q}^T & r \end{pmatrix} \\ &= \begin{pmatrix} \tilde{\boldsymbol{L}} & \boldsymbol{0} \\ \boldsymbol{s}^T & t \end{pmatrix} \begin{pmatrix} \tilde{\boldsymbol{L}}^T & \boldsymbol{s} \\ \boldsymbol{0} & t \end{pmatrix} \\ &= \begin{pmatrix} \tilde{\boldsymbol{L}}\tilde{\boldsymbol{L}}^T & \tilde{\boldsymbol{L}}\boldsymbol{s} \\ \boldsymbol{s}^T\tilde{\boldsymbol{L}}^T & \boldsymbol{s}^T\boldsymbol{s} + t^2 \end{pmatrix}. \end{aligned} \tag{3.40}$$

From this expression $\bar{\boldsymbol{L}}$ can be found via $\tilde{\boldsymbol{L}}$, $\boldsymbol{s}$ and t. Furthermore from (3.40) and the fact, that $\boldsymbol{H} = \boldsymbol{L}\boldsymbol{L}^T$, we have

$$\boldsymbol{H} = \tilde{\boldsymbol{L}}\tilde{\boldsymbol{L}}^T = \boldsymbol{L}\boldsymbol{L}^T, \tag{3.41}$$

which means that

$$\tilde{\boldsymbol{L}} = \boldsymbol{L}. \tag{3.42}$$

From (3.40) and (3.42) we know that $\boldsymbol{s}$ can be found from the expression

$$\boldsymbol{q} = \tilde{\boldsymbol{L}}\boldsymbol{s} = \boldsymbol{L}\boldsymbol{s}, \tag{3.43}$$

and from (3.40) we also have

$$r = \boldsymbol{s}^T\boldsymbol{s} + t^2. \tag{3.44}$$

On this basis t can be found as

$$t = \sqrt{r - \boldsymbol{s}^T\boldsymbol{s}}. \tag{3.45}$$

Now $\tilde{\boldsymbol{L}}$, $\boldsymbol{s}$ and t have been isolated, and the new Cholesky factorization has been shown to be easily found from (3.39). Algorithm 3.3.1 summarizes how the updating procedure of the Cholesky factorization is carried out, when appending a column to constraint matrix.

Algorithm 3.3.1: Updating the Cholesky factorization when appending a column.
Note: The constraint matrix is $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ and the corresponding matrix $\boldsymbol{A}^T\boldsymbol{G}^{-1}\boldsymbol{A} = \boldsymbol{H} \in \mathbb{R}^{m\times m}$ has the Cholesky factorization $\boldsymbol{L}\boldsymbol{L}^T$, where $\boldsymbol{L} \in \mathbb{R}^{m\times m}$. Appending a column $\boldsymbol{c}$ to matrix $\boldsymbol{A}$ at index $m+1$ changes $\boldsymbol{H}$ into $\bar{\boldsymbol{H}} \in \mathbb{R}^{(m+1)\times(m+1)}$. The new Cholesky factorization is $\bar{\boldsymbol{H}} = \bar{\boldsymbol{L}}\bar{\boldsymbol{L}}^T$.

Let $\boldsymbol{p}$ be the last column of $\bar{\boldsymbol{H}}$
Let $\boldsymbol{q}$ be $\boldsymbol{p}$ except the last element
Let r be the last element of $\boldsymbol{p}$
Solve for $\boldsymbol{s}$ in $\boldsymbol{q} = \boldsymbol{L}\boldsymbol{s}$
Solve for t in $r = \boldsymbol{s}^T\boldsymbol{s} + t^2$
Compute $\bar{\boldsymbol{L}} = \begin{pmatrix} \boldsymbol{L} & \boldsymbol{0} \\ \boldsymbol{s}^T & t \end{pmatrix}$

Removing a Constraint

Before removing a constraint, i.e. the i^{th} column, from the constraint matrix $\boldsymbol{A} \in \mathbb{R}^{n\times m}$, the matrix $\boldsymbol{H} \in \mathbb{R}^{m\times m}$ can be formulated as

$$\boldsymbol{H} = \begin{pmatrix} \boldsymbol{H}_{11} & \boldsymbol{a} & \boldsymbol{H}_{12} \\ \boldsymbol{a}^T & c & \boldsymbol{b}^T \\ \boldsymbol{H}_{12}^T & \boldsymbol{b} & \boldsymbol{H}_{22} \end{pmatrix}, \tag{3.46}$$

where $\boldsymbol{H}_{11} \in \mathbb{R}^{(i-1)\times(i-1)}$, $\boldsymbol{H}_{22} \in \mathbb{R}^{(m-i)\times(m-i)}$, $\boldsymbol{H}_{12} \in \mathbb{R}^{(i-1)\times(m-i)}$, $\boldsymbol{a} \in \mathbb{R}^{(i-1)}$, $\boldsymbol{b} \in \mathbb{R}^{(m-i)}$ and $c \in \mathbb{R}$. The matrix $\boldsymbol{H}$ is Cholesky factorized as follows

$$\boldsymbol{H} = \boldsymbol{L}\boldsymbol{L}^T, \quad \boldsymbol{L} = \begin{pmatrix} \boldsymbol{L}_{11} & & \\ \boldsymbol{d}^T & e & \\ \boldsymbol{L}_{12} & \boldsymbol{f} & \boldsymbol{L}_{22} \end{pmatrix}, \tag{3.47}$$

where $\boldsymbol{L} \in \mathbb{R}^{m\times m}$, $\boldsymbol{L}_{11} \in \mathbb{R}^{(i-1)\times(i-1)}$ and $\boldsymbol{L}_{22} \in \mathbb{R}^{(m-i)\times(m-i)}$ are lower triangular and non-singular matrices with positive diagonal-entries. Also having $\boldsymbol{L}_{12} \in \mathbb{R}^{(m-i)\times(i-1)}$. The vectors $\boldsymbol{d}$ and $\boldsymbol{f}$ have dimensions $\mathbb{R}^{(i-1)}$ and $\mathbb{R}^{(m-i)}$ respectively and $e \in \mathbb{R}$. The column $(\boldsymbol{a}^T\, c\, \boldsymbol{b}^T)^T$ and the row $(\boldsymbol{a}^T\, c\, \boldsymbol{b}^T)$ in (3.46) are removed, when the constraint at column i is removed from $\boldsymbol{A}$. This gives us

$\bar{\boldsymbol{H}} \in \mathbb{R}^{(m-1)\times(m-1)}$, which is both symmetric and positive definite

$$\bar{\boldsymbol{H}} = \begin{pmatrix} \boldsymbol{H}_{11} & \boldsymbol{H}_{12} \\ \boldsymbol{H}_{12}^T & \boldsymbol{H}_{22} \end{pmatrix}. \tag{3.48}$$

This matrix has the following Cholesky factorization

$$\bar{\boldsymbol{H}} = \bar{\boldsymbol{L}}\bar{\boldsymbol{L}}^T, \quad \bar{\boldsymbol{L}} = \begin{pmatrix} \bar{\boldsymbol{L}}_{11} & \\ \bar{\boldsymbol{L}}_{12} & \bar{\boldsymbol{L}}_{22} \end{pmatrix}, \tag{3.49}$$

which is equivalent to

$$\begin{aligned} \bar{\boldsymbol{H}} &= \begin{pmatrix} \bar{\boldsymbol{L}}_{11} & \\ \bar{\boldsymbol{L}}_{12} & \bar{\boldsymbol{L}}_{22} \end{pmatrix} \begin{pmatrix} \bar{\boldsymbol{L}}_{11}^T & \bar{\boldsymbol{L}}_{12}^T \\ & \bar{\boldsymbol{L}}_{22}^T \end{pmatrix} \\ &= \begin{pmatrix} \bar{\boldsymbol{L}}_{11}\bar{\boldsymbol{L}}_{11}^T & \bar{\boldsymbol{L}}_{11}\bar{\boldsymbol{L}}_{12}^T \\ \bar{\boldsymbol{L}}_{12}\bar{\boldsymbol{L}}_{11}^T & \bar{\boldsymbol{L}}_{12}\bar{\boldsymbol{L}}_{12}^T + \bar{\boldsymbol{L}}_{22}\bar{\boldsymbol{L}}_{22}^T \end{pmatrix}, \end{aligned} \tag{3.50}$$

where $\bar{\boldsymbol{H}} \in \mathbb{R}^{(m-1)\times(m-1)}$, $\bar{\boldsymbol{L}}_{11} \in \mathbb{R}^{(i-1)\times(i-1)}$ and $\bar{\boldsymbol{L}}_{22} \in \mathbb{R}^{(m-i)\times(m-i)}$ are lower triangular, non-singular matrices with positive diagonal entries. Matrix $\bar{\boldsymbol{L}}_{12}$ is of dimension $\mathbb{R}^{(m-i)\times(i-1)}$.

From (3.46) and (3.47) we then have

$$\begin{aligned} \begin{pmatrix} \boldsymbol{H}_{11} & \boldsymbol{a} & \boldsymbol{H}_{12} \\ \boldsymbol{a}^T & c & \boldsymbol{b}^T \\ \boldsymbol{H}_{12}^T & \boldsymbol{b} & \boldsymbol{H}_{22} \end{pmatrix} &= \boldsymbol{H} \\ &= \boldsymbol{L}\boldsymbol{L}^T \\ &= \begin{pmatrix} \boldsymbol{L}_{11} & & \\ \boldsymbol{d}^T & e & \\ \boldsymbol{L}_{12} & \boldsymbol{f} & \boldsymbol{L}_{22} \end{pmatrix} \begin{pmatrix} \boldsymbol{L}_{11}^T & \boldsymbol{d} & \boldsymbol{L}_{12}^T \\ & e & \boldsymbol{f}^T \\ & & \boldsymbol{L}_{22}^T \end{pmatrix}, \end{aligned} \tag{3.51}$$

which gives

$$\begin{pmatrix} \boldsymbol{H}_{11} & \boldsymbol{a} & \boldsymbol{H}_{12} \\ \boldsymbol{a}^T & c & \boldsymbol{b}^T \\ \boldsymbol{H}_{12}^T & \boldsymbol{b} & \boldsymbol{H}_{22} \end{pmatrix} =$$
$$\begin{pmatrix} \boldsymbol{L}_{11}\boldsymbol{L}_{11}^T & \boldsymbol{L}_{11}\boldsymbol{d} & \boldsymbol{L}_{11}\boldsymbol{L}_{12}^T \\ \boldsymbol{d}^T\boldsymbol{L}_{11}^T & \boldsymbol{d}^T\boldsymbol{d} + e^2 & \boldsymbol{d}^T\boldsymbol{L}_{12}^T + e\boldsymbol{f}^T \\ \boldsymbol{L}_{12}\boldsymbol{L}_{11}^T & \boldsymbol{L}_{12}\boldsymbol{d} + \boldsymbol{f}e & \boldsymbol{L}_{12}\boldsymbol{L}_{12}^T + \boldsymbol{f}\boldsymbol{f}^T + \boldsymbol{L}_{22}\boldsymbol{L}_{22}^T \end{pmatrix}. \tag{3.52}$$

From (3.48) and (3.50) we know that

$$\begin{pmatrix} \boldsymbol{H}_{11} & \boldsymbol{H}_{12} \\ \boldsymbol{H}_{12}^T & \boldsymbol{H}_{22} \end{pmatrix} = \begin{pmatrix} \bar{\boldsymbol{L}}_{11}\bar{\boldsymbol{L}}_{11}^T & \bar{\boldsymbol{L}}_{11}\bar{\boldsymbol{L}}_{12}^T \\ \bar{\boldsymbol{L}}_{12}\bar{\boldsymbol{L}}_{11}^T & \bar{\boldsymbol{L}}_{12}\bar{\boldsymbol{L}}_{12}^T + \bar{\boldsymbol{L}}_{22}\bar{\boldsymbol{L}}_{22}^T \end{pmatrix}. \tag{3.53}$$

Expressions (3.52) and (3.53) give

$$\boldsymbol{H}_{11} = \boldsymbol{L}_{11}\boldsymbol{L}_{11}^T = \bar{\boldsymbol{L}}_{11}\bar{\boldsymbol{L}}_{11}^T, \tag{3.54}$$

and

$$\boldsymbol{H}_{12} = \boldsymbol{L}_{11}\boldsymbol{L}_{12}^T = \bar{\boldsymbol{L}}_{11}\bar{\boldsymbol{L}}_{12}^T, \tag{3.55}$$

which means that

$$\bar{\boldsymbol{L}}_{11} = \boldsymbol{L}_{11} \quad \text{and} \quad \bar{\boldsymbol{L}}_{12} = \boldsymbol{L}_{12}. \tag{3.56}$$

From (3.52) and (3.53) we also get

$$\boldsymbol{H}_{22} = \boldsymbol{L}_{12}\boldsymbol{L}_{12}^T + \boldsymbol{f}\boldsymbol{f}^T + \boldsymbol{L}_{22}\boldsymbol{L}_{22}^T = \bar{\boldsymbol{L}}_{12}\bar{\boldsymbol{L}}_{12}^T + \bar{\boldsymbol{L}}_{22}\bar{\boldsymbol{L}}_{22}^T, \tag{3.57}$$

and together with (3.56) this gives

$$\boldsymbol{H}_{22} = \boldsymbol{L}_{12}\boldsymbol{L}_{12}^T + \boldsymbol{f}\boldsymbol{f}^T + \boldsymbol{L}_{22}\boldsymbol{L}_{22}^T = \boldsymbol{L}_{12}\boldsymbol{L}_{12}^T + \bar{\boldsymbol{L}}_{22}\bar{\boldsymbol{L}}_{22}^T, \tag{3.58}$$

which is equivalent to

$$\boldsymbol{H}_{22} = \boldsymbol{f}\boldsymbol{f}^T + \boldsymbol{L}_{22}\boldsymbol{L}_{22}^T = \bar{\boldsymbol{L}}_{22}\bar{\boldsymbol{L}}_{22}^T. \tag{3.59}$$

From this expression we get

$$\bar{\boldsymbol{L}}_{22}\bar{\boldsymbol{L}}_{22}^T = (\boldsymbol{f}\ \boldsymbol{L}_{22})(\boldsymbol{f}\ \boldsymbol{L}_{22})^T. \tag{3.60}$$

From (3.47) we know that $(\boldsymbol{f}\ \boldsymbol{L}_{22})$ is not triangular. Therefore we now construct a sequence of Givens rotations $\tilde{\boldsymbol{Q}} \in \mathbb{R}^{(m-i+1)\times(m-i+1)}$ so that

$$\begin{aligned}
(\boldsymbol{f}\ \boldsymbol{L}_{22})\tilde{\boldsymbol{Q}} &= \left(\begin{array}{c|ccccc} x & x & & & \\ \vdots & \vdots & x & & \\ \vdots & \vdots & \vdots & \ddots & \\ x & x & x & \dots & x \end{array}\right)\tilde{\boldsymbol{Q}} \\
&= \left(\begin{array}{c|cccc} x & & & & \\ \vdots & x & & & \\ \vdots & \vdots & \ddots & & \\ x & x & \dots & x & 0 \end{array}\right) \\
&= (\tilde{\boldsymbol{L}}\ \ \boldsymbol{0}),
\end{aligned} \tag{3.61}$$

where $\tilde{\boldsymbol{L}}$ is lower triangular. As the Givens rotation matrix $\tilde{\boldsymbol{Q}}$ is orthogonal, we have that $\tilde{\boldsymbol{Q}}\tilde{\boldsymbol{Q}}^T = \boldsymbol{I}$, and therefore we can reformulate (3.60) as

$$\bar{\boldsymbol{L}}_{22}\bar{\boldsymbol{L}}_{22}^T = (\boldsymbol{f}\ \boldsymbol{L}_{22})\tilde{\boldsymbol{Q}}\tilde{\boldsymbol{Q}}^T(\boldsymbol{f}\ \boldsymbol{L}_{22})^T, \tag{3.62}$$

which is equivalent to

$$\bar{\boldsymbol{L}}_{22}\bar{\boldsymbol{L}}_{22}^T = ((\boldsymbol{f}\ \boldsymbol{L}_{22})\tilde{\boldsymbol{Q}})((\boldsymbol{f}\ \boldsymbol{L}_{22})\tilde{\boldsymbol{Q}})^T, \tag{3.63}$$

and according to (3.61) this renders

$$\bar{\boldsymbol{L}}_{22}\bar{\boldsymbol{L}}_{22}^T = (\tilde{\boldsymbol{L}}\ \ \boldsymbol{0})(\tilde{\boldsymbol{L}}\ \ \boldsymbol{0})^T = \tilde{\boldsymbol{L}}\tilde{\boldsymbol{L}}^T. \tag{3.64}$$

Finally we now know that

$$\bar{\boldsymbol{L}}_{22} = \tilde{\boldsymbol{L}}, \tag{3.65}$$

which means that $\bar{\boldsymbol{L}}_{22}$ may be constructed as

$$(\bar{\boldsymbol{L}}_{22} \;\; \boldsymbol{0}) = (\tilde{\boldsymbol{L}} \;\; \boldsymbol{0}) = (\boldsymbol{f} \;\; \boldsymbol{L}_{22})\tilde{\boldsymbol{Q}}. \tag{3.66}$$

Hence we now have everything for constructing the new Cholesky factorization:

$$\bar{\boldsymbol{H}} = \bar{\boldsymbol{L}}\bar{\boldsymbol{L}}^T, \quad \bar{\boldsymbol{L}} = \begin{pmatrix} \bar{\boldsymbol{L}}_{11} & \\ \bar{\boldsymbol{L}}_{12} & \bar{\boldsymbol{L}}_{22} \end{pmatrix}. \tag{3.67}$$

Algorithm 3.3.2 summarizes the updating procedure of the Cholesky factorization, when a column is removed from the constraint matrix.

Algorithm 3.3.2: Updating the Cholesky factorization when removing a column.
Note: Having the constraint matrix $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ and the corresponding matrix $\boldsymbol{A}^T\boldsymbol{G}^{-1}\boldsymbol{A} = \boldsymbol{H} \in \mathbb{R}^{m\times m}$ with the Cholesky factorization $\boldsymbol{L}\boldsymbol{L}^T$, where $\boldsymbol{L} \in \mathbb{R}^{m\times m}$. Removing column $\boldsymbol{c}$ from matrix $\boldsymbol{A}$ at index i changes $\boldsymbol{H}$ into $\bar{\boldsymbol{H}} \in \mathbb{R}^{(m-1)\times(m-1)}$. The new Cholesky factorization is $\bar{\boldsymbol{H}} = \bar{\boldsymbol{L}}\bar{\boldsymbol{L}}^T$.

Let $\boldsymbol{L} = \begin{pmatrix} \boldsymbol{L}_{11} & & \\ \boldsymbol{d}^T & e & \\ \boldsymbol{L}_{12} & \boldsymbol{f} & \boldsymbol{L}_{22} \end{pmatrix}$, where $(\boldsymbol{d}^T \;\; e)$ is the row at index i and $(e \;\; \boldsymbol{f}^T)^T$ is the column at index i.
Let $\bar{\boldsymbol{L}}_{11} = \boldsymbol{L}_{11}$.
Let $\bar{\boldsymbol{L}}_{12} = \boldsymbol{L}_{12}$.
Let $\hat{\boldsymbol{L}} = (\boldsymbol{f} \;\; \boldsymbol{L}_{22})$
Compute the Givens rotation matrix $\tilde{\boldsymbol{Q}}$ such that $\hat{\boldsymbol{L}}\tilde{\boldsymbol{Q}}$=$(\bar{\boldsymbol{L}}_{22} \;\; \boldsymbol{0})$, where $\bar{\boldsymbol{L}}_{22}$ is triangular.
Compute $\bar{\boldsymbol{L}}$=$\begin{pmatrix} \bar{\boldsymbol{L}}_{11} & \boldsymbol{0} \\ \bar{\boldsymbol{L}}_{12} & \bar{\boldsymbol{L}}_{22} \end{pmatrix}$

Chapter 4

Active Set Methods

In this chapter we investigate how to solve an inequality constrained convex QP of type

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} \quad f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x} \tag{4.1a}$$

$$\text{s.t.} \quad c_i(\boldsymbol{x}) = \boldsymbol{a}_i\boldsymbol{x} - b_i \geq 0, \qquad i \in \mathcal{I}. \tag{4.1b}$$

The solution of this problem $\boldsymbol{x}^*$ is also the same as to the equality constrained convex QP

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} \quad f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x} \tag{4.2a}$$

$$\text{s.t.} \quad c_i(\boldsymbol{x}) = \boldsymbol{a}_i\boldsymbol{x} - b_i = 0, \qquad i \in \mathcal{A}(\boldsymbol{x}^*). \tag{4.2b}$$

In other words, this means that in order to find the optimal point we need to find the active set $\mathcal{A}(\boldsymbol{x}^*)$ of (4.1). As we shall see in the following, this is done by solving a sequence of equality constrained convex QP's. We will investigate two methods for solving (4.1), namely the primal active set method (section 4.1) and the dual active set method (section 4.3).

4.1 Primal Active Set Method

The primal active set method discussed in this section is based on the work of Gill and Murray [8] and Gill *et al.* [9]. The algorithm solves a convex QP with inequality constraints (4.1).

4.1.1 Survey

The inequality constrained QP is written on the form

$$\min_{\boldsymbol{x} \in \mathbb{R}^n} \quad \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x} \tag{4.3a}$$

$$\text{s.t.} \quad \boldsymbol{a}_i^T\boldsymbol{x} \geq b_i \qquad i \in \mathcal{I} \tag{4.3b}$$

where $\boldsymbol{G} \in \mathbb{R}^{n \times n}$ is symmetric and positive definite.

The objective function of the QP is given as

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x} \tag{4.4}$$

and the feasible region is

$$\Omega = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{a}_i^T\boldsymbol{x} \geq b_i, i \in \mathcal{I}\} \tag{4.5}$$

The idea of the primal active set method is to compute a feasible sequence $\{\boldsymbol{x}_k \in \Omega\}$, where $k = \mathbb{N}_0$, with decreasing value of the objective function, $f(\boldsymbol{x}_{k+1}) < f(\boldsymbol{x}_k)$. For each step in the sequence we solve an equality constraint QP

$$\min_{\boldsymbol{x}_k \in \mathbb{R}^n} \quad \frac{1}{2}\boldsymbol{x}_k^T\boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{g}^T\boldsymbol{x}_k \tag{4.6a}$$

$$\text{s.t.} \quad \boldsymbol{a}_i^T\boldsymbol{x}_k = b_i \qquad i \in \mathcal{A}(\boldsymbol{x}_k) \tag{4.6b}$$

where $\mathcal{A}(\boldsymbol{x}_k) = \{i \in \mathcal{I} : \boldsymbol{a}_i^T \boldsymbol{x}_k = b_i\}$ is the current active set. Because the vectors $\boldsymbol{a}_i$ are linearly independent for $i \in \mathcal{A}(\boldsymbol{x}_k)$, the strictly convex equality constrained QP can be solved by solving the corresponding KKT system using the range space or the null space procedure.

The sequence of equality constrained QP's is generated, so that the sequence $\{\boldsymbol{x}_k\}$ converges to the optimal point $\boldsymbol{x}^*$, where the following KKT conditions

$$\boldsymbol{G}\boldsymbol{x}^* + \boldsymbol{g} - \sum_{i \in I} \boldsymbol{a}_i \mu_i^* = \boldsymbol{0} \tag{4.7a}$$

$$\boldsymbol{a}_i^T \boldsymbol{x}^* = b_i \qquad i \in \mathcal{W}_k \tag{4.7b}$$

$$\boldsymbol{a}_i^T \boldsymbol{x}^* \geq b_i \qquad i \in \mathcal{I} \backslash \mathcal{W}_k \tag{4.7c}$$

$$\mu_i^* \geq 0 \qquad i \in \mathcal{W}_k \tag{4.7d}$$

$$\mu_i^* = 0 \qquad i \in \mathcal{I} \backslash \mathcal{W}_k \tag{4.7e}$$

are satisfied.

4.1.2 Improving Direction and Step Length

For every feasible point $\boldsymbol{x}_k \in \Omega$, we have a corresponding working set $\mathcal{W}_k$, which is a subset of the active set $\mathcal{A}(\boldsymbol{x}_k)$, $\mathcal{W}_k \subset \mathcal{A}(\boldsymbol{x}_k)$. $\mathcal{W}_k$ is selected so that the vectors $\boldsymbol{a}_i$, $i \in \mathcal{W}_k$, are linearly independent, which corresponds to full column rank of $\boldsymbol{A}_k = [\boldsymbol{a}_i]_{i \in \mathcal{W}_k}$.

If no constraints are active, $\boldsymbol{a}_i^T \boldsymbol{x}_0 > b_i$ for $i \in \mathcal{I}$, then the corresponding working set is empty, $\mathcal{W}_0 = \emptyset$. The objective function (4.4) is convex, and therefore if $\min_{\boldsymbol{x} \in \mathbb{R}} f(\boldsymbol{x}) \notin \Omega$, then one or more constraints will be violated, when $\boldsymbol{x}_k$ seeks the minimum of the objective function. This explains why the working set is never empty, once a constraint has become active, i.e. $\boldsymbol{a}_i^T \boldsymbol{x}_k = b_i$, $i \in \mathcal{W}_k \neq \emptyset$, $k \in \mathbb{N}$.

Improving Direction

The feasible sequence $\{\boldsymbol{x}_k \in \Omega\}$ with decreasing value, $f(\boldsymbol{x}_{k+1}) < f(\boldsymbol{x}_k)$, is generated by following the improving direction $\boldsymbol{p} \in \mathbb{R}^n$ such that $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{p}$. This leads us to

$$
\begin{aligned}
f(\boldsymbol{x}_{k+1}) &= f(\boldsymbol{x}_k + \boldsymbol{p}) \\
&= \frac{1}{2}(\boldsymbol{x}_k + \boldsymbol{p})^T \boldsymbol{G}(\boldsymbol{x}_k + \boldsymbol{p}) + \boldsymbol{g}^T(\boldsymbol{x}_k + \boldsymbol{p}) \\
&= \frac{1}{2}(\boldsymbol{x}_k^T \boldsymbol{G} + \boldsymbol{p}^T \boldsymbol{G})(\boldsymbol{x}_k + \boldsymbol{p}) + \boldsymbol{g}^T \boldsymbol{x}_k + \boldsymbol{g}^T \boldsymbol{p} \\
&= \frac{1}{2}(\boldsymbol{x}_k^T \boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{x}_k^T \boldsymbol{G}\boldsymbol{p} + \boldsymbol{p}^T \boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{p}^T \boldsymbol{G}\boldsymbol{p}) + \boldsymbol{g}^T \boldsymbol{x}_k + \boldsymbol{g}^T \boldsymbol{p} \\
&= \frac{1}{2}\boldsymbol{x}_k^T \boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{g}^T \boldsymbol{x}_k + (\boldsymbol{x}_k^T \boldsymbol{G} + \boldsymbol{g}^T)\boldsymbol{p} + \frac{1}{2}\boldsymbol{p}^T \boldsymbol{G}\boldsymbol{p} \\
&= f(\boldsymbol{x}_k) + (\boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{g})^T \boldsymbol{p} + \frac{1}{2}\boldsymbol{p}^T \boldsymbol{G}\boldsymbol{p} \\
&= f(\boldsymbol{x}_k) + \phi(\boldsymbol{p}). \qquad (4.8)
\end{aligned}
$$

and in order to satisfy $f(\boldsymbol{x}_{k+1}) < f(\boldsymbol{x}_k)$, the improving direction $\boldsymbol{p}$ must be computed so that $\phi(\boldsymbol{p}) < 0$ and $\bar{\boldsymbol{x}} = (\boldsymbol{x}_k + \boldsymbol{p}) \in \Omega$. Instead of computing the new optimal point $\bar{\boldsymbol{x}}$ directly, computational savings are achieved by only computing $\boldsymbol{p}$. When $\bar{\boldsymbol{x}} = \boldsymbol{x}_k + \boldsymbol{p}$, the constraint $\boldsymbol{a}_i^T \bar{\boldsymbol{x}} = b_i$ becomes

$$
\boldsymbol{a}_i^T \bar{\boldsymbol{x}} = \boldsymbol{a}_i^T(\boldsymbol{x}_k + \boldsymbol{p}) = \boldsymbol{a}_i^T \boldsymbol{x}_k + \boldsymbol{a}_i^T \boldsymbol{p} = b_i + \boldsymbol{a}_i^T \boldsymbol{p}, \qquad (4.9)
$$

so

$$
\boldsymbol{a}_i^T \boldsymbol{p} = 0 \qquad (4.10)
$$

and the objective function becomes

$$
f(\bar{\boldsymbol{x}}) = f(\boldsymbol{x}_k + \boldsymbol{p}) = f(\boldsymbol{x}_k) + \phi(\boldsymbol{p}). \qquad (4.11)
$$

So for the subspaces $\mathcal{M}_k = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{a}_i^T \boldsymbol{x} = b_i, i \in \mathcal{W}_k\}$ and $\mathcal{S}_k = \{\boldsymbol{p} \in \mathbb{R}^n : \boldsymbol{a}_i^T \boldsymbol{p} = 0, i \in \mathcal{W}_k\}$, we get

$$
\min_{\bar{\boldsymbol{x}} \in \mathcal{M}_k} f(\bar{\boldsymbol{x}}) = \min_{\boldsymbol{p} \in \mathcal{S}_k} f(\boldsymbol{x}_k + \boldsymbol{p}) = f(\boldsymbol{x}_k) + \min_{\boldsymbol{p} \in \mathcal{S}_k} \phi(\boldsymbol{p}) \qquad (4.12)
$$

and hereby we have the following relations

$$\bar{\boldsymbol{x}}^* = \boldsymbol{x}_k + \boldsymbol{p}^* \tag{4.13}$$
$$f(\bar{\boldsymbol{x}}^*) = f(\boldsymbol{x}_k) + \phi(\boldsymbol{p}^*). \tag{4.14}$$

For these relations $\bar{\boldsymbol{x}}^*$ and $f(\bar{\boldsymbol{x}}^*)$ respectively are the optimal solution and the optimal value of

$$\min_{\bar{\boldsymbol{x}} \in \mathbb{R}^n} \quad f(\bar{\boldsymbol{x}}) = \frac{1}{2}\bar{\boldsymbol{x}}^T \boldsymbol{G} \bar{\boldsymbol{x}} + \boldsymbol{g}^T \bar{\boldsymbol{x}} \tag{4.15a}$$
$$\text{s.t.} \quad \boldsymbol{a}_i^T \bar{\boldsymbol{x}} = b_i \qquad i \in \mathcal{W}_k \tag{4.15b}$$

and $\boldsymbol{p}^*$ and $\phi(\boldsymbol{p}^*)$ respectively are the optimal solution and the optimal value of

$$\min_{\boldsymbol{p} \in \mathbb{R}^n} \quad \phi(\boldsymbol{p}) = \frac{1}{2}\boldsymbol{p}^T \boldsymbol{G} \boldsymbol{p} + (\boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{g})^T \boldsymbol{p} \tag{4.16a}$$
$$\text{s.t.} \quad \boldsymbol{a}_i^T \boldsymbol{p} = 0 \qquad i \in \mathcal{W}_k. \tag{4.16b}$$

The right hand side of the constraints in (4.16) is zero,which is why it is easier to find the improving direction $\boldsymbol{p}$ than to solve (4.15). This means that the improving direction is found by solving (4.16).

Step Length

If we take the full step $\boldsymbol{p}$, we cannot be sure, that $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{p}$ is feasible. In this section we will therefore find the step length $\alpha \in \mathbb{R}$, which ensures feasibility.

If the optimal solution is $\boldsymbol{p}^* = \boldsymbol{0}$, then $\phi(\boldsymbol{p}^*) = 0$. And because (4.4) is strictly convex, then $\phi(\boldsymbol{p}^*) < \phi(\boldsymbol{0}) = 0$, if $\boldsymbol{p}^* \neq \boldsymbol{0}$, so

$$\phi(\boldsymbol{p}^*) = 0, \quad \boldsymbol{p}^* = \boldsymbol{0} \tag{4.17}$$
$$\phi(\boldsymbol{p}^*) < 0, \quad \boldsymbol{p}^* \in \{\boldsymbol{p} \in \mathbb{R}^n : \boldsymbol{a}_i^T \boldsymbol{p} = 0, i \in \mathcal{W}_k\} \backslash \{\boldsymbol{0}\}. \tag{4.18}$$

The relation between $f(\boldsymbol{x}_k + \alpha \boldsymbol{p}^*)$ and $\phi(\alpha \boldsymbol{p}^*)$ is

$$\begin{aligned}
f(\boldsymbol{x}_{k+1}) &= f(\boldsymbol{x}_k + \alpha \boldsymbol{p}) \\
&= \frac{1}{2}(\boldsymbol{x}_k + \alpha \boldsymbol{p})^T \boldsymbol{G}(\boldsymbol{x}_k + \alpha \boldsymbol{p}) + \boldsymbol{g}^T(\boldsymbol{x}_k + \alpha \boldsymbol{p}) \\
&= \frac{1}{2}(\boldsymbol{x}_k^T \boldsymbol{G} + \alpha \boldsymbol{p}^T \boldsymbol{G})(\boldsymbol{x}_k + \alpha \boldsymbol{p}) + \boldsymbol{g}^T \boldsymbol{x}_k + \boldsymbol{g}^T \alpha \boldsymbol{p} \\
&= \frac{1}{2}(\boldsymbol{x}_k^T \boldsymbol{G} \boldsymbol{x}_k + \boldsymbol{x}_k^T \boldsymbol{G} \alpha \boldsymbol{p} + \alpha \boldsymbol{p}^T \boldsymbol{G} \boldsymbol{x}_k + \alpha \boldsymbol{p}^T \boldsymbol{G} \alpha \boldsymbol{p}) \\
&\quad + \boldsymbol{g}^T \boldsymbol{x}_k + \boldsymbol{g}^T \alpha \boldsymbol{p} \\
&= \frac{1}{2}\boldsymbol{x}_k^T \boldsymbol{G} \boldsymbol{x}_k + \boldsymbol{g}^T \boldsymbol{x}_k + (\boldsymbol{x}_k^T \boldsymbol{G} + \boldsymbol{g}^T)\alpha \boldsymbol{p} + \frac{1}{2}\alpha \boldsymbol{p}^T \boldsymbol{G} \alpha \boldsymbol{p} \\
&= f(\boldsymbol{x}_k) + (\boldsymbol{G} \boldsymbol{x}_k + \boldsymbol{g})^T \alpha \boldsymbol{p} + \frac{1}{2}\alpha \boldsymbol{p}^T \boldsymbol{G} \alpha \boldsymbol{p} \\
&= f(\boldsymbol{x}_k) + \phi(\alpha \boldsymbol{p}). \qquad (4.19)
\end{aligned}$$

For $\boldsymbol{p}^* \neq \boldsymbol{0}$ we have

$$\begin{aligned}
f(\boldsymbol{x}_k + \alpha \boldsymbol{p}) &= f(\boldsymbol{x}_k + \alpha \boldsymbol{x}_k - \alpha \boldsymbol{x}_k + \alpha \boldsymbol{p}) \\
&= f((1-\alpha)\boldsymbol{x}_k + \alpha(\boldsymbol{x}_k + \boldsymbol{p})) \\
&\leq (1-\alpha) f(\boldsymbol{x}_k) + \alpha f(\boldsymbol{x}_k + \boldsymbol{p}) \\
&< (1-\alpha) f(\boldsymbol{x}_k) + \alpha f(\boldsymbol{x}_k) \\
&= f(\boldsymbol{x}_k) + \alpha f(\boldsymbol{x}_k) - \alpha f(\boldsymbol{x}_k) \\
&= f(\boldsymbol{x}_k) \qquad (4.20)
\end{aligned}$$

and because of the convexity of the objective function f, this is a fact for all $\alpha \in]0;1]$.

We now know, that if an $\alpha \in]0;1]$ exists, then $f(\boldsymbol{x}_k + \alpha \boldsymbol{p}^*) < f(\boldsymbol{x}_k)$. On this basis we want to find a point on the line segment $\boldsymbol{p}^* = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$, whereby the largest possible reduction of the objective function is achieved, and at the same time the constraints not in the current working set, i.e. $i \in \mathcal{I} \backslash \mathcal{W}_k$, remain satisfied. In other words, looking from point $\boldsymbol{x}_k$ in the improving direction $\boldsymbol{p}^*$, we would like to find an α, so that the point $\boldsymbol{x}_k + \alpha \boldsymbol{p}^*$ remains feasible. In this way the greatest reduction of the objective function is obtained by choosing the largest possible α without leaving the feasible region.

As we want to retain feasibility, we only need to consider the potentially violated constraints. This means the constraints not in the current working set satisfy

$$\boldsymbol{a}_i^T(\boldsymbol{x}_k + \alpha\boldsymbol{p}^*) \geq b_i, \quad i \in \mathcal{I}\backslash\mathcal{W}_k. \tag{4.21}$$

Since $\boldsymbol{x}_k \in \Omega$, we have

$$\alpha\boldsymbol{a}_i^T\boldsymbol{p}^* \geq b_i - \boldsymbol{a}_i^T\boldsymbol{x}_k \leq 0, \quad i \in \mathcal{I}\backslash\mathcal{W}_k, \tag{4.22}$$

and whenever $\boldsymbol{a}_i^T\boldsymbol{p}^* \geq 0$, this relation is satisfied for all $\alpha \geq 0$. As $b_i - \boldsymbol{a}_i^T\boldsymbol{x}_k \leq 0$, the relation can still be satisfied for $\boldsymbol{a}_i^T\boldsymbol{p}^* < 0$, if we consider an upper bound $0 \leq \alpha \leq \bar{\alpha}_i$, where

$$\bar{\alpha}_i = \frac{b_i - \boldsymbol{a}_i^T\boldsymbol{x}_k}{\boldsymbol{a}_i^T\boldsymbol{p}^*} \geq 0, \quad \boldsymbol{a}_i^T\boldsymbol{p}^* < 0, \quad i \in \mathcal{I}\backslash\mathcal{W}_k. \tag{4.23}$$

Whenever $\boldsymbol{a}_i^T\boldsymbol{x}_k = b_i$, and $\boldsymbol{a}_i^T\boldsymbol{p}^* < 0$ for $i \in \mathcal{I}\backslash\mathcal{W}_k$, we have $\bar{\alpha}_i = 0$. So $\bar{\boldsymbol{x}} = \boldsymbol{x}_k + \alpha\boldsymbol{p}^*$ will remain feasible, $\bar{\boldsymbol{x}} \in \Omega$, whenever $0 \leq \alpha \leq \min_{i\in\mathcal{I}\backslash\mathcal{W}_k} \bar{\alpha}_i$. In other words, the upper bound of α will be chosen in a way, that the nearest constraint not in the current working set will become active.

From the Lagrangian function of (4.16), we know by definition, that $\boldsymbol{p}^*$ satisfies

$$\begin{aligned} \boldsymbol{G}\boldsymbol{p}^* + (\boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{g}) - \boldsymbol{A}\boldsymbol{\mu}^* &= \boldsymbol{0} && \text{(4.24a)} \\ \boldsymbol{A}^T\boldsymbol{p}^* &= \boldsymbol{0}, && \text{(4.24b)} \end{aligned}$$

and by transposing and multiplying with $\boldsymbol{p}^*$ we get

$$\begin{aligned} (\boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{g})^T\boldsymbol{p}^* &= (\boldsymbol{A}\boldsymbol{\mu}^* - \boldsymbol{G}\boldsymbol{p}^*)^T\boldsymbol{p}^* \\ &= \boldsymbol{\mu}^{*T}\underbrace{\boldsymbol{A}^T\overset{*}{\boldsymbol{p}}}_{=0} - \boldsymbol{p}^{*T}\boldsymbol{G}\boldsymbol{p}^* \\ &= -\boldsymbol{p}^{*T}\boldsymbol{G}\boldsymbol{p}^*. \end{aligned} \tag{4.25}$$

From (4.19) and (4.25) we define the line search function $h(\alpha)$ as

$$\begin{aligned} h(\alpha) &= f(\boldsymbol{x}_k + \alpha \boldsymbol{p}) \\ &= f(\boldsymbol{x}_k) + \alpha(\boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{g})^T \boldsymbol{p} + \frac{1}{2}\alpha^2 \boldsymbol{p}^T \boldsymbol{G}\boldsymbol{p} \\ &= f(\boldsymbol{x}_k) - \alpha \boldsymbol{p}^{*T} \boldsymbol{G}\boldsymbol{p}^* + \frac{1}{2}\alpha^2 \boldsymbol{p}^{*T} \boldsymbol{G}\boldsymbol{p}^* \\ &= \frac{1}{2}\boldsymbol{p}^{*T} \boldsymbol{G}\boldsymbol{p}^* \alpha^2 - \boldsymbol{p}^{*T} \boldsymbol{G}\boldsymbol{p}^* \alpha + f(\boldsymbol{x}_k). \end{aligned} \tag{4.26}$$

If $\boldsymbol{p}^* \neq 0$ is the solution of (4.16), we have $\boldsymbol{p}^{*T}\boldsymbol{G}\boldsymbol{p}^* > 0$, as $\boldsymbol{G}$ is positive definite. So the line search function is a parabola with upward legs. The first order derivative is

$$\frac{dh}{d\alpha}(\alpha) = \boldsymbol{p}^{*T}\boldsymbol{G}\boldsymbol{p}^*\alpha - \boldsymbol{p}^{*T}\boldsymbol{G}\boldsymbol{p}^* \tag{4.27a}$$

$$= (\alpha - 1)\boldsymbol{p}^{*T}\boldsymbol{G}\boldsymbol{p}^*, \tag{4.27b}$$

which tells us, that the line search function has its minimum at $\frac{dh}{d\alpha}(1) = 0$. Therefore the largest possible reduction in the line search function (4.26) is achieved by selecting $\alpha \in [0;1]$ as large as possible. So the optimal solution of

$$\min_{\boldsymbol{\alpha}\in\mathbb{R}} \quad h(\alpha) = \frac{1}{2}\boldsymbol{p}^{*T}\boldsymbol{G}\boldsymbol{p}^*\alpha^2 - \boldsymbol{p}^{*T}\boldsymbol{G}\boldsymbol{p}^*\alpha + f(\boldsymbol{x}_k) \tag{4.28a}$$

$$\text{s.t.} \quad \boldsymbol{a}_i^T(\boldsymbol{x}_k + \alpha\boldsymbol{p}^*) \geq b_i \qquad i \in \mathcal{I} \tag{4.28b}$$

is

$$\begin{aligned} \alpha^* &= \min\left(1, \min_{i\in\mathcal{I}\backslash\mathcal{W}_k : \boldsymbol{a}_i^T\boldsymbol{p}^* < 0} \bar{\alpha}_i\right) \\ &= \min\left(1, \min_{i\in\mathcal{I}\backslash\mathcal{W}_k : \boldsymbol{a}_i^T\boldsymbol{p}^* < 0} \frac{b_i - \boldsymbol{a}_i^T\boldsymbol{x}_k}{\boldsymbol{a}_i^T\boldsymbol{p}^*}\right) \geq 0. \end{aligned} \tag{4.29}$$

The largest possible reduction in the objective function along the improving direction $\boldsymbol{p}^*$ is obtained by the new point $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha^*\boldsymbol{p}^*$.

4.1.3 Appending and Removing a Constraint

The largest possible reduction of the objective function in the affine space $\mathcal{M}_k = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{a}_i^T\boldsymbol{x} = b_i, i \in \mathcal{W}_k\}$ is obtained at point $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{p}^*$, i.e. by selecting $\alpha^* = 1$ and $\mathcal{W}_{k+1} = \mathcal{W}_k$. This point satisfies $f(\boldsymbol{x}_{k+1}) < f(\boldsymbol{x}_k)$, and since $\mathcal{W}_{k+1} = \mathcal{W}_k$, this point will also be the optimal solution in the affine space $\mathcal{M}_{k+1} = \mathcal{M}_k$, thus a new iterate will give $\boldsymbol{p}^* = \boldsymbol{0}$. So, in order to minimize the objective function further, we must update the working set for each iteration. This is done either by appending or removing a constraint from the current working set $\mathcal{W}_k$.

Appending a Constraint

If the point $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{p}^* \notin \Omega = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{a}_i^T\boldsymbol{x} \geq b_i, i \in \mathcal{I}\}$, then the point is not feasible with respect to one or more constraints not in the current working set $\mathcal{W}_k$. Therefore, by choosing the point $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha^*\boldsymbol{p}^* \in \mathcal{M}_k \cap \Omega$, where $\alpha^* \in [0; 1[$, feasibility is sustained and the largest possible reduction of the objective function is achieved. In other words, we have a blocking constraint with index $j \in \mathcal{I}\backslash\mathcal{W}_k$, such that $\boldsymbol{a}_j^T(\boldsymbol{x}_k + \alpha^*\boldsymbol{p}^*) = b_j$. So, by appending constraint j to the current working set, we get a new working set $\mathcal{W}_{k+1} = \mathcal{W}_k \cup \{j\}$ corresponding to $\boldsymbol{x}_{k+1}$, which is then a feasible point by construction.

The set of blocking constraints is defined as

$$\mathcal{J} = \arg \min_{i \in \mathcal{I}\backslash\mathcal{W}_k : \boldsymbol{a}_i^T\boldsymbol{p}^* < 0} \frac{b_i - \boldsymbol{a}_i^T\boldsymbol{x}_k}{\boldsymbol{a}_i^T\boldsymbol{p}^*} \tag{4.30}$$

The blocking constraint to be appended, is the most violated constraint. In other words, it is the violated constraint, found closest to the current point $\boldsymbol{x}_k$. As mentioned, the working set is updated as $\mathcal{W}_{k+1} = \mathcal{W}_k \cup \{j\}$, which means, that we append the vector $\boldsymbol{a}_j$, where $j \in \mathcal{J}$, to the current working set. The constraints in the current working set, i.e. the vectors $\boldsymbol{a}_i$ for which $i \in \mathcal{W}_k$, satisfies

$$\boldsymbol{a}_i^T\boldsymbol{p}^* = 0, \quad i \in \mathcal{W}_k. \tag{4.31}$$

If vector $\boldsymbol{a}_j$, where $j \in \mathcal{J}$, is linearly dependent of the constraints in the current working set, i.e. $\boldsymbol{a}_j \in \text{span}\{\boldsymbol{a}_i\}_{i\in\mathcal{W}_k}$, then we have

$$\exists \gamma_i \in \mathbb{R} : \boldsymbol{a}_j = \sum_{i\in\mathcal{W}_k} \gamma_i \boldsymbol{a}_i, \tag{4.32}$$

hence $\boldsymbol{a}_j$ must satisfy

$$\boldsymbol{a}_j^T \boldsymbol{p}^* = \sum_{i\in\mathcal{W}_k} \gamma_i \underbrace{(\boldsymbol{a}_i^T \boldsymbol{p}^*)}_{=0} = 0, \quad j \in \mathcal{I}\backslash\mathcal{W}_k. \tag{4.33}$$

But since we choose $j \in \mathcal{I}\backslash\mathcal{W}_k$, such that $\boldsymbol{a}_i^T \boldsymbol{p}^* < 0$, we have $\boldsymbol{a}_j \notin \text{span}\{\boldsymbol{a}_i\}_{i\in\mathcal{W}_k}$. So we are guaranteed, that the blocking constraint j is linearly independent of the constraints in the current working set, i.e. $(\boldsymbol{A}\ \boldsymbol{a}_j)$ maintains full column rank.

Removing a Constraint

We now have to decide whether $\boldsymbol{x}_k$ is a global minimizer of the inequality constrained QP (4.3).

From the optimality conditions

$$\boldsymbol{G}\boldsymbol{x}^* + \boldsymbol{g} - \sum_{i\in\mathcal{I}} \boldsymbol{a}_i \mu_i^* = \boldsymbol{0} \tag{4.34a}$$

$$\boldsymbol{a}_i^T \boldsymbol{x}^* \geq b_i \qquad i \in \mathcal{I} \tag{4.34b}$$

$$\mu_i^* \geq 0 \qquad i \in \mathcal{I} \tag{4.34c}$$

$$\mu_i^* (\boldsymbol{a}_i^T \boldsymbol{x}^* - b_i) = 0 \qquad i \in \mathcal{I} \tag{4.34d}$$

it is seen that $\boldsymbol{x}_k$ is the global minimizer $\boldsymbol{x}^*$ if and only if the pair $\boldsymbol{x}_k, \mu$ satisfies

$$\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g} - \sum_{i \in \mathcal{I}} \boldsymbol{a}_i \mu_i = \underbrace{\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g} - \sum_{i \in \mathcal{W}_k} \boldsymbol{a}_i \mu_i}_{=0} - \sum_{i \in \mathcal{I} \backslash \mathcal{W}_k} \boldsymbol{a}_i \underbrace{\mu_i}_{=0} = \boldsymbol{0} \tag{4.35a}$$

$$\boldsymbol{a}_i^T \boldsymbol{x}_k \geq b_i \qquad i \in \mathcal{I} \tag{4.35b}$$

$$\mu_i \geq 0 \qquad i \in \mathcal{I}. \tag{4.35c}$$

We must remark, that

$$\mu_i \underbrace{(\boldsymbol{a}_i^T \boldsymbol{x}_k - b_i)}_{=0} = 0, \quad i \in \mathcal{W}_k \tag{4.36a}$$

$$\underbrace{\mu_i}_{=0} (\boldsymbol{a}_i^T \boldsymbol{x}_k - b_i) = 0, \quad i \in \mathcal{I} \backslash \mathcal{W}_k \tag{4.36b}$$

from which we have

$$\mu_i (\boldsymbol{a}_i^T \boldsymbol{x}_k - b_i) = 0, \quad i \in \mathcal{I}. \tag{4.37}$$

So we see, that $\boldsymbol{x}_k$ is the unique global minimizer of (4.3), if the computed Lagrange multipliers μ_i for $i \in \mathcal{W}_k$ are non-negative. The remaining Lagrangian multipliers for $i \in \mathcal{I} \backslash \mathcal{W}_k$ are then selected according to the optimality conditions (4.7), so

$$\boldsymbol{x}^* = \boldsymbol{x}_k \tag{4.38}$$

$$\mu_i^* = \begin{cases} \mu_i, & i \in \mathcal{W}_k \\ 0, & i \in \mathcal{I} \backslash \mathcal{W}_k. \end{cases} \tag{4.39}$$

But if there exists an index $j \in \mathcal{W}_k$ such that $\mu_j < 0$, then the point $\boldsymbol{x}_k$ cannot be the global minimizer of (4.3). So we have to relax, in other words leave, the constraint $\boldsymbol{a}_j^T \boldsymbol{x}_k = b_j$ and move in a direction $\boldsymbol{p}$ such that $\boldsymbol{a}_j^T (\boldsymbol{x}_k + \alpha \boldsymbol{p}) > b_j$. From sensitivity theory in Nocedal and Wright [14] we know, that a decrease in function value f is obtained by choosing any constraint for which the Lagrange multiplier is negative. The largest rate of decrease is obtained by selecting $j \in \mathcal{W}_k$ corresponding to the most negative Lagrange multiplier.

So if an index $j \in \mathcal{W}_k$ exists, where $\mu_j < 0$, we will find an improving direction $\boldsymbol{p}^*$, which is a solution to

$$\min_{\boldsymbol{p} \in \mathbb{R}^n} \quad \phi(\boldsymbol{p}) = \frac{1}{2}\boldsymbol{p}^T \boldsymbol{G}\boldsymbol{p} + (\boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{g})^T \boldsymbol{p} \tag{4.40a}$$

$$\text{s.t.} \quad \boldsymbol{a}_i^T \boldsymbol{p} = 0 \qquad i \in \mathcal{W}_k \backslash \{j\}. \tag{4.40b}$$

As $\boldsymbol{p}^*$ is the global minimizer of (4.40), there exists multipliers $\boldsymbol{\mu}^*$ so that

$$\boldsymbol{G}\boldsymbol{p}^* + \boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{g} - \sum_{i \in \mathcal{W}_k \backslash \{j\}} \boldsymbol{a}_i \mu_i^* = \boldsymbol{0}, \tag{4.41}$$

and if we let $\boldsymbol{x}_k$ and $\hat{\boldsymbol{\mu}}$ satisfy

$$\boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{g} - \sum_{i \in \mathcal{W}_k} \boldsymbol{a}_i \hat{\mu}_i = \boldsymbol{0} \tag{4.42a}$$

$$\boldsymbol{a}_i^T \boldsymbol{x}_k = b_i \qquad i \in \mathcal{W}_k, \tag{4.42b}$$

and we subtract (4.42a) from (4.41) we get

$$\boldsymbol{G}\boldsymbol{p}^* - \sum_{i \in \mathcal{W}_k \backslash \{j\}} \boldsymbol{a}_i (\mu_i^* - \hat{\mu}_i) + \boldsymbol{a}_j \hat{\mu}_j = 0, \tag{4.43}$$

which is equivalent to

$$\boldsymbol{a}_j = \sum_{i \in \mathcal{W}_k \backslash \{j\}} \frac{\mu_i^* - \hat{\mu}_i}{\hat{\mu}_j} \boldsymbol{a}_i - \frac{\boldsymbol{G}\boldsymbol{p}^*}{\hat{\mu}_j}. \tag{4.44}$$

Since $\boldsymbol{a}_i$ is linearly independent for $i \in \mathcal{W}_k$ and thereby also for $i \in \mathcal{W}_k \backslash \{j\}$, then $\boldsymbol{a}_j$ cannot be a linear combination of $\boldsymbol{a}_i$, which means

$$\boldsymbol{a}_j \neq \sum_{i \in \mathcal{W}_k \setminus \{j\}} \frac{\mu_i^* - \hat{\mu}_i}{\hat{\mu}_j} \boldsymbol{a}_i \tag{4.45}$$

implying that $\boldsymbol{p}^* \neq \boldsymbol{0}$. Now we will shortly summarize, what have been stated in this section so far. If the optimal solution has not been found at iteration k some negative Lagrange multipliers exist. The constraint $j \in \mathcal{W}_k$, which correspond to the most negative Lagrange multiplier μ_j is removed from the working set. The new improving direction is computed by solving (4.40) and it is guaranteed to be non-zero $\boldsymbol{p}^* \neq \boldsymbol{0}$. This statement is important in the following derivations.

By taking a new step after removing constraint j, we must now guarantee, that the relaxed constraint is not violated again, in other words that the remaining constraints in the current working set are still satisfied and that we actually get a decrease in function value f.

Taking the dot-product of $\boldsymbol{a}_j$ and $\boldsymbol{p}^*$, by means of multiplying (4.44) with $\boldsymbol{p}^{*T}$, we get

$$\boldsymbol{p}^{*T} \boldsymbol{a}_j = \sum_{i \in \mathcal{W}_k \setminus \{j\}} \frac{\mu_i^* - \hat{\mu}_i}{\hat{\mu}_j} \boldsymbol{p}^{*T} \boldsymbol{a}_i - \frac{\boldsymbol{p}^{*T} \boldsymbol{G} \boldsymbol{p}^*}{\hat{\mu}_j} \tag{4.46}$$

which by transposing becomes

$$\boldsymbol{a}_j^T \boldsymbol{p}^* = \sum_{i \in \mathcal{W}_k \setminus \{j\}} \frac{\mu_i^* - \hat{\mu}_i}{\hat{\mu}_j} \underbrace{\boldsymbol{a}_i^T \boldsymbol{p}^*}_{=0} - \frac{\boldsymbol{p}^{*T} \boldsymbol{G} \boldsymbol{p}^*}{\hat{\mu}_j} = -\frac{\boldsymbol{p}^{*T} \boldsymbol{G} \boldsymbol{p}^*}{\hat{\mu}_j} \quad . \tag{4.47}$$

Since $\boldsymbol{p}^{*T} \boldsymbol{G} \boldsymbol{p}^* > 0$, $\boldsymbol{p}^* \neq 0$, and $\hat{\mu}_j < 0$, it follows that

$$\boldsymbol{a}_j^T \boldsymbol{p}^* > 0. \tag{4.48}$$

Bearing in mind that $\boldsymbol{x}_k \in \mathcal{M}_k = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{a}_i^T \boldsymbol{x}_k = b_i, i \in \mathcal{W}_k\}$, we see that

$$\boldsymbol{a}_j^T (\boldsymbol{x}_k + \alpha \boldsymbol{p}^*) = \underbrace{\boldsymbol{a}_j^T \boldsymbol{x}_k}_{=b_j} + \alpha \underbrace{\boldsymbol{a}_j^T \boldsymbol{p}^*}_{>0} > b_j, \quad \forall \alpha \in]0, 1] \tag{4.49}$$

and

$$\boldsymbol{a}_i^T(\boldsymbol{x}_k + \alpha \boldsymbol{p}^*) = \underbrace{\boldsymbol{a}_i^T \boldsymbol{x}_k}_{=b_i} + \alpha \underbrace{\boldsymbol{a}_i^T \boldsymbol{p}^*}_{=0} = b_i, \quad \forall \alpha \in]0,1], \quad i \in \mathcal{W}_k \backslash \{j\}. \tag{4.50}$$

As expected, we see that the relaxed constraint j and the constraints in the new active set are still satisfied.

As (4.40) is strictly convex, we know, that $\phi(\boldsymbol{p}^*) < 0$ for the improving direction $\boldsymbol{p}^*$, and the feasible non-optimal solution $\boldsymbol{p} = \boldsymbol{0}$ has the value $\phi(\boldsymbol{0}) = 0$. If we also keep in mind that

$$f(\boldsymbol{x}_k + \alpha \boldsymbol{p}^*) = f(\boldsymbol{x}_k) + \phi(\alpha \boldsymbol{p}^*) \tag{4.51}$$

and when $\alpha = 1$, then we get

$$f(\boldsymbol{x}_k + \boldsymbol{p}^*) = f(\boldsymbol{x}_k) + \phi(\boldsymbol{p}^*) < f(\boldsymbol{x}_k). \tag{4.52}$$

From this relation, the convexity of f and because $\alpha \in]0;1]$, we know that

$$\begin{aligned} f(\boldsymbol{x}_k + \alpha \boldsymbol{p}^*) &= f(\boldsymbol{x}_k + \alpha \boldsymbol{x}_k - \alpha \boldsymbol{x}_k + \alpha \boldsymbol{p}^*) \\ &= f((1-\alpha)\boldsymbol{x}_k + \alpha(\boldsymbol{x}_k + \boldsymbol{p}^*)) \\ &\leq (1-\alpha) f(\boldsymbol{x}_k) + \alpha f(\boldsymbol{x}_k + \boldsymbol{p}^*) \\ &< (1-\alpha) f(\boldsymbol{x}_k) + \alpha f(\boldsymbol{x}_k) \\ &= f(\boldsymbol{x}_k) + \alpha f(\boldsymbol{x}_k) - \alpha f(\boldsymbol{x}_k) \\ &= f(\boldsymbol{x}_k). \end{aligned} \tag{4.53}$$

So in fact we actually get a decrease in function value f, by taking the new step having relaxed constraint j.

In this section we have found that if $\mu_j < 0$, then the current point $\boldsymbol{x}_k$ cannot be a global minimizer. So to proceed we have to remove constraint j from the current working set, $\mathcal{W}_{k+1} = \mathcal{W}_k \backslash \{j\}$, and update the current point by taking a zero step, $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k$.

A constraint removed from the working set cannot be appended to the working set in the iteration immediately after taking the zero step. This is because a blocking constraint is characterized by $\boldsymbol{a}_j^T\boldsymbol{p}^* < 0$, while from (4.48) we know, that $\boldsymbol{a}_j^T\boldsymbol{p}^* > 0$. Still, the possibility of cycling can be a problem for the primal active set method, for example in cases like

$$\ldots \longrightarrow \{i,j,l\} \xrightarrow{-j} \{i,l\} \xrightarrow{-l} \{i\} \xrightarrow{+j} \{i,j\} \xrightarrow{+l} \{i,j,l\} \xrightarrow{-j} \{i,l\} \longrightarrow \ldots \tag{4.54}$$

This and similar cases are not considered, so if any cycling occurs, it is stopped by setting a maximum number of iterations for the method.

The procedure of the primal active set method is stated in algorithm 4.1.1.

Algorithm 4.1.1: Primal Active Set Algorithm for Convex Inequality Constrained QP's.

Input: Feasible point $\boldsymbol{x}_0$, $\mathcal{W} = \mathcal{A}_0 = \{i : \boldsymbol{a}_i^T \boldsymbol{x}_0 = b_i\}$.
while NOT STOP do `/* find improving direction` $\boldsymbol{p}^*$ `*/`
  Find the improving direction $\boldsymbol{p}^*$ by solving the equality constrained QP:

$$\min_{\boldsymbol{p}\in\mathbb{R}^n} \phi(\boldsymbol{p}) = \frac{1}{2}\boldsymbol{p}^T\boldsymbol{G}\boldsymbol{p} + (\boldsymbol{G}\boldsymbol{x}+\boldsymbol{g})^T\boldsymbol{p}$$
$$\text{s.t.} \quad \boldsymbol{a}_i^T\boldsymbol{p} = 0, \qquad i \in \mathcal{W}$$

  if $||\boldsymbol{p}^*|| = 0$ **then** `/* compute Lagrange multipliers` μ_i `*/`
    Compute the Lagrange multipliers $\mu_i, i \in \mathcal{W}$ by solving:

$$\sum_{i\in\mathcal{W}} \boldsymbol{a}_i\mu_i = \boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}$$

    $\mu_i \leftarrow 0, i \in \mathcal{I}\backslash\mathcal{W}$
    if $\mu_i \geq 0 \; \forall i \in \mathcal{W}$ **then**
      STOP, the optimal solution $\boldsymbol{x}^*$ has been found!
    else `/* remove constraint` j `*/`
      $\boldsymbol{x} \leftarrow \boldsymbol{x}$
      $\mathcal{W} \leftarrow \mathcal{W}\backslash\{j\}, j \in \mathcal{W} : \mu_j < 0$
  else `/* compute step length` α `*/`

$$\alpha = \min\left(1, \min_{i\in\mathcal{I}\backslash\mathcal{W}:\boldsymbol{a}_i^T\boldsymbol{p}^*<0} \frac{b_i - \boldsymbol{a}_i^T\boldsymbol{x}}{\boldsymbol{a}_i^T\boldsymbol{p}^*}\right)$$
$$\mathcal{J} = \arg \min_{i\in\mathcal{I}\backslash\mathcal{W}:\boldsymbol{a}_i^T\boldsymbol{p}^*<0} \frac{b_i - \boldsymbol{a}_i^T\boldsymbol{x}}{\boldsymbol{a}_i^T\boldsymbol{p}^*}$$

    if $\alpha < 1$ **then** `/* append constraint` j `*/`
      $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha\boldsymbol{p}^*$
      $\mathcal{W} \leftarrow \mathcal{W} \cup \{j\}, j \in \mathcal{J}$
    else
      $\boldsymbol{x} \leftarrow \boldsymbol{x} + \boldsymbol{p}^*$
      $\mathcal{W} \leftarrow \mathcal{W}$

4.2 Primal active set method by example

We will now demonstrate how the primal active set method finds the optimum in the following example:

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x}, \quad \boldsymbol{G} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \boldsymbol{g} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\begin{aligned} \text{s.t.} \qquad & c_1 = -x_1 + x_2 - 1 \geq 0 \\ & c_2 = -\frac{1}{2}x_1 - x_2 + 2 \geq 0 \\ & c_3 = -x_2 + 2.5 \geq 0 \\ & c_4 = -3x_1 + x_2 + 3 \geq 0. \end{aligned}$$

At every iteration k the path $(\boldsymbol{x}^1 \ldots \boldsymbol{x}^k)$ is plotted together with the constraints, where active constraints are indicated in red. The 4 constraints and their column index in $\boldsymbol{A}$ are labeled on the constraint in the plot. The feasible area is in the top right corner where the plot is lightest. The start position is chosen to be $\boldsymbol{x} = [4.0, 4.0]^T$ which is feasible and the active set is empty $\mathcal{W} = \emptyset$. For every iteration we have plotted the situation when we enter the while-loop at $\boldsymbol{x}$, see algorithm 4.1.1 .

Iteration 1

The situation is illustrated in figure 4.1. On entering the while-loop the working set is empty and therefore the improving direction is found to be $\boldsymbol{p} = [-4.0, -4.0]^T$. As figure 4.1 suggests, the first constraint to be violated taking this step is c_3 which is at step length $\alpha = 0.375$. The step $\bar{\boldsymbol{x}} = \boldsymbol{x} + \alpha\boldsymbol{p}$ is taken and the constraint c_3 is appended to the working set.

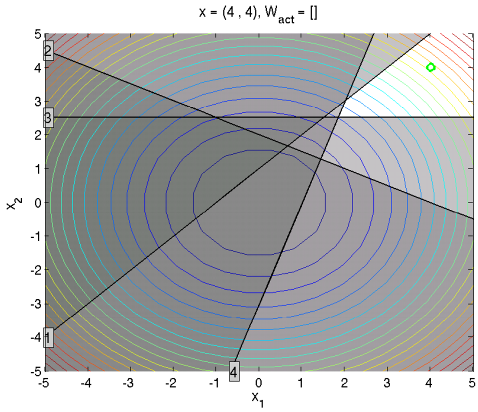


Figure 4.1: Iteration 1, $\mathcal{W} = \emptyset$, $\boldsymbol{x} = [4.0, 4.0]^T$.

Iteration 2
The situation is illustrated in figure 4.2. Now the working set is $\mathcal{W} = [3]$ which means that the new improving direction $\boldsymbol{p}$ is found by minimizing $f(\boldsymbol{x})$ subject to $c_3(\boldsymbol{x} + \boldsymbol{p}) = 0$. The improving direction is found to be $\boldsymbol{p} = [-2.5, 0.0]^T$ and $\boldsymbol{\mu} = [2.5]$. The first constraint to be violated in this direction is c_4 which is at step length $\alpha = 0.267$. The step $\bar{\boldsymbol{x}} = \boldsymbol{x} + \alpha\boldsymbol{p}$ is taken and the constraint c_4 is appended to the working set.

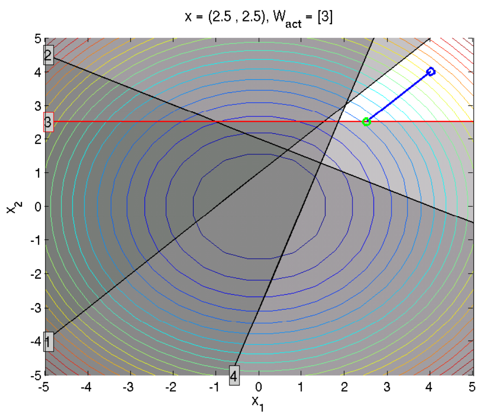


Figure 4.2: Iteration 2, $\mathcal{W} = [3]$, $\boldsymbol{x} = [2.5, 2.5]^T$.

Iteration 3

The situation is illustrated in figure 4.3. Here the working set is $\mathcal{W} = [3, 4]$ and the new improving direction is found to be $\boldsymbol{p} = [0, 0]^T$ and $\boldsymbol{\mu} = [3.1, 0.6]^T$. Because $\boldsymbol{p} = \boldsymbol{0}$ and no negative Lagrange Multipliers exist, position $\boldsymbol{x}$ is optimal. Therefore the method terminates with $\boldsymbol{x}^* = [1.8, 2.5]^T$.

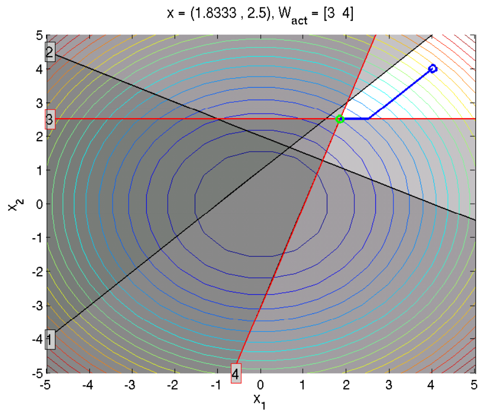


Figure 4.3: Iteration 3, $\mathcal{W} = [3, 4]$, $\boldsymbol{x}^* = [1.8, 2.5]^T$.

An interactive demo application `QP_demo.m` is found in appendix D.5.

4.3 Dual active set method

In the foregoing, we have described the primal active set method which solves an inequality constrained convex QP, by keeping track of a working set $\mathcal{W}$. In this section we will examine the dual active set method, which requires the QP to be strictly convex. The dual active set method uses the dual set of $\mathcal{W}$, which we will call $\mathcal{W}_{\mathcal{D}}$. The method benefits from always having an easily calculated feasible starting point and the method does not have the possibility of cycling. The theory is based on earlier works by Goldfarb and Idnani [10], Schmid and Biegler [11] and Schittkowski [12].

4.3.1 Survey

The inequality constrained strictly convex QP that we want to solve is as follows

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x} \tag{4.55a}$$

$$\text{s.t.} \quad c_i(\boldsymbol{x}) = \boldsymbol{a}_i^T\boldsymbol{x} - b_i \geq 0, \quad i \in \mathcal{I}. \tag{4.55b}$$

The corresponding Lagrangian function is

$$L(\boldsymbol{x},\boldsymbol{\mu}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x} - \sum_{i\in\mathcal{I}} \mu_i(\boldsymbol{a}_i^T\boldsymbol{x} - b_i). \tag{4.56}$$

The dual program of (4.55) is

$$\max_{\boldsymbol{x}\in\mathbb{R}^n,\boldsymbol{\mu}\in\mathbb{R}^m} L(\boldsymbol{x},\boldsymbol{\mu}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x} - \sum_{i\in\mathcal{I}} \mu_i(\boldsymbol{a}_i^T\boldsymbol{x} - b_i) \tag{4.57a}$$

$$\text{s.t.} \quad \boldsymbol{G}\boldsymbol{x} + \boldsymbol{g} - \sum_{i\in\mathcal{I}} \boldsymbol{a}_i\mu_i = \boldsymbol{0} \tag{4.57b}$$

$$\mu_i \geq 0 \qquad i \in \mathcal{I}. \tag{4.57c}$$

The necessary and sufficient conditions for optimality of the dual program is

$$\boldsymbol{Gx}^* + \boldsymbol{g} - \sum_{i \in \mathcal{I}} \boldsymbol{a}_i \mu_i^* = \boldsymbol{0} \tag{4.58a}$$

$$c_i(\boldsymbol{x}^*) = \boldsymbol{a}_i^T \boldsymbol{x}^* - b_i = 0 \qquad i \in A(x^*) \tag{4.58b}$$

$$c_i(\boldsymbol{x}^*) = \boldsymbol{a}_i^T \boldsymbol{x}^* - b_i > 0 \qquad i \in \mathcal{I} \backslash A(x^*) \tag{4.58c}$$

$$\mu_i \geq 0 \qquad i \in A(x^*) \tag{4.58d}$$

$$\mu_i = 0 \qquad i \in \mathcal{I} \backslash A(x^*). \tag{4.58e}$$

These conditions are exactly the same as the optimality conditions of the primal program (4.55), and this corresponds to the fact that the optimal value $L(\boldsymbol{x}^*, \boldsymbol{\mu}^*)$ of the dual program is equivalent to the optimal value $f(\boldsymbol{x}^*)$ of the primal program. This is why the solution of the primal program can be found by solving the dual program (4.57).

The method maintains dual feasibility at any iteration $\{\boldsymbol{x}^k, \boldsymbol{\mu}^k\}$ by satisfying (4.57b) and (4.57c). This is done by keeping track of a working set $\mathcal{W}$. The constraints in the working set satisfy

$$\boldsymbol{Gx}^k + \boldsymbol{g} - \sum_{i \in \mathcal{W}} \boldsymbol{a}_i \mu_i^k = \boldsymbol{0} \tag{4.59a}$$

$$c_i(\boldsymbol{x}^k) = \boldsymbol{a}_i^T \boldsymbol{x}^k - b_i = 0 \qquad i \in \mathcal{W} \tag{4.59b}$$

$$\mu_i^k \geq 0 \qquad i \in \mathcal{W}. \tag{4.59c}$$

The constraints in the complementary set $\mathcal{W_D} = \mathcal{I} \backslash \mathcal{W}$, i.e. the active set of the dual program, satisfy

$$\boldsymbol{Gx}^k + \boldsymbol{g} - \sum_{i \in \mathcal{W_D}} \boldsymbol{a}_i \mu_i^k = \boldsymbol{0} \tag{4.60a}$$

$$\mu_i^k = 0 \qquad i \in \mathcal{W_D}, \tag{4.60b}$$

and from (4.58), (4.59) and (4.60) it is clear that an optimum has been found $\boldsymbol{x}^k = \boldsymbol{x}^*$ if

$$c_i(\boldsymbol{x}^k) = \boldsymbol{a}_i^T \boldsymbol{x}^k - b_i \geq 0 \quad i \in \mathcal{W_D}. \tag{4.61}$$

If this is not the case some violated constraint $r \in \mathcal{W}_\mathcal{D}$ exists, i.e. $c_r(\boldsymbol{x}^k) < 0$. The following relationship explains why $\{\boldsymbol{x}^k, \boldsymbol{\mu}^k\}$ cannot be an optimum in this case

$$\frac{\partial L}{\partial \mu_r}(\boldsymbol{x}^k, \boldsymbol{\mu}^k) = -c_r(\boldsymbol{x}^k) > 0. \tag{4.62}$$

This means that (4.57a) can be increased by increasing the Lagrangian multiplier μ_r. In fact, this explains the key idea of the dual active set method. The idea is to choose a constraint c_r from the dual active working set $\mathcal{W}_\mathcal{D}$ which is violated $c_r(\boldsymbol{x}^k) < 0$ and make it satisfied $c_r(\boldsymbol{x}^k) \geq 0$ by increasing the Lagrangian multiplier μ_r. This procedure continues iteratively until no constraints from $\mathcal{W}_\mathcal{D}$ are violated. At this point the optimum has been found and the method terminates.

4.3.2 Improving Direction and Step Length

If optimality has not been found at iteration k it indicates that a constraint c_r is violated, which means that $c_r(\boldsymbol{x}^k) < 0$. In this section we will investigate how to find both an improving direction and a step length which satisfy the violated constraint c_r.

Improving Direction

The Lagrangian multiplier μ_r of the violated constraint c_r from $\mathcal{W}_\mathcal{D}$ should be changed from zero to some value that will optimize (4.57a) and satisfy (4.57b) and (4.57c). After this operation the new position is

$$\bar{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{s} \tag{4.63a}$$
$$\bar{\mu}_i = \mu_i + u_i \quad i \in \mathcal{W} \tag{4.63b}$$
$$\bar{\mu}_r = \mu_r + t \tag{4.63c}$$
$$\bar{\mu}_i = \mu_i = 0 \quad i \in \mathcal{W}_\mathcal{D} \backslash r. \tag{4.63d}$$

From (4.57b) and (4.59b) we know that $\bar{\boldsymbol{x}}$ and $\bar{\boldsymbol{\mu}}$ should satisfy

$$\boldsymbol{G}\bar{\boldsymbol{x}} + \boldsymbol{g} - \sum_{i \in \mathcal{I}} \boldsymbol{a}_i \bar{\mu}_i = \boldsymbol{0} \tag{4.64a}$$

$$c_i(\bar{\boldsymbol{x}}) = \boldsymbol{a}_i^T \bar{\boldsymbol{x}} - b_i = 0 \qquad i \in \mathcal{W}. \tag{4.64b}$$

As $\mu_i \neq 0$ for $i \in \mathcal{W}$, $\bar{\mu}_r \neq 0$ and r yet not in $\mathcal{W}$, this can be written as

$$\begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \bar{\boldsymbol{x}} \\ \bar{\boldsymbol{\mu}} \end{pmatrix} + \begin{pmatrix} \boldsymbol{g} \\ \boldsymbol{b} \end{pmatrix} - \begin{pmatrix} \boldsymbol{a}_r \\ \boldsymbol{0} \end{pmatrix} \bar{\mu}_r = \boldsymbol{0}, \tag{4.65}$$

where $\boldsymbol{A} = [\boldsymbol{a}_i]_{i \in \mathcal{W}}$ has full column rank, $\boldsymbol{G}$ is symmetric and positive definite, $\bar{\boldsymbol{\mu}} = [\bar{\mu}_i]_{i \in \mathcal{W}}^T$, $\boldsymbol{a}_r$ is the constraint from $\mathcal{W}_\mathcal{D}$ we are looking at and μ_r is the corresponding Lagrangian multiplier. Using (4.63) this can be formulated as

$$\begin{aligned} &\begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\mu} \end{pmatrix} + \begin{pmatrix} \boldsymbol{g} \\ \boldsymbol{b} \end{pmatrix} - \begin{pmatrix} \boldsymbol{a}_r \\ \boldsymbol{0} \end{pmatrix} \mu_r + \\ &\qquad \begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{s} \\ \boldsymbol{u} \end{pmatrix} - \begin{pmatrix} \boldsymbol{a}_r \\ \boldsymbol{0} \end{pmatrix} t = \boldsymbol{0}. \end{aligned} \tag{4.66}$$

From (4.64) we have

$$\begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\mu} \end{pmatrix} + \begin{pmatrix} \boldsymbol{g} \\ \boldsymbol{b} \end{pmatrix} - \begin{pmatrix} \boldsymbol{a}_r \\ \boldsymbol{0} \end{pmatrix} \mu_r = \boldsymbol{0} \tag{4.67}$$

and therefore (4.66) is simplified as follows

$$\begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{s} \\ \boldsymbol{u} \end{pmatrix} - \begin{pmatrix} \boldsymbol{a}_r \\ \boldsymbol{0} \end{pmatrix} t = \boldsymbol{0}, \tag{4.68}$$

which is equivalent to

$$\begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{v} \end{pmatrix} = \begin{pmatrix} \boldsymbol{a}_r \\ \boldsymbol{0} \end{pmatrix}, \qquad \begin{pmatrix} \boldsymbol{s} \\ \boldsymbol{u} \end{pmatrix} = \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{v} \end{pmatrix} t. \tag{4.69}$$

The new improving direction $(\boldsymbol{p},\ \boldsymbol{v})^T$ is found by solving (4.69), using a solver for equality constrained QP's, e.g. the range space or the null space procedure.

Step Length

Having the improving direction we now would like to find the step length t (4.69). This step length should be chosen in a way, that makes (4.57c) satisfied. From (4.63), (4.66) and (4.69) we have the following statements about the new step

$$\bar{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{s} = \boldsymbol{x} + t\boldsymbol{p} \tag{4.70a}$$

$$\bar{\mu}_i = \mu_i + u_i = \mu_i + tv_i \quad i \in \mathcal{W} \tag{4.70b}$$

$$\bar{\mu}_r = \mu_r + t \tag{4.70c}$$

$$\bar{\mu}_i = \mu_i = 0 \quad\quad i \in \mathcal{W}_{\mathcal{D}} \backslash r. \tag{4.70d}$$

To make sure that $\bar{\mu}_r \geq 0$ (4.57c), we must require, that $t \geq 0$. When $v_i \geq 0$ we have $\bar{\mu}_r \geq 0$ and (4.57c) is satisfied for any value of $t \geq 0$. When $v_i < 0$ we must require t to be some positive value less than $\frac{-\mu_i}{v_i}$, which makes $\bar{\mu}_i \geq 0$ as $\mu_i \geq 0$ and $v_i < 0$. This means that t should be chosen as

$$t \in [0, t_{\max}], \quad t_{\max} = \min(\infty, \min_{i: v_i < 0} \frac{-\mu_i}{v_i}) \geq 0. \tag{4.71}$$

Now we know what values of t we can choose, in order to retain dual feasibility when taking the new step. To find out what exact value of t in the interval (4.71) we should choose to make the step optimal we need to examine what happens to the primal objective function (4.55a), the dual objective function (4.57a), and the constraint c_r as we take the step.

The relation between $c_r(\boldsymbol{x})$ and $c_r(\bar{\boldsymbol{x}})$

Now we shall examine how $c_r(\bar{\boldsymbol{x}})$ is related to $c_r(\boldsymbol{x})$. For this reason, we need to state the following properties. From (4.69) we have

$$\boldsymbol{a}_r = \boldsymbol{G}\boldsymbol{p} - \boldsymbol{A}\boldsymbol{v} \tag{4.72a}$$

$$\boldsymbol{A}^T\boldsymbol{p} = \boldsymbol{0}. \tag{4.72b}$$

Multiplying (4.72a) with $\boldsymbol{p}$ gives

$$\boldsymbol{a}_r^T\boldsymbol{p} = (\boldsymbol{G}\boldsymbol{p} - \boldsymbol{A}\boldsymbol{v})^T\boldsymbol{p} = \boldsymbol{p}^T\boldsymbol{G}\boldsymbol{p} - \boldsymbol{v}^T\boldsymbol{A}^T\boldsymbol{p} = \boldsymbol{p}^T\boldsymbol{G}\boldsymbol{p} \tag{4.73}$$

and because $\boldsymbol{G}$ is positive definite, we have

$$\boldsymbol{a}_r^T\boldsymbol{p} = \boldsymbol{p}^T\boldsymbol{G}\boldsymbol{p} \geq 0 \tag{4.74a}$$
$$\boldsymbol{a}_r^T\boldsymbol{p} = \boldsymbol{p}^T\boldsymbol{G}\boldsymbol{p} = 0 \quad \Leftrightarrow \quad \boldsymbol{p} = \boldsymbol{0} \tag{4.74b}$$
$$\boldsymbol{a}_r^T\boldsymbol{p} = \boldsymbol{p}^T\boldsymbol{G}\boldsymbol{p} > 0 \quad \Leftrightarrow \quad \boldsymbol{p} \neq \boldsymbol{0}. \tag{4.74c}$$

As $\bar{\boldsymbol{x}} = \boldsymbol{x} + t\boldsymbol{p}$ we get

$$c_r(\bar{\boldsymbol{x}}) = c_r(\boldsymbol{x} + t\boldsymbol{p}) = \boldsymbol{a}_r^T(\boldsymbol{x} + t\boldsymbol{p}) - b_r = \boldsymbol{a}_r^T\boldsymbol{x} - b_r + t\boldsymbol{a}_r^T\boldsymbol{p} \tag{4.75}$$

and because $c_r(\boldsymbol{x}) = \boldsymbol{a}_r^T\boldsymbol{x} - b_r$ this is equivalent to

$$c_r(\bar{\boldsymbol{x}}) = c_r(\boldsymbol{x}) + t\boldsymbol{a}_r^T\boldsymbol{p}. \tag{4.76}$$

From (4.71) and (4.74a) we know that $t\boldsymbol{a}_r^T\boldsymbol{p} \geq 0$ and therefore

$$c_r(\bar{\boldsymbol{x}}) \geq c_r(\boldsymbol{x}). \tag{4.77}$$

This means that the constraint c_r is increasing (if $t > 0$) as we move from $\boldsymbol{x}$ to $\bar{\boldsymbol{x}}$ and this is exactly what we want as it is negative and violated at $\boldsymbol{x}$.

The relation between $f(\boldsymbol{x})$ and $f(\bar{\boldsymbol{x}})$

In addition to the foregoing we will now investigate what happens to the primal objective function (4.55a) as we move from $\boldsymbol{x}$ to $\bar{\boldsymbol{x}}$. Inserting $\bar{\boldsymbol{x}} = \boldsymbol{x} + t\boldsymbol{p}$ in (4.55a) gives

$$f(\bar{\boldsymbol{x}}) = f(\boldsymbol{x} + t\boldsymbol{p}) = \frac{1}{2}(\boldsymbol{x} + t\boldsymbol{p})^T\boldsymbol{G}(\boldsymbol{x} + t\boldsymbol{p}) + \boldsymbol{g}^T(\boldsymbol{x} + t\boldsymbol{p}), \tag{4.78}$$

which may be reformulated as

$$f(\bar{\boldsymbol{x}}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x} + \frac{1}{2}t^2\boldsymbol{p}^T\boldsymbol{G}\boldsymbol{p} + t(\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g})^T\boldsymbol{p}, \tag{4.79}$$

and using (4.55a) this leads to

$$f(\bar{\boldsymbol{x}}) = f(\boldsymbol{x}) + \frac{1}{2}t^2\boldsymbol{p}^T\boldsymbol{G}\boldsymbol{p} + t(\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g})^T\boldsymbol{p}. \tag{4.80}$$

From (4.67) we have the relation

$$\boldsymbol{G}\boldsymbol{x} - \boldsymbol{A}\boldsymbol{\mu} + \boldsymbol{g} - \boldsymbol{a}_r\mu_r = \boldsymbol{0}, \tag{4.81}$$

which is equivalent to

$$\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g} = \boldsymbol{A}\boldsymbol{\mu} + \boldsymbol{a}_r\mu_r \tag{4.82}$$

and when multiplied with $\boldsymbol{p}$ this gives

$$(\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g})^T\boldsymbol{p} = (\boldsymbol{A}\boldsymbol{\mu} + \boldsymbol{a}_r\mu_r)^T\boldsymbol{p} = \boldsymbol{\mu}^T\boldsymbol{A}^T\boldsymbol{p} + \mu_r\boldsymbol{a}_r^T\boldsymbol{p}. \tag{4.83}$$

Furthermore using the fact that $\boldsymbol{A}^T\boldsymbol{p} = \boldsymbol{0}$, this is equivalent to

$$(\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g})^T\boldsymbol{p} = \mu_r\boldsymbol{a}_r^T\boldsymbol{p}. \tag{4.84}$$

Inserting this in (4.80) gives

$$f(\bar{\boldsymbol{x}}) = f(\boldsymbol{x}) + \frac{1}{2}t^2\boldsymbol{p}^T\boldsymbol{G}\boldsymbol{p} + t\mu_r\boldsymbol{a}_r^T\boldsymbol{p}. \tag{4.85}$$

Using $\boldsymbol{p}^T\boldsymbol{G}\boldsymbol{p} = \boldsymbol{a}_r^T\boldsymbol{p}$ from (4.74a) we get

$$f(\bar{\boldsymbol{x}}) = f(\boldsymbol{x}) + \frac{1}{2}t^2\boldsymbol{a}_r^T\boldsymbol{p} + t\mu_r\boldsymbol{a}_r^T\boldsymbol{p} = f(\boldsymbol{x}) + t(\mu_r + \frac{1}{2}t)\boldsymbol{a}_r^T\boldsymbol{p}. \tag{4.86}$$

As $t \geq 0$, $\boldsymbol{a}_r^T\boldsymbol{p} \geq 0$ and $\mu_r \geq 0$ the primal objective function does not decrease when we move from $\boldsymbol{x}$ to $\bar{\boldsymbol{x}}$.

The relation between $L(\boldsymbol{x}, \boldsymbol{\mu})$ and $L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}})$

We will now investigate what happens to the Lagrangian function (4.57a) as we move from $(\boldsymbol{x}, \boldsymbol{\mu})$ to $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}})$. After taking a new step we have

$$L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) = f(\bar{\boldsymbol{x}}) - \sum_{i \in \mathcal{I}} \mu_i c_i(\bar{\boldsymbol{x}}), \tag{4.87}$$

and because $\mu_i = 0$ for $i \in \mathcal{W}_{\mathcal{D}}$ and $c_i(\bar{\boldsymbol{x}}) = 0$ for $i \in \mathcal{W}$ this is equivalent to

$$L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) = f(\bar{\boldsymbol{x}}) - \bar{\mu}_r c_r(\bar{\boldsymbol{x}}). \tag{4.88}$$

By replacing $f(\bar{\boldsymbol{x}})$ with (4.86), $\bar{\mu}_r$ with $\mu_r + t$ and $c_r(\bar{\boldsymbol{x}})$ with (4.76), we then have

$$L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) = f(\boldsymbol{x}) + t(\mu_r + \frac{1}{2}t)\boldsymbol{a}_r^T \boldsymbol{p} - (\mu_r + t)(c_r(\boldsymbol{x}) + t\boldsymbol{a}_r^T \boldsymbol{p}), \tag{4.89}$$

which we reformulate as

$$L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) = f(\boldsymbol{x}) + \mu_r t\boldsymbol{a}_r^T \boldsymbol{p} + \frac{1}{2}t^2 \boldsymbol{a}_r^T \boldsymbol{p} - \mu_r c_r(\boldsymbol{x}) - \mu_r t\boldsymbol{a}_r^T \boldsymbol{p} - tc_r(\boldsymbol{x}) - t^2 \boldsymbol{a}_r^T \boldsymbol{p} \tag{4.90}$$

and finally this gives

$$L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) = f(\boldsymbol{x}) - \mu_r c_r(\boldsymbol{x}) - \frac{1}{2}t^2 \boldsymbol{a}_r^T \boldsymbol{p} - tc_r(\boldsymbol{x}). \tag{4.91}$$

The Lagrangian $L(\boldsymbol{x}, \boldsymbol{\mu})$ before taking the new step is

$$L(\boldsymbol{x}, \boldsymbol{\mu}) = f(\boldsymbol{x}) - \sum_{i \in \mathcal{I}} \mu_i c_i(\boldsymbol{x}) \tag{4.92}$$

and as in the case above we have the precondition $\mu_i = 0$ for $i \in \mathcal{W}_{\mathcal{D}}$ and

$c_i(\boldsymbol{x}) = 0$ for $i \in \mathcal{W}$ and therefore (4.92) is equivalent to

$$L(\boldsymbol{x}, \boldsymbol{\mu}) = f(\boldsymbol{x}) - \mu_r c_r(\boldsymbol{x}), \tag{4.93}$$

and inserting this in (4.91) gives us

$$L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) = L(\boldsymbol{x}, \boldsymbol{\mu}) - \frac{1}{2} t^2 \boldsymbol{a}_r^T \boldsymbol{p} - t c_r(\boldsymbol{x}). \tag{4.94}$$

Now we want to know what values of t make $L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) \geq L(\boldsymbol{x}, \boldsymbol{\mu})$, i.e what values of t that satisfy

$$L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) - L(\boldsymbol{x}, \boldsymbol{\mu}) = -\frac{1}{2} t^2 \boldsymbol{a}_r^T \boldsymbol{p} - t c_r(\boldsymbol{x}) \geq 0. \tag{4.95}$$

This inequality is satisfied when

$$t \in [0, 2\frac{-c_r(\boldsymbol{x})}{\boldsymbol{a}_r^T \boldsymbol{p}}]. \tag{4.96}$$

When t is in this interval, the Lagrangian function increases as we move from $(\boldsymbol{x}, \boldsymbol{\mu})$ to $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}})$. To find the value of t that gives the greatest increment we must differentiate (4.94) with respect to t

$$\frac{dL}{dt} = -t \boldsymbol{a}_r^T \boldsymbol{p} - c_r(\boldsymbol{x}). \tag{4.97}$$

The greatest increment is at t^* where

$$-t^* \boldsymbol{a}_r^T \boldsymbol{p} - c_r(\boldsymbol{x}) = 0 \qquad \Leftrightarrow \qquad t^* = \frac{-c_r(\boldsymbol{x})}{\boldsymbol{a}_r^T \boldsymbol{p}}. \tag{4.98}$$

At this point we would like to stop up and present a short summary of what has been revealed throughout the latest sections. If optimality has not been found at iteration k, some violated constraint $c_r(\boldsymbol{x}^k) \leq 0$ must exist. The new improving direction is found by solving the equality constrained QP

$$\begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{v} \end{pmatrix} = \begin{pmatrix} \boldsymbol{a}_r \\ \boldsymbol{0} \end{pmatrix} \tag{4.99}$$

where $\boldsymbol{A} = [\boldsymbol{a}_i]_{i \in \mathcal{W}}$ has full column rank, $\boldsymbol{G}$ is symmetric and positive definite and $\boldsymbol{a}_r$ is the violated constraint. The optimal step length t, which ensures feasibility is found from statements (4.71) and (4.98)

$$t = \min\left(\min_{i:v_i<0} \frac{-\mu_i}{v_i}, \ \frac{-c_r(\boldsymbol{x})}{\boldsymbol{a}_r^T \boldsymbol{p}}\right). \tag{4.100}$$

Both the dual objective function (4.57) and the violated constraint increase as we take the step.

4.3.3 Linear Dependency

The KKT system (4.69) can only be solved if $\boldsymbol{G}$ is positive definite, and $\boldsymbol{A}$ has full column rank. If the constraints in $\boldsymbol{A} = [\boldsymbol{a}_i]_{i \in \mathcal{W}}$ and $\boldsymbol{a}_r$ are linearly dependent, it is not possible to add constraint r to the working set $\mathcal{W}$, as $\boldsymbol{A} = [\boldsymbol{a}_i]_{i \in \mathcal{W} \cup r}$ in this case would not have full column rank. This problem is solved by removing constraint j from $\mathcal{W}$, which makes the constraints in the new working set $\bar{\mathcal{W}} = \mathcal{W} \backslash \{j\}$ and $\boldsymbol{a}_r$ linearly independent. This particular case will be investigated in the following. The linear dependency of $\boldsymbol{A} = [\boldsymbol{a}_i]_{i \in \mathcal{W}}$ and $\boldsymbol{a}_r$ can be written as

$$\boldsymbol{a}_r = \sum_{i=1}^{m} \gamma_i \boldsymbol{a}_i = \boldsymbol{A}\gamma. \tag{4.101}$$

When multiplied with $\boldsymbol{p}$ we get

$$\boldsymbol{a}_r^T \boldsymbol{p} = \gamma^T \boldsymbol{A}^T \boldsymbol{p}, \tag{4.102}$$

and as $\boldsymbol{A}^T\boldsymbol{p} = \boldsymbol{0}$ (4.72b) we then have

$$\boldsymbol{a}_r^T\boldsymbol{p} = 0 \;\Leftrightarrow \boldsymbol{a}_r \in \text{span}\boldsymbol{A} = [\boldsymbol{a}_i]_{i\in\mathcal{W}}. \tag{4.103}$$

Now we will investigate what to do when

$$c_r(\boldsymbol{x}) < 0 \quad \wedge \quad \boldsymbol{a}_r \in \text{span}\boldsymbol{A} = [\boldsymbol{a}_i]_{i\in\mathcal{W}}. \tag{4.104}$$

When $\boldsymbol{a}_r$ and the constraints in the working set are linearly dependent and (4.69) is solved

$$\begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{v} \end{pmatrix} = \begin{pmatrix} \boldsymbol{a}_r \\ \boldsymbol{0} \end{pmatrix} \tag{4.105}$$

we know from (4.74b) and (4.103) that $\boldsymbol{p} = \boldsymbol{0}$. If $\boldsymbol{v}$ contains any negative values, t can be calculated using (4.71)

$$t = \min_{j:v_j<0} \frac{-\mu_j}{v_j} \geq 0, \qquad j = \arg\min_{j:v_j<0} \frac{-\mu_j}{v_j}. \tag{4.106}$$

When we move from $\boldsymbol{x}$ to $\bar{\boldsymbol{x}}$ we then have

$$\bar{\mu}_j = \mu_j + tv_j = 0 \tag{4.107}$$

and this is why we need to remove constraint c_j from $\mathcal{W}$ and we call the new set $\bar{\mathcal{W}} = \mathcal{W}\backslash\{j\}$. Now we will see that $\boldsymbol{a}_r$ is linearly independent of the vectors $\boldsymbol{a}_i$ for $i \in \bar{\mathcal{W}}$. This proof is done by contradiction. Lets assume that $\boldsymbol{a}_r$ is linearly dependent of the vectors $\boldsymbol{a}_i$ for $i \in \bar{\mathcal{W}}$, i.e. $\boldsymbol{a}_r \in \text{span}\boldsymbol{A} = [\boldsymbol{a}_i]_{i\in\bar{\mathcal{W}}}$.

As $\boldsymbol{a}_r \in \text{span}\boldsymbol{A} = [\boldsymbol{a}_i]_{i\in\mathcal{W}}$ and hence $\boldsymbol{p} = \boldsymbol{0}$ we can therefore write

$$\boldsymbol{a}_r = \boldsymbol{G}\boldsymbol{p} - \boldsymbol{A}\boldsymbol{v} = -\boldsymbol{A}\boldsymbol{v} = \boldsymbol{A}(-\boldsymbol{v}) = \sum_{i\in\mathcal{W}} \boldsymbol{a}_i(-v_i). \tag{4.108}$$

At the same time because $\mathcal{W} = \bar{\mathcal{W}} \cup \{j\}$, we have

$$\boldsymbol{a}_r = \sum_{i \in \bar{\mathcal{W}}} \boldsymbol{a}_i(-v_i) + \boldsymbol{a}_j(-v_j) \tag{4.109}$$

and isolation of $\boldsymbol{a}_j$ gives

$$\boldsymbol{a}_j = \frac{1}{-v_j}\boldsymbol{a}_r + \frac{1}{-v_j}\sum_{i \in \bar{\mathcal{W}}} \boldsymbol{a}_i v_i. \tag{4.110}$$

Since we assumed $\boldsymbol{a}_r \in \operatorname{span}\boldsymbol{A} = [\boldsymbol{a}_i]_{i \in \bar{\mathcal{W}}}$, using (4.101) $\boldsymbol{a}_r$ can be formulated as

$$\boldsymbol{a}_r = \sum_{i \in \bar{\mathcal{W}}} \boldsymbol{a}_i \gamma_i \tag{4.111}$$

and inserting this equation in (4.110) gives us

$$\boldsymbol{a}_j = \frac{1}{-v_j}\sum_{i \in \bar{\mathcal{W}}} \boldsymbol{a}_i \gamma_i + \frac{1}{-v_j}\sum_{i \in \bar{\mathcal{W}}} \boldsymbol{a}_i v_i, \tag{4.112}$$

which is equivalent to

$$\boldsymbol{a}_j = \sum_{i \in \bar{\mathcal{W}}} \frac{\gamma_i + v_i}{-v_j}\boldsymbol{a}_i. \tag{4.113}$$

As we have

$$\boldsymbol{a}_j = \sum_{i \in \bar{\mathcal{W}}} \beta_i \boldsymbol{a}_i, \quad \beta_i = \frac{\gamma_i + v_i}{-v_j} \tag{4.114}$$

clearly $\boldsymbol{a}_j$ is linearly dependent on the vectors $\boldsymbol{a}_i$ for $i \in \bar{\mathcal{W}}$ and this is a contradiction to the fact that $\boldsymbol{A} = [\boldsymbol{a}_i]_{i \in \mathcal{W}}$ has full column rank, i.e. the vectors $\boldsymbol{a}_i$ for $i \in \bar{\mathcal{W}} \cup j$ are linearly independent. This means that the assumption

$\boldsymbol{a}_r \in \text{span}\boldsymbol{A} = [\boldsymbol{a}_i]_{i\in\bar{\mathcal{W}}}$ cannot be true, and therefore we must conclude that $\boldsymbol{a}_r \notin \text{span}\boldsymbol{A} = [\boldsymbol{a}_i]_{i\in\bar{\mathcal{W}}}$.

Furthermore from (4.94) we know that

$$L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) = L(\boldsymbol{x}, \boldsymbol{\mu}) - \frac{1}{2}t^2\boldsymbol{a}_r^T\boldsymbol{p} - tc_r(\boldsymbol{x}) \tag{4.115}$$

and because of linear dependency $\boldsymbol{a}_r^T\boldsymbol{p} = 0$, this is equivalent to

$$L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) = L(\boldsymbol{x}, \boldsymbol{\mu}) - tc_r(\boldsymbol{x}) \tag{4.116}$$

and as $tc_r(\boldsymbol{x}) \leq 0$ we know that $L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) \geq L(\boldsymbol{x}, \boldsymbol{\mu})$ when $\boldsymbol{a}_r \in \text{span}\boldsymbol{A} = [\boldsymbol{a}_i]_{i\in\mathcal{W}}$.

Now we shall see what happens when no negative elements exist in $\boldsymbol{v}$ from (4.105). From (4.71) we know that t can be chosen as any non negative value, and therefore (4.116) becomes

$$\lim_{t\to\infty} L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) = \infty. \tag{4.117}$$

In this case the dual program is unbounded which means that the primal program (4.55) is infeasible. Proof of this is to be found in Jørgensen [13]. This means that no solution exists.

In short, what has been stated in this section, is that when (4.69) is solved and $\boldsymbol{a}_r^T\boldsymbol{p} = 0$ we know that $\boldsymbol{A} = [\boldsymbol{a}_i]_{i\in\mathcal{W}}$ and $\boldsymbol{a}_r$ are linearly dependent and $\boldsymbol{p} = \boldsymbol{0}$. If there are no negative elements in $\boldsymbol{v}$ the problem is infeasible and no solution exist. If some negative Lagrangian multipliers exist we should find constraint c_j from $\mathcal{W}$ where

$$j = \arg\min_{j:v_j<0} \frac{-\mu_j}{v_j}, \quad t = \min_{j:v_j<0} \frac{-\mu_j}{v_j} \geq 0 \tag{4.118}$$

and the following step is taken

$$\bar{\mu}_i = \mu_i + tv_i, \quad i \in \mathcal{W} \tag{4.119a}$$

$$\bar{\mu}_r = \mu_r + t. \tag{4.119b}$$

As $\boldsymbol{p} = \boldsymbol{0}$, we know that $\bar{\boldsymbol{x}} = \boldsymbol{x}$ and therefore this step is not mentioned in (4.119). Again, when $\bar{\mu}_j = 0$ it means that constraint c_j belongs to the dual active set $\mathcal{W}_{\mathcal{D}}$ and is therefore removed from $\mathcal{W}$. The constraints in the new working set $\bar{\mathcal{W}} = \mathcal{W} \backslash \{j\}$ and $\boldsymbol{a}_r$ are linearly independent, and as a result a new improving direction and step length may be calculated.

4.3.4 Starting Guess

One of the forces of the dual active set method is that a feasible starting point is easily calculated. Starting out with all constraints in the dual active set $\mathcal{W}_{\mathcal{D}}$ and therefore $\mathcal{W}$ being empty

$$\mu_i = 0, \quad i \in \mathcal{W}_{\mathcal{D}} = \mathcal{I}, \quad \mathcal{W} = \emptyset \tag{4.120}$$

and if we start in

$$\boldsymbol{x} = -\boldsymbol{G}^{-1}\boldsymbol{g} \tag{4.121}$$

(4.57b) and (4.57c) are satisfied

$$\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g} - \sum_{i \in \mathcal{I}} \boldsymbol{a}_i \mu_i = \boldsymbol{G}\boldsymbol{x} + \boldsymbol{g} = \boldsymbol{0} \tag{4.122a}$$

$$\mu_i \geq 0 \qquad i \in \mathcal{I}. \tag{4.122b}$$

The Lagrangian function is

$$\begin{aligned} L(\boldsymbol{x}, \boldsymbol{\mu}) &= \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x} - \sum_{i \in \mathcal{I}} \mu_i(\boldsymbol{a}_i^T\boldsymbol{x} - b_i) \\ &= \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x} \\ &= f(\boldsymbol{x}), \end{aligned} \tag{4.123}$$

which means that the starting point is at the minimum of the objective function of the primal program (4.55a) without taking notice of the constrains (4.55b).

Because the inverse Hessian matrix is used we must require the QP to be strictly convex.

4.3.5 In summary

Now we will summarize what has been discussed in this section and show how the dual active set method works. At iteration k we have $(\boldsymbol{x}, \boldsymbol{\mu}, r, \mathcal{W}, \mathcal{W}_{\mathcal{D}})$ where $c_r(\boldsymbol{x}) < 0$. Using the null space or the range space procedure the new improving direction is calculated by solving

$$\begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{v} \end{pmatrix} = \begin{pmatrix} \boldsymbol{a}_r \\ \boldsymbol{0} \end{pmatrix}, \quad \boldsymbol{A} = [\boldsymbol{a}_i]_{i\in\mathcal{W}}. \tag{4.124}$$

If $\boldsymbol{A} = [\boldsymbol{a}_i]_{i\in\mathcal{W}}$ and $\boldsymbol{a}_r$ are linearly dependent and no elements from $\boldsymbol{v}$ are negative the problem is infeasible and the method is terminated. Using (4.103) and (4.117) this is the case when

$$\boldsymbol{a}_r^T\boldsymbol{p} = 0 \quad \wedge \quad v_i \geq 0, \quad i \in \mathcal{W}. \tag{4.125}$$

If on the other hand $\boldsymbol{A} = [\boldsymbol{a}_i]_{i\in\mathcal{W}}$ and $\boldsymbol{a}_r$ are linearly dependent and some elements from $\boldsymbol{v}$ are negative, c_j is removed from $\mathcal{W}$, step length t is calculated according to (4.106) and a new step is taken

$$t = \min_{j:v_j<0} \frac{-\mu_j}{v_j} \geq 0, \quad j = \arg\min_{j:v_j<0} \frac{-\mu_j}{v_j} \tag{4.126a}$$

$$\bar{\mu}_i = \mu_i + tv_i \qquad i \in \mathcal{W} \tag{4.126b}$$

$$\bar{\mu}_r = \mu_r + t \tag{4.126c}$$

$$\mathcal{W} = \mathcal{W}\backslash\{j\}. \tag{4.126d}$$

If $\boldsymbol{A} = [\boldsymbol{a}_i]_{i\in\mathcal{W}}$ and $\boldsymbol{a}_r$ are linearly independent and some elements from $\boldsymbol{v}$ are negative, we calculate two step lengths t_1 and t_2 according to (4.71) and (4.98)

$$t_1 = \min(\infty, \min_{j:v_j<0} \frac{-\mu_j}{v_j}), \quad j = \arg\min_{j:v_j<0} \frac{-\mu_j}{v_j} \tag{4.127a}$$

$$t_2 = \frac{-c_r(\boldsymbol{x})}{\boldsymbol{a}_r^T\boldsymbol{p}}, \tag{4.127b}$$

where t_1 can be regarded as the step length in dual space because it assures that (4.57c) is satisfied whenever $0 \leq t \leq t_1$. Constraint (4.55b) is satisfied for

c_r when $t \geq t_2$ and therefore t_2 can be regarded as the step length in primal space. Therefore we will call t_1 t_D and t_2 t_P.

If $t_D < t_P$ then t_D is used as the step length and (4.57c) remain satisfied when we take the step. After taking the step, c_j is removed from $\mathcal{W}$ because $\bar{\mu}_j = 0$

$$\bar{\boldsymbol{x}} = \boldsymbol{x} + t_D \boldsymbol{p} \tag{4.128a}$$
$$\bar{\mu}_i = \mu_i + t_D v_i, \quad i \in \mathcal{W} \tag{4.128b}$$
$$\bar{\mu}_r = \mu_r + t_D \tag{4.128c}$$
$$\mathcal{W} = \mathcal{W} \backslash \{j\}. \tag{4.128d}$$

If $t_P \leq t_D$ then t_P is used as the step length and (4.55b) get satisfied for c_r. After taking the step we have that $c_r(\bar{\boldsymbol{x}}) = 0$ and therefore r is appended to $\mathcal{W}$

$$\bar{\boldsymbol{x}} = \boldsymbol{x} + t_P \boldsymbol{p} \tag{4.129a}$$
$$\bar{\mu}_i = \mu_i + t_P v_i, \quad i \in \mathcal{W} \tag{4.129b}$$
$$\bar{\mu}_r = \mu_r + t_P \tag{4.129c}$$
$$\mathcal{W} = \mathcal{W} \cup \{r\}. \tag{4.129d}$$

If $\boldsymbol{A} = [\boldsymbol{a}_i]_{i \in \mathcal{W}}$ and $\boldsymbol{a}_r$ are linearly independent and no elements from $\boldsymbol{v}$ are negative we have found the optimum and the program is terminated.

The procedure of the dual active set method is stated in algorithm 4.3.1.

Algorithm 4.3.1: Dual Active Set Algorithm for Strictly Convex Inequality Constrained QP's. **Note:** $\mathcal{W}_\mathcal{D} = \mathcal{I} \backslash \mathcal{W}$.

Compute $\boldsymbol{x}_0 = -\boldsymbol{G}^{-1}\boldsymbol{g}$, set $\mu_i = 0, i \in \mathcal{W}_D$ and $\mathcal{W} = \emptyset$.

while NOT STOP do

- **if** $c_i(\boldsymbol{x}) \geq 0 \; \forall i \in \mathcal{W}_D$ **then**
  - **STOP**, the optimal solution $\boldsymbol{x}^*$ has been found!
- Select $r \in \mathcal{W}_D : c_r(\boldsymbol{x}) < 0$.
- **while** $c_r(\boldsymbol{x}) < 0$ **do** `/* find improving direction` $\boldsymbol{p}$ `*/`
  - Find the improving direction $\boldsymbol{p}$ by solving the equality constrained QP:

    $$\begin{pmatrix} \boldsymbol{G} & -\boldsymbol{A} \\ -\boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{v} \end{pmatrix} = \begin{pmatrix} \boldsymbol{a}_r \\ \boldsymbol{0} \end{pmatrix}, \boldsymbol{A} = [\boldsymbol{a}_i]_{i \in \mathcal{W}}$$

  - **if** $\boldsymbol{a}_r^T \boldsymbol{p} = 0$ **then**
    - **if** $v_i \geq 0 \; \forall i \in \mathcal{W}$ **then**
      - **STOP**, the problem is infeasible!
    - **else** `/* compute step length` t`, remove constraint` j `*/`

      $$t = \min_{i \in \mathcal{W}: v_i < 0} \frac{-\mu_i}{v_i}, \mathcal{J} = \arg \min_{i \in \mathcal{W}: v_i < 0} \frac{-\mu_i}{v_i}$$

      - $\boldsymbol{x} \leftarrow \boldsymbol{x}$
      - $\mu_i \leftarrow \mu_i + t v_i, i \in \mathcal{W}$
      - $\mu_r \leftarrow \mu_r + t$
      - $\mathcal{W} \leftarrow \mathcal{W} \backslash \{j\}, j \in \mathcal{J}$
  - **else** `/* compute step length` t_D `and` t_P `*/`

    $$t_D = \min \left(\infty, \min_{i \in \mathcal{W}: v_i < 0} \frac{-\mu_i}{v_i} \right), \mathcal{J} = \arg \min_{i \in \mathcal{W}: v_i < 0} \frac{-\mu_i}{v_i}$$

    $$t_P = \frac{-c_r(\boldsymbol{x})}{\boldsymbol{a}_r^T \boldsymbol{p}}$$

    - **if** $t_P \leq t_D$ **then** `/* append constraint` r `*/`
      - $\boldsymbol{x} \leftarrow \boldsymbol{x} + t_P \boldsymbol{p}$
      - $\mu_i \leftarrow \mu_i + t_P v_i, i \in \mathcal{W}$
      - $\mu_r \leftarrow \mu_r + t_P$
      - $\mathcal{W} \leftarrow \mathcal{W} \cup \{r\}$
    - **else** `/* remove constraint` j `*/`
      - $\boldsymbol{x} \leftarrow \boldsymbol{x} + t_D \boldsymbol{p}$
      - $\mu_i \leftarrow \mu_i + t_D v_i, i \in \mathcal{W}$
      - $\mu_r \leftarrow \mu_r + t_D$
      - $\mathcal{W} \leftarrow \mathcal{W} \backslash \{j\}, j \in \mathcal{J}$

4.3.6 Termination

The dual active set method does not have the ability to cycle as it terminates in a finite number of steps. This is one of the main forces of the method, and therefore we will now investigate this property.

As the algorithm (4.3.1) suggests, the method mainly consists of two while-loops which we call outer-loop and inner-loop. In the outer-loop we test if optimality has been found. If this is not the case we choose some violated constraint r, $c_r(\boldsymbol{x}) < 0$ and move to the inner-loop.

At every iteration of the inner-loop we calculate a new improving direction and a corresponding step length: $t = min(t_D, t_P)$, where t_D is the step length in dual space and t_P is the step length in primal space. The step length in primal space is always positive, $t_P > 0$ as $t_P = \frac{-c_r(\boldsymbol{x})}{\boldsymbol{a}_r^T \boldsymbol{p}}$, where $c_r(\boldsymbol{x}) < 0$ and $\boldsymbol{a}_r^T \boldsymbol{p} > 0$. From (4.95) and (4.96) we know that $L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\mu}}) > L(\boldsymbol{x}, \boldsymbol{\mu})$ whenever $0 < t < \frac{-2c_r(\boldsymbol{x})}{\boldsymbol{a}_r^T \boldsymbol{p}}$. This means that the dual objective function L increases when a step in primal space is taken. A step in primal space also means that we leave the inner-loop as constraint c_r is satisfied $c_r(\bar{\boldsymbol{x}}) = c_r(\boldsymbol{x}) + t\boldsymbol{a}_r^T \boldsymbol{p} = 0$.

A step in dual space is taken whenever $t_D < t_P$ and in this case we have $c_r(\bar{\boldsymbol{x}}) = c_r(\boldsymbol{x}) + t_D \boldsymbol{a}_r^T \boldsymbol{p} < c_r(\boldsymbol{x}) + t_P \boldsymbol{a}_r^T \boldsymbol{p} = 0$. This means that we will never leave the inner-loop after a step in dual space as $c_r(\bar{\boldsymbol{x}}) < 0$. A constraint c_j is removed from the working set $\mathcal{W}$ when we take a step in dual space, which means that $|\mathcal{W}|$ is the maximum number of steps in dual space that can be taken in succession. After a sequence of $0 \leq s \leq |\mathcal{W}|$ steps in dual space, a step in primal space will cause us to leave the inner-loop. This step in primal space guarantees that L is strictly larger when we leave the inner-loop than when we entered it.

As the constraints in the working set $\mathcal{W}$ are linearly independent at any time, the corresponding solution $(\boldsymbol{x}, \boldsymbol{\mu})$ is unique. Also as $L(\boldsymbol{x}^{q+1}, \boldsymbol{\mu}^{q+1}) > L(\boldsymbol{x}^q, \boldsymbol{\mu}^q)$ (where q defines the $q'th$ iteration of the outer loop) we know that the combination of constraints in $\mathcal{W}$ is unique for any iteration q. And because the number of different ways the working set can be chosen from $\mathcal{I}$ is finite and bounded by $2^{|\mathcal{I}|}$, we know that the method will terminate in a finite number of iterations.

4.4 Dual active set method by example

In the following example we will demonstrate how the dual active set method finds the optimum

$$
\begin{aligned}
\min_{\boldsymbol{x}\in\mathbb{R}^n} f(\boldsymbol{x}) =& \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x}, \quad \boldsymbol{G} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \boldsymbol{g} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\
\text{s.t.} \qquad & c_1 = -x_1 + x_2 - 1 \geq 0 \\
& c_2 = -\frac{1}{2}x_1 - x_2 + 2 \geq 0 \\
& c_3 = -x_2 + 2.5 \geq 0 \\
& c_4 = -3x_1 + x_2 + 3 \geq 0.
\end{aligned}
$$

At every iteration k we plot the path $(\boldsymbol{x}^1 \ldots \boldsymbol{x}^k)$ together with the constraints, where active constraints are indicated with red. The 4 constraints and their column-index in $\boldsymbol{A}$ are labeled on the constraint in the plots. The primal feasible area is in the top right corner where the plot is lightest. We use the least negative value of $c(\boldsymbol{x}^k)$ every time c_r is chosen, even though any negative constraint could be used. For every iteration we have plotted the situation when we enter the while loop.

Iteration 1

This situation is illustrated in figure 4.4. The starting point is at $\boldsymbol{x} = \boldsymbol{G}^{-1}\boldsymbol{g} = (0, 0)^T$, $\boldsymbol{\mu} = \boldsymbol{0}$ and $\mathcal{W} = \emptyset$. On entering the while loop we have $c(\boldsymbol{x}) = [1.0, -2.0, -2.5, -3.0]^T$ and therefore $r = 2$ is chosen because the second element is least negative. The working set is empty and therefore the new improving direction is found to $\boldsymbol{p} = [0.5, 1.0]^T$ and $\boldsymbol{u} = []$. As the step length in primal space is $t_P = 1.6$ and the step length in dual space is $t_D = \infty$, t_P is used, and therefore r is appended to the working set. The step is taken as seen in figure 4.5.

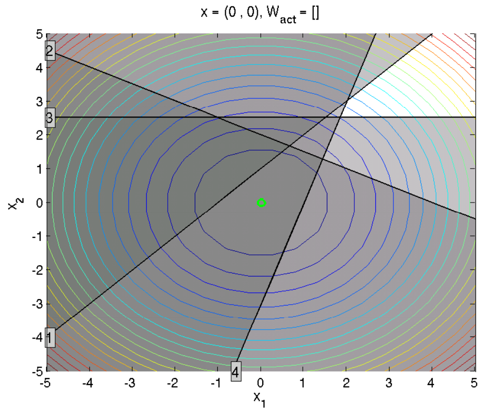


Figure 4.4: Iteration 1, $\mathcal{W} = \emptyset$, $\boldsymbol{x} = [0, 0]^T$, $\boldsymbol{\mu} = [0, 0, 0, 0]^T$.

Iteration 2

This situation is illustrated in figure 4.5. On entering the while loop we have $\mathcal{W} = [2]$, and because $c_r(\boldsymbol{x}) = 0$ we should choose a new r. As $c(\boldsymbol{x}) = [0.2, 0, -0.9, -2.2]^T$, $r = 3$ is chosen. The new improving direction is found to be $\boldsymbol{p} = [-0.4, 0.2]^T$ and $\boldsymbol{u} = [-0.8]$. As $t_D = 2.0$ and $t_P = 4.5$ a step in dual space is taken and c_2 is removed from $\mathcal{W}$.

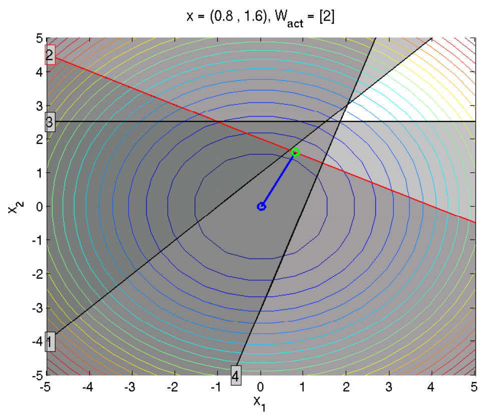


Figure 4.5: Iteration 2, $\mathcal{W} = [2]$, $\boldsymbol{x} = [0.8, 1.6]^T$, $\boldsymbol{\mu} = [0, 1.6, 0, 0]^T$.

Iteration 3

This situation is illustrated in figure 4.6. Because $c_r(\boldsymbol{x}) \neq 0$ we keep $r = 3$. The working set is empty $\mathcal{W} = \emptyset$. The improving direction is found to be $\boldsymbol{p} = [0.0, 1.0]^T$ and $\boldsymbol{u} = []$. A step in primal space is taken as $t_D = \infty$ and

$t_P = 0.5$ and r is appended to $\mathcal{W}$.

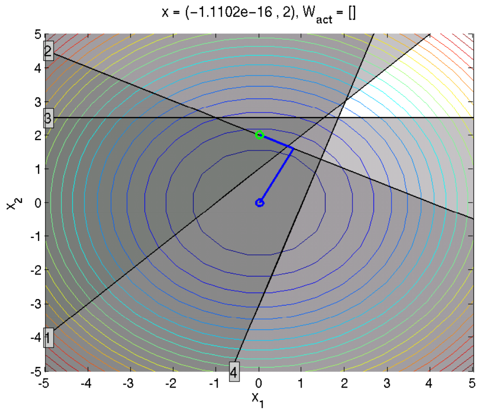


Figure 4.6: Iteration 3, $\mathcal{W} = []$, $\boldsymbol{x} = [0, 2]^T$, $\boldsymbol{\mu} = [0, 0, 2, 0]^T$.

Iteration 4

This situation is illustrated in figure 4.7. Now $c_r(\boldsymbol{x}) = 0$ and therefore a new r should be chosen. As $c(\boldsymbol{x}) = [-1.5, 0.5, 0, -5.5]^T$, we choose $r = 1$. The working set is $\mathcal{W} = [3]$, and the new improving direction is $\boldsymbol{p} = [1.0, 0.0]^T$ and $\boldsymbol{u} = [1]$. We use t_P as $t_D = \infty$ and $t_P = 1.5$ and r is appended to $\mathcal{W}$ after taking the step.

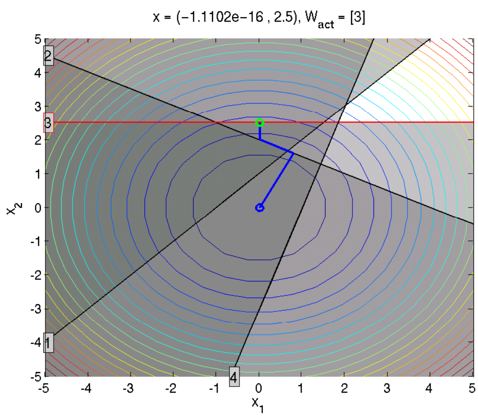


Figure 4.7: Iteration 4, $\mathcal{W} = [3]$, $\boldsymbol{x} = [0, 2.5]^T$, $\boldsymbol{\mu} = [0, 0, 2.5, 0]^T$.

Iteration 5

This situation is illustrated in figure 4.8. As $c_r(\boldsymbol{x}) = 0$ we must choose a new r and as $c(\boldsymbol{x}) = [0, 1.25, 0, -1.0]^T$, $r = 4$ is chosen. At this point $\mathcal{W} = [3, 1]$.

The new improving direction is $\boldsymbol{p} = [0.0, 0.0]^T$ and $\boldsymbol{u} = [-2, -3]$ and therefore $\boldsymbol{a}_r^T \boldsymbol{p} = 0$. This means that $\boldsymbol{a_r}$ is linearly dependent of the constraints in $\mathcal{W}$ and therefore 1 is removed from $\mathcal{W}$.

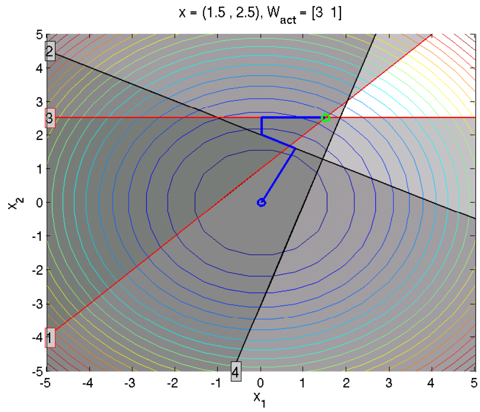


Figure 4.8: Iteration 5, $\mathcal{W} = [3, 1]$, $\boldsymbol{x} = [1.5, 2.5]^T$, $\boldsymbol{\mu} = [1.5, 0, 4, 0]^T$.

Iteration 6

This situation is illustrated in figure 4.9. On entering the while loop we have c_3 in the working set $\mathcal{W} = [3]$. And r remains 4 because $c_r(\boldsymbol{x}) \neq 0$. The new improving direction is $\boldsymbol{p} = [3.0, 0.0]^T$ and $\boldsymbol{u} = [1]$. The step lengths are $t_D = \infty$ and $t_P = 0.11$ and therefore a step in primal space is taken and r is appended to $\mathcal{W}$.

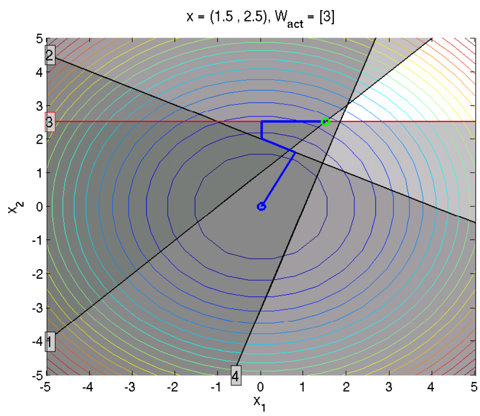


Figure 4.9: Iteration 6, $\mathcal{W} = [3]$, $\boldsymbol{x} = [1.5, 2.5]^T$, $\boldsymbol{\mu} = [0, 0, 3, 0.5]^T$.

Iteration 7

This situation is illustrated in figure 4.10. Now the working set is $\mathcal{W} = [3, 4]$ and as $c_r(\boldsymbol{x}) = 0$, a new r must be chosen. But $c(\boldsymbol{x}) = [0.33, 1.42, 0, 0]^T$ (no negative elements) and therefore the global optimal solution has been found and the algorithm is terminated. The optimal solution is $\boldsymbol{x}^* = [1.83, 2.50]^T$ and $\boldsymbol{\mu}^* = [0, 0, 3.11, 0.61]^T$.

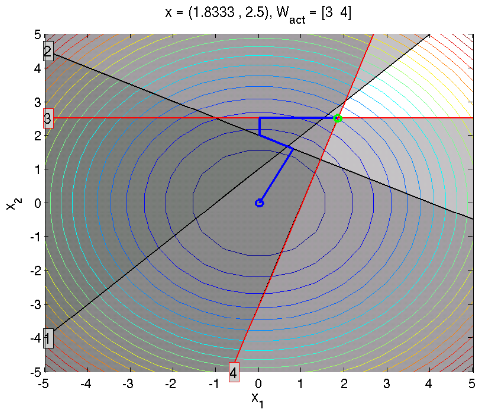


Figure 4.10: Iteration 7, $\mathcal{W} = [3, 4]$, $\boldsymbol{x}^* = [1.8, 2.5]^T$, $\boldsymbol{\mu}^* = [0, 0, 3.11, 0.61]^T$.

An interactive demo application `QP_demo.m` is found in appendix D.5.

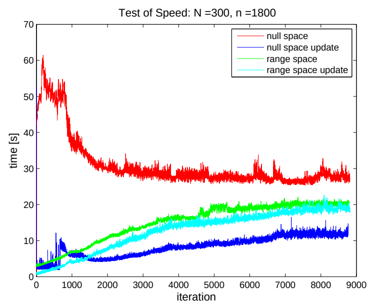
Chapter 5

Test and Refinements

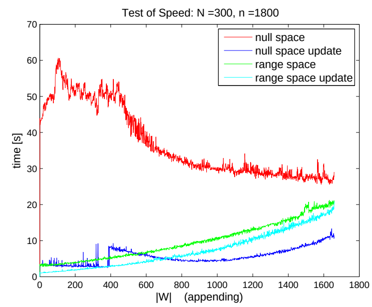
When solving an inequality constrained convex QP, we use either the primal active set method or the dual active set method. In both methods we solve a sequence of KKT systems, where each KKT system correspond to an equality constrained QP. To solve the KKT system we use one of four methods: The range space procedure, the null space procedure, or one of the two with factorization update instead of complete factorizations. We will test these four methods for computational speed to find out how they perform compared to each other.
Usually the constraints in an inequality constrained QP are divided into bounded variables and general constraints. This division can be used to further optimization of the factorization updates, as we will discuss later in this chapter. As a test case we will use the quadruple tank problem which is described in appendix A.

5.1 Computational Cost of the Range and the Null Space Procedures with Update

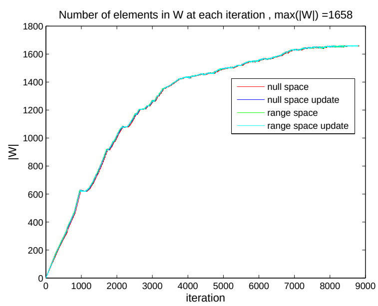
The active set methods solve a sequence of KKT systems by use of the range space procedure, the null space procedure or one of the two with factorization update. Now we will compare the performance of these methods by solving the quadruple tank problem. By discretizing with N = 300 we define an inequality constrained strictly convex QP with $n = 1800$ variables and $|\mathcal{I}| = 7200$ constraints. We have chosen to use the dual active set method because it does not need a precalculated starting point. Different parts of the process are illustrated in figure 5.1. Figure 5.1(a) shows the computational time for solving the KKT system for each iteration and figure 5.1(c) shows the number of active constraints $|W|$ for each iteration. The size of the active set grows rapidly in the first third of the iterations after which this upward movement fades out a little. This explains why the computational time for the null space procedure decreases fast to begin with and then fades out, as it is proportional to the size of the null space $(n - m)$. Likewise, the computational time for the range space procedure grows proportional to the dimension of the range space (m). The null space procedure with factorization update is much faster than the null space procedure with complete factorization even though some disturbance is observed in the beginning. This disturbance is probably due to the fact that the testruns are carried out on shared servers. The range space procedure improves slightly whenever factorization update is used. When solving this particular problem, it is clear from figures 5.1(a) and 5.1(c), that range space procedure with factorization update should be used until approximately 800 constraints are active, corresponding to $\frac{800}{1800}n = \simeq 0.45n$, after which the null space procedure with factorization update should be used. In theory the total number of iterations should be exactly the same for all four methods, however they differ a little due to numerical instability as seen in figure 5.1(c), where the curves are not completely aligned. The number of active constraints at the optimal solution is $|W| = 1658$.



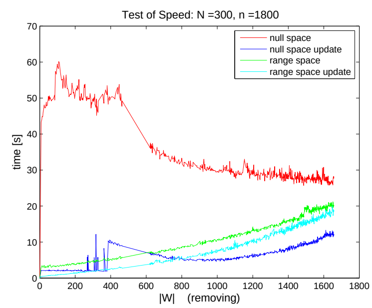
(a) Computational time for solving the KKT system plotted for each iteration.



(b) Computational time for solving the KKT system each time a constraint is appended to the active set W.



(c) Number of active constraints plotted at each iteration.



(d) Computational time for solving the KKT system each time a constraint is removed from the active set W.

Figure 5.1: The process of solving the quadruple tank problem with N=300, (1800 variables and 7200 constraints). Around 8800 iterations are needed (depending on the method) and the number of active constraints at the solution is $|W| = 1658$.

5.2 Fixed and Free Variables

So far, we have only considered constraints defined like $\boldsymbol{a}_i^T\boldsymbol{x} \geq b_i$. Since some of the constraints are likely to be bounds on variables $x_i \geq b_i$, we divide all constraints into bounds and general constraints. The structure of our QP solver is then defined as follows

$$\begin{aligned}
&\min_{\boldsymbol{x}\in\mathbb{R}^n} && \frac{1}{2}\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{x} && && (5.1a)\\
&\text{s.t.} && l_i \leq x_i \leq u_i && i \in \mathcal{I}_b = 1, 2, ..., n && (5.1b)\\
& && (b_l)_i \leq \boldsymbol{a}_i^T\boldsymbol{x} \leq (b_u)_i && i \in \mathcal{I}_{gc} = 1, 2, ..., m_{gc} && (5.1c)
\end{aligned}$$

where $\mathcal{I}_b$ is the set of bounds and $\mathcal{I}_{gc}$ is the set of general constraints. This means that we have upper and lower limits on every bound and on every general constraint, so that the total number of constraints is $|\mathcal{I}| = 2n + 2m_{gc}$. We call the active constraint matrix $\boldsymbol{C} \in \mathbb{R}^{n\times m}$ and it contains both active bounds and active general constraints. Whenever a bound is active we say that the corresponding variable x_i is fixed. By use of a permutation matrix $\boldsymbol{P} \in \mathbb{R}^{n\times n}$ we organize $\boldsymbol{x}$ and $\boldsymbol{C}$ in the following manner

$$\begin{pmatrix} \tilde{\boldsymbol{x}} \\ \hat{\boldsymbol{x}} \end{pmatrix} = \boldsymbol{P}\boldsymbol{x}, \quad \begin{pmatrix} \tilde{\boldsymbol{C}} \\ \hat{\boldsymbol{C}} \end{pmatrix} = \boldsymbol{P}\boldsymbol{C} \tag{5.2}$$

where $\tilde{\boldsymbol{x}} \in \mathbb{R}^{\tilde{n}}$, $\hat{\boldsymbol{x}} \in \mathbb{R}^{\hat{n}}$, $\tilde{\boldsymbol{C}} \in \mathbb{R}^{\tilde{n}\times m}$ and $\hat{\boldsymbol{C}} \in \mathbb{R}^{\hat{n}\times m}$, $\tilde{n}$ is the number of free variables and $\hat{n}$ is the number of fixed variables ($\hat{n} = n - \tilde{n}$). Now we reorganize the active constraint matrix $\boldsymbol{P}\boldsymbol{C}$

$$\boldsymbol{P}\boldsymbol{C} = \boldsymbol{P}\,(\boldsymbol{B}\ \boldsymbol{A}) = \begin{pmatrix} \boldsymbol{0} & \tilde{\boldsymbol{A}} \\ \boldsymbol{I} & \hat{\boldsymbol{A}} \end{pmatrix} \tag{5.3}$$

where $\boldsymbol{B} \in \mathbb{R}^{n\times\hat{n}}$ contains the bounds and $\boldsymbol{A} \in \mathbb{R}^{n\times(m-\hat{n})}$ contains the general constraints. So we have $\tilde{\boldsymbol{A}} \in \mathbb{R}^{\tilde{n}\times(m-\hat{n})}$ and $\hat{\boldsymbol{A}} \in \mathbb{R}^{\hat{n}\times(m-\hat{n})}$ and $\boldsymbol{I} \in \mathbb{R}^{\hat{n}\times\hat{n}}$ as the identity matrix.

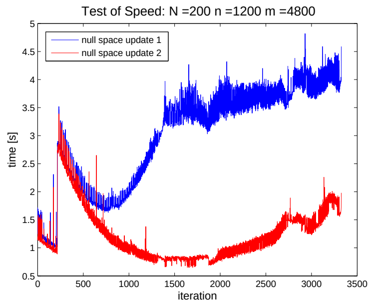
The QT factorization (which is used in the null space procedure) of (5.3) is defined as

$$\boldsymbol{Q} = \begin{pmatrix} \tilde{\boldsymbol{Q}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I} \end{pmatrix}, \quad \boldsymbol{T} = \begin{pmatrix} \boldsymbol{0} & \tilde{\boldsymbol{T}} \\ \boldsymbol{I} & \hat{\boldsymbol{A}} \end{pmatrix} \tag{5.4}$$

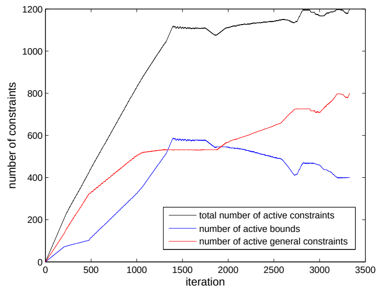
where $\boldsymbol{Q} \in \mathbb{R}^{n \times n}$, $\tilde{\boldsymbol{Q}} \in \mathbb{R}^{\tilde{n} \times \tilde{n}}$, $\boldsymbol{I} \in \mathbb{R}^{\hat{n} \times \hat{n}}$, $\boldsymbol{T} \in \mathbb{R}^{n \times m}$ and $\tilde{\boldsymbol{T}} \in \mathbb{R}^{\tilde{n} \times (m-\hat{n})}$, Gill *et al.* [9]. This is a modified QT factorization, as only $\tilde{\boldsymbol{T}}$ in $\boldsymbol{T}$ is lower triangular.

The part of the QT factorization which corresponds to the free variables consists of $\tilde{\boldsymbol{Q}}$ and $\tilde{\boldsymbol{T}}$. From (5.4) it is clear that this is the only part that needs to be updated whenever a constraint is appended to or removed from the active set. The details of how these updates are carried out can be found in Gill *et al.* [9] but the basic idea is similar to the one described in chapter 3. The QT structure is obtained using givens rotations on specific parts of the modified QT factorization after appending or removing a constraint.

We have implemented these updates. To find out how performance may be improved we have plotted the computational speed when solving the quadruple tank problem with N = 200, defining 1200 variables and 4800 constraints. We tested both the null space procedure with the factorization update as described in chapter 3 and the null space procedure with factorization update based on fixed and free variables. From figure 5.2 it is clear that the recent update has made a great improvement in computational time. But of course the improvement is dependent on the number of active bounds in the specific problem.



(a) Computational time for solving the KKT system plotted for each iteration. Null space update 2 is the new update based on fixed and free variables.



(b) The number of active bounds and active general constraints and the sum of the two plotted at each iteration.

Figure 5.2: Computational time and the corresponding number of active bounds and active general constraints plotted for each iteration when solving the quadruple tank problem with N=200, (1200 variables and 4800 constraints). The problem is solved using both the null space update and the null space update based on fixed and free variables.

5.3 Corresponding Constraints

In our implementation we only consider inequality constraints, and they are organized as shown in (5.1), where bounds and general constraints are connected in pairs. So all constraints, by means all bounds and all general constraints together, indexed i, are organized in $\mathcal{I}$ as follows

$$
\begin{aligned}
i \in \mathcal{I} = \{ & \underbrace{1, 2, ..., n,}_{x \geq l} && (5.5)\\
& \underbrace{n+1, n+2, ..., 2n,}_{-x \geq -u} && (5.6)\\
& \underbrace{2n+1, 2n+2, ..., 2n+m_{gc},}_{\boldsymbol{a}^T\boldsymbol{x} \geq b_l} && (5.7)\\
& \underbrace{2n+m_{gc}+1, 2n+m_{gc}+2, ..., 2n+2m_{gc}\}}_{-\boldsymbol{a}^T\boldsymbol{x} \geq -b_u} && (5.8)
\end{aligned}
$$

and the corresponding pairs, indexed p, are then organized in $\mathcal{P}$ in the following manner

$$p \in \mathcal{P} = \{\underbrace{n+1, n+2, ..., 2n,}_{-x \geq -u} \tag{5.9}$$

$$\underbrace{1, 2, ..., n,}_{x \geq l} \tag{5.10}$$

$$\underbrace{2n + m_{gc} + 1, 2n + m_{gc} + 2, ..., 2n + 2m_{gc},}_{-\boldsymbol{a}^T\boldsymbol{x} \geq -b_u} \tag{5.11}$$

$$\underbrace{2n + 1, 2n + 2, ..., 2n + m_{gc}}_{\boldsymbol{a}^T\boldsymbol{x} \geq b_l}\}. \tag{5.12}$$

$$\tag{5.13}$$

Unbounded variables and unbounded general constraints, where the upper and/or lower limits are $\pm\infty$ respectively, are never violated. So they are not considered, when $\mathcal{I}$ and $\mathcal{P}$ are initialized. E.g. if $l_2 = -\infty$, then $i = 2$ will not exist in $\mathcal{I}$, and $p_2 = n + 2$ will not exist in $\mathcal{P}$.

In practice, the primal and the dual active set methods are implemented using two sets, the active set given as the working set $\mathcal{W}_k$ and the inactive set $\mathcal{I}\backslash\mathcal{W}_k$. When a constraint $j \in \mathcal{I}\backslash\mathcal{W}_k$ becomes active, it is appended to the active set

$$\mathcal{W}_{k+1} = \mathcal{W}_k \cup \{j\} \tag{5.14}$$

and because two corresponding inequality constraints cannot be active at the same time, it is removed together with its corresponding pair $p_j \in \mathcal{P}$ from the inactive set as follows

$$\mathcal{I}\backslash\mathcal{W}_{k+1} = \{\mathcal{I}\backslash\mathcal{W}_k\}\backslash\{j, p_j\}. \tag{5.15}$$

When it becomes inactive it is removed from the active set

$$\mathcal{W}_{k+1} = \mathcal{W}_k\backslash\{j\} \tag{5.16}$$

and appended to the inactive set together with its corresponding pair

$$\mathcal{I}\backslash\mathcal{W}_{k+1} = \{\mathcal{I}\backslash\mathcal{W}_k\} \cup \{j, p_j\}. \tag{5.17}$$

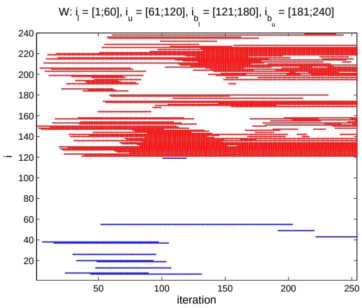
So by using corresponding pairs, we have two constraints less to examine feasibility for, every time a constraint is found to be active.

Besides the gain of computational speed, the stability of the dual active set method is also increased. Equality constraints are given as two inequalities with the same value as upper and lower limits. So because of numerical instabilities, the method tends to append corresponding constraints to the active set, when it is close to the solution. If this is the case, the constraint matrix $\boldsymbol{A}$ becomes linearly depended, and the dual active set method terminates because of infeasibility. But by removing the corresponding pairs from the inactive set, this problem will never occur. The primal active set method will always find the solution before the possibility of two corresponding constraints becomes active simultaneously, so for this method we gain computational speed only.

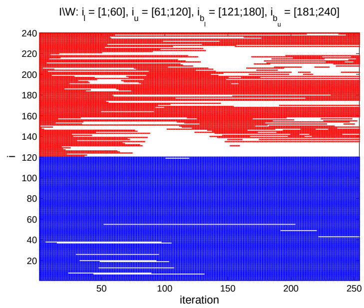
The quadruple tank problem is now solved, see figure 5.3, without removing the corresponding pairs from the inactive set - so only the active constraints are removed.

In figure 5.3(a) and 5.3(b) we see the indices of the constraints of the active set $\mathcal{W}_k$ and the inactive set $\mathcal{I}\backslash\mathcal{W}_k$ respectively. Not surprisingly it is seen, that the constraints currently in the active set are missing in the inactive set. Also in figure 5.3(c) and 5.3(d) we see, that the relation between the number of constraints in the active set and the number of constraints in the inactive set as expected satisfy

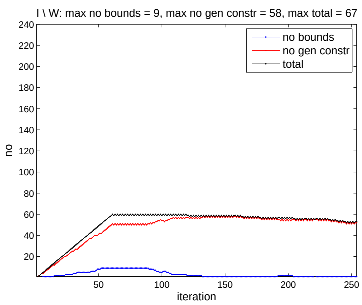
$$|\mathcal{W}_k| + |\mathcal{I}\backslash\mathcal{W}_k| = |\mathcal{I}|. \tag{5.18}$$



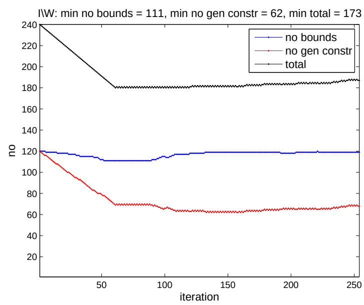
(a) Indices of active bounds (blue) and active general constraints (red) per iteration.



(b) Indices of inactive bounds (blue) and inactive general constraints (red) per iteration.



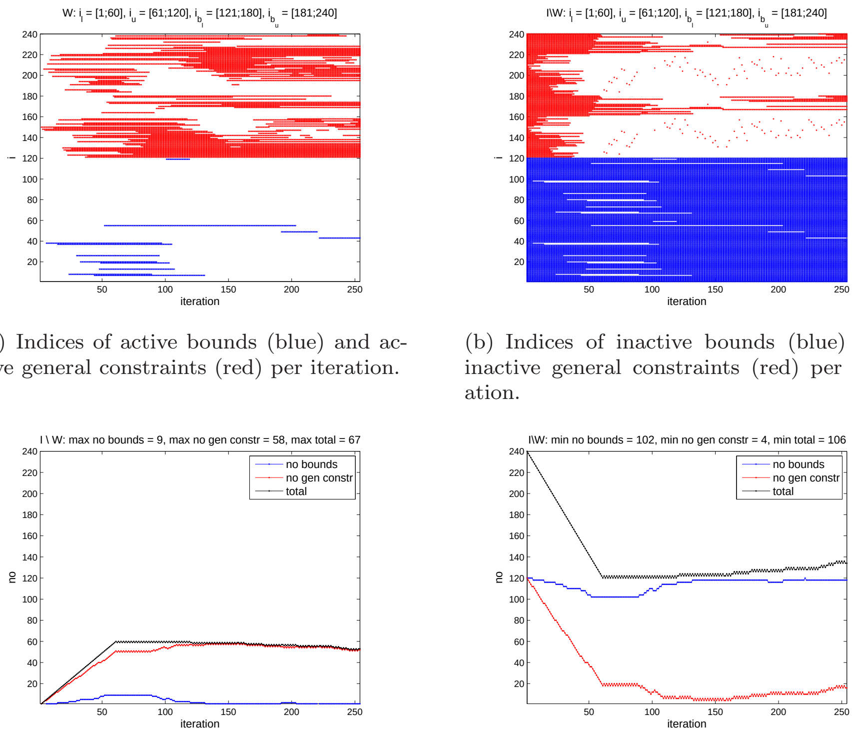
(c) Number of active bounds and general constraints per iteration.



(d) Number of inactive bounds and general constraints per iteration.

Figure 5.3: The process of solving the quadruple tank problem using the primal active set method with $N = 10$, so $n = 60$ and $|\mathcal{I}| = 240$, without removing the corresponding pairs from the inactive set $\mathcal{I}\backslash\mathcal{W}_k$.

The quadruple tank problem is now solved again, see figure 5.4, but this time we remove the corresponding pairs from the inactive set as well.



(a) Indices of active bounds (blue) and active general constraints (red) per iteration.

(b) Indices of inactive bounds (blue) and inactive general constraints (red) per iteration.

(c) Number of active bounds and general constraints per iteration.

(d) Number of inactive bounds and general constraints per iteration.

Figure 5.4: The process of solving the quadruple tank problem using the primal active set method with $N = 10$, so $n = 60$ and $|\mathcal{I}| = 240$, showing the effect of removing the corresponding pairs from the inactive set $\mathcal{I}\backslash\mathcal{W}_k$.

In figure 5.4(a) the indices of the constraints in the active set $\mathcal{W}_k$ are the same as before the removal of the corresponding pairs. And in figure 5.4(b) we now see, that all active constraints and their corresponding pairs are removed from the inactive set $\mathcal{I}\backslash\mathcal{W}_k$, and the set is seen to be much more sparse. The new relation between the number of constraints in the active set and the number of constraints in the inactive set is seen in figure 5.4(c) and 5.4(d). And the indices of the constraints in the inactive set, when we also remove the corresponding pairs, are found to be $\{\mathcal{I}\backslash\mathcal{W}_k\}\backslash\{p_i\}, i \in \mathcal{W}_k$. So now we have the new relation described as follows

$$2|\mathcal{W}_k| + |\{\mathcal{I}\backslash\mathcal{W}_k\}\backslash\{p_i\}| = |\mathcal{I}|, \quad i \in \mathcal{W}_k. \tag{5.19}$$

The size of $\{\mathcal{I}\backslash\mathcal{W}_k\}\backslash\{p_i\}, i \in \mathcal{W}_k$ is found by combining (5.18) and (5.19) as follows

$$2|\mathcal{W}_k| + |\{\mathcal{I}\backslash\mathcal{W}_k\}\backslash\{p_i\}| = |\mathcal{W}_k| + |\mathcal{I}\backslash\mathcal{W}_k|, \quad i \in \mathcal{W}_k \tag{5.20}$$

which leads to

$$|\{\mathcal{I}\backslash\mathcal{W}_k\}\backslash\{p_i\}| = |\mathcal{I}\backslash\mathcal{W}_k| - |\mathcal{W}_k|, \quad i \in \mathcal{W}_k. \tag{5.21}$$

So we see, that the inactive set overall is reduced twice the size of the active set by also removing all corresponding constraints $p_i, i \in \mathcal{W}_k$, from the inactive set. This is also seen by comparing figure 5.4(c) and 5.3(c).

5.4 Distinguishing Between Bounds and General Constraints

In both the primal and the dual active set methods some computations involving the constraints are made, e.g. checking the feasibility of the constraints. All constraints in $\mathcal{I}$ are divided into bounds and general constraints, and via the indices $i \in \mathcal{I}$ it is easy to distinguish, if a constraint is a bound or a general constraint. This can be exploited to gain some computational speed, since computations regarding a bound only involve the fixed variable, and therefore it is very cheap to carry out.

Chapter 6

Nonlinear Programming

In this chapter we will investigate how nonlinear convex programs with nonlinear constraints can be solved by solving a sequence of QP's. The nonlinear program is solved using Newton's method and the calculation of a Newton step can be formulated as a QP and found using a QP solver. As Newton's method solves a nonlinear program by a sequence of Newton steps, this method is called sequential quadratic programming (SQP).

6.1 Sequential Quadratic Programming

Each step of Newton's method is found by solving a QP. The theory is based on the work of Nocedal and Wright [14] and Jørgensen [15]. To begin with, we will focus on solving the equality constrained nonlinear program

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} \quad f(\boldsymbol{x}) \tag{6.1a}$$

$$\text{s.t.} \quad h(\boldsymbol{x}) = \mathbf{0} \tag{6.1b}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ and $h(\boldsymbol{x}) \in \mathbb{R}^m$. This is done using the corresponding Lagrangian function

$$L(\boldsymbol{x}, \boldsymbol{y}) = f(\boldsymbol{x}) - \boldsymbol{y}^T h(\boldsymbol{x}). \tag{6.2}$$

The optimum is found by solving the corresponding KKT system

$$\nabla_x L(\boldsymbol{x}, \boldsymbol{y}) = \nabla f(\boldsymbol{x}) - \nabla h(\boldsymbol{x})\boldsymbol{y} = \mathbf{0} \tag{6.3a}$$
$$\nabla_y L(\boldsymbol{x}, \boldsymbol{y}) = -h(\boldsymbol{x}) = \mathbf{0}. \tag{6.3b}$$

The KKT system is written as a system of nonlinear equations as follows

$$\begin{aligned} F(\boldsymbol{x}, \boldsymbol{y}) &= \begin{pmatrix} F_1(\boldsymbol{x}, \boldsymbol{y}) \\ F_2(\boldsymbol{x}, \boldsymbol{y}) \end{pmatrix} \\ &= \begin{pmatrix} \nabla_x L(\boldsymbol{x}, \boldsymbol{y}) \\ \nabla_y L(\boldsymbol{x}, \boldsymbol{y}) \end{pmatrix} \\ &= \begin{pmatrix} \nabla f(\boldsymbol{x}) - \nabla h(\boldsymbol{x})\boldsymbol{y} \\ -h(\boldsymbol{x}) \end{pmatrix} = \mathbf{0}. \end{aligned} \tag{6.4}$$

Newton's method is used to solve this system. Newton's method approximates the root of a given function $g(\boldsymbol{x})$ by taking successive steps in the direction of $\nabla g(\boldsymbol{x})$. A Newton step is calculated like this

$$g(\boldsymbol{x}^k) + J(\boldsymbol{x}^k)\Delta\boldsymbol{x} = \mathbf{0}, \qquad J(\boldsymbol{x}^k) = \nabla g(\boldsymbol{x}^k)^T. \tag{6.5}$$

As we want to solve (6.4) using Newton's method, we need the gradient of $F(\boldsymbol{x}, \boldsymbol{y})$ which is given by

$$\begin{aligned} \nabla F(\boldsymbol{x}, \boldsymbol{y}) &= \nabla \begin{pmatrix} F_1(\boldsymbol{x}, \boldsymbol{y}) \\ F_2(\boldsymbol{x}, \boldsymbol{y}) \end{pmatrix} \\ &= \begin{pmatrix} \frac{\partial F_1}{\partial \boldsymbol{x}_1} & \frac{\partial F_2}{\partial \boldsymbol{x}_1} \\ \frac{\partial F_1}{\partial \boldsymbol{y}_2} & \frac{\partial F_2}{\partial \boldsymbol{y}_2} \end{pmatrix} \\ &= \begin{pmatrix} \nabla_{xx}^2 L(\boldsymbol{x}, \boldsymbol{y}) & -\nabla h(\boldsymbol{x}) \\ -\nabla h(\boldsymbol{x})^T & \mathbf{0} \end{pmatrix}, \end{aligned} \tag{6.6}$$

where $\nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y})$ is the Hessian of $L(\boldsymbol{x}, \boldsymbol{y})$

$$\nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y}) = \nabla^2 f(\boldsymbol{x}) - \sum_{i=1}^{m} y_i \nabla^2 h_i(\boldsymbol{x}). \tag{6.7}$$

Because $\nabla F(\boldsymbol{x}, \boldsymbol{y})$ is symmetric we know that $J(\boldsymbol{x}, \boldsymbol{y}) = \nabla F(\boldsymbol{x}, \boldsymbol{y})^T = \nabla F(\boldsymbol{x}, \boldsymbol{y})$, and therefore Newton's method (6.5) gives

$$\begin{pmatrix} \nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y}) & -\nabla h(\boldsymbol{x}) \\ -\nabla h(\boldsymbol{x})^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \Delta \boldsymbol{x} \\ \Delta \boldsymbol{y} \end{pmatrix} = -\begin{pmatrix} \nabla f(\boldsymbol{x}) - \nabla h(\boldsymbol{x}) \boldsymbol{y} \\ -h(\boldsymbol{x}) \end{pmatrix}. \tag{6.8}$$

This system is the KKT system of the following QP

$$\min_{\Delta \boldsymbol{x} \in \mathbb{R}^n} \quad \frac{1}{2} \Delta \boldsymbol{x}^T (\nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y})) \Delta \boldsymbol{x} + (\nabla_x L(\boldsymbol{x}, \boldsymbol{y}))^T \Delta \boldsymbol{x} \tag{6.9a}$$

$$\text{s.t.} \quad \nabla h(\boldsymbol{x})^T \Delta \boldsymbol{x} = -h(\boldsymbol{x}). \tag{6.9b}$$

This is clearly a QP and the optimum $(\Delta \boldsymbol{x}^T, \Delta \boldsymbol{y}^T)$ from (6.8) is found by using a QP-solver, e.g. the one implemented in this thesis, see appendix B.

The system (6.8) can be expressed in a simpler form, by replacing $\Delta \boldsymbol{y}$ with $\boldsymbol{\mu} - \boldsymbol{y}$

$$\begin{pmatrix} \nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y}) & -\nabla h(\boldsymbol{x}) \\ -\nabla h(\boldsymbol{x})^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \Delta \boldsymbol{x} \\ \boldsymbol{\mu} - \boldsymbol{y} \end{pmatrix} = -\begin{pmatrix} \nabla f(\boldsymbol{x}) - \nabla h(\boldsymbol{x}) \boldsymbol{y} \\ -h(\boldsymbol{x}) \end{pmatrix} \tag{6.10}$$

which is equivalent to

$$\begin{aligned} \begin{pmatrix} \nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y}) & -\nabla h(\boldsymbol{x}) \\ -\nabla h(\boldsymbol{x})^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \Delta \boldsymbol{x} \\ \boldsymbol{\mu} \end{pmatrix} + \begin{pmatrix} \nabla h(\boldsymbol{x}) \boldsymbol{y} \\ \mathbf{0} \end{pmatrix} = \\ -\begin{pmatrix} \nabla f(\boldsymbol{x}) \\ -h(\boldsymbol{x}) \end{pmatrix} + \begin{pmatrix} \nabla h(\boldsymbol{x}) \boldsymbol{y} \\ \mathbf{0} \end{pmatrix}. \end{aligned} \tag{6.11}$$

This means that (6.8) can be reformulated as

$$\begin{pmatrix} \nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y}) & -\nabla h(\boldsymbol{x}) \\ -\nabla h(\boldsymbol{x})^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \Delta \boldsymbol{x} \\ \boldsymbol{\mu} \end{pmatrix} = -\begin{pmatrix} \nabla f(\boldsymbol{x}) \\ -h(\boldsymbol{x}) \end{pmatrix}, \tag{6.12}$$

and the corresponding QP is

$$\min_{\Delta \boldsymbol{x}} \quad \frac{1}{2} \Delta \boldsymbol{x}^T \nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y}) \Delta \boldsymbol{x} + \nabla f(\boldsymbol{x})^T \Delta \boldsymbol{x} \tag{6.13a}$$

$$\text{s.t.} \quad \nabla h(\boldsymbol{x})^T \Delta \boldsymbol{x} = -h(\boldsymbol{x}). \tag{6.13b}$$

As Newton's method approximates numerically, a sequence of Newton iterations is thus necessary to find an acceptable solution. At every iteration the improving direction is found as the solution of the QP (6.13), and therefore the process is called sequential quadratic programming. Whenever $\nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y})$ is positive definite and $\nabla h(\boldsymbol{x})$ has full column rank, the solution to (6.13) can be found using either the range space procedure or the null space procedure. Also, if the program (6.1) is extended to include inequalities

$$\min_{\boldsymbol{x} \in \mathbb{R}^n} \quad f(\boldsymbol{x}) \tag{6.14a}$$

$$\text{s.t.} \quad h(\boldsymbol{x}) \geq \mathbf{0} \tag{6.14b}$$

then the program, that defines the Newton step is an inequality constrained QP of the form

$$\min_{\Delta \boldsymbol{x}} \quad \frac{1}{2} \Delta \boldsymbol{x}^T \nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y}) \Delta \boldsymbol{x} + \nabla f(\boldsymbol{x})^T \Delta \boldsymbol{x} \tag{6.15a}$$

$$\text{s.t.} \quad \nabla h(\boldsymbol{x})^T \Delta \boldsymbol{x} \geq -h(\boldsymbol{x}). \tag{6.15b}$$

When $\nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y})$ is positive definite and $\nabla h(\boldsymbol{x})^T$ has full column rank the solution to this program can be found using either the primal active set method or the dual active set method.

6.2 SQP by example

In this section our SQP implementation will be tested and each Newton step will be illustrated graphically. The nonlinear program that we want to solve is

$$
\begin{aligned}
\min_{\boldsymbol{x}\in\mathbb{R}^n} \quad & f(\boldsymbol{x}) = x_1^4 + x_2^4 \\
\text{s.t.} \quad & x_2 \geq x_1^2 - x_1 + 1 \\
& x_2 \geq x_1^2 - 4x_1 + 6 \\
& x_2 \leq -x_1^2 + 3x_1 + 2.
\end{aligned}
$$

The procedure is to minimize the corresponding Lagrangian function

$$L(\boldsymbol{x}, \boldsymbol{y}) = f(\boldsymbol{x}) - \boldsymbol{y}^T h(\boldsymbol{x}) \tag{6.17}$$

where $\boldsymbol{y}$ are the Lagrangian multipliers and $h(\boldsymbol{x})$ are the function values of the constraints. This is done by using Newton's method to find the solution of

$$
\begin{aligned}
F(\boldsymbol{x}, \boldsymbol{y}) &= \begin{pmatrix} F_1(\boldsymbol{x}, \boldsymbol{y}) \\ F_2(\boldsymbol{x}, \boldsymbol{y}) \end{pmatrix} \\
&= \begin{pmatrix} \nabla_x L(\boldsymbol{x}, \boldsymbol{y}) \\ \nabla_y L(\boldsymbol{x}, \boldsymbol{y}) \end{pmatrix} \\
&= \begin{pmatrix} \nabla f(\boldsymbol{x}) - \nabla h(\boldsymbol{x})\boldsymbol{y} \\ -h(\boldsymbol{x}) \end{pmatrix} = \mathbf{0}.
\end{aligned} \tag{6.18}
$$

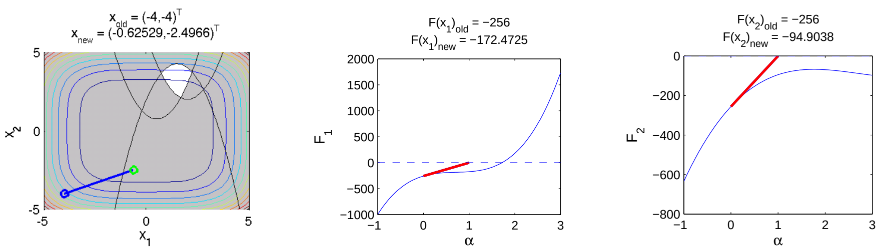
A Newton step is defined by the following QP

$$\begin{pmatrix} \nabla_{xx}^2 L(\boldsymbol{x}, \boldsymbol{y}) & -\nabla h(\boldsymbol{x}) \\ -\nabla h(\boldsymbol{x})^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \Delta \boldsymbol{x} \\ \Delta \boldsymbol{y} \end{pmatrix} = -\begin{pmatrix} \nabla f(\boldsymbol{x}) - \nabla h(\boldsymbol{x})\boldsymbol{y} \\ -h(\boldsymbol{x}) \end{pmatrix}. \tag{6.19}$$

In this example we have calculated the analytical Hessian matrix $\nabla_{xx}^2 L(\boldsymbol{x}, \boldsymbol{y})$ of (6.17) to make the relation (6.19) exact, even though a BFGS update by Powell [16] has been implemented. This is done for illustrational purpose alone, as we want to plot each Newton step in $F_{1_1}(\boldsymbol{x}, \boldsymbol{y})$ and $F_{1_2}(\boldsymbol{x}, \boldsymbol{y})$ from (6.18) in relation to the improving direction. The analytical Hessian matrix can be very expensive to evaluate, and therefore the BFGS approximation is usually preferred. When the improving direction $[\Delta\boldsymbol{x}, \Delta\boldsymbol{y}]^T$ has been found by solving (6.19), the step size α is calculated in a line search function implemented according to the one suggested by Powell [16].

Iteration 1

We start at position $(\boldsymbol{x}_0, \boldsymbol{y}_0) = ([-4, -4]^T, [0, 0, 0]^T)$ with the corresponding Lagrangian function value $L(\boldsymbol{x}_0, \boldsymbol{y}_0) = 512$. The first Newton step leads us to $(\boldsymbol{x}_1, \boldsymbol{y}_1) = ([-0.6253, -2.4966]^T, [0, 32.6621, 0]^T)$ with $L(\boldsymbol{x}_1, \boldsymbol{y}_1) = 410.9783$, and the path is illustrated in figure 6.1(a). In figures 6.1(b) and 6.1(c) F_{1_1} and F_{1_2} are plotted in relation to the step size α, where the red line illustrates the step taken. We have plotted F_{1_1} and F_{1_2} for $\alpha \in [-1, 3]$ even though the line search function returns $\alpha \in [0, 1]$. In figures 6.1(b) and 6.1(c), $\alpha = 0$ is the position before the step is taken and $\alpha \in [0, 1]$ where the red line ends illustrates the position after taking the step. It is clear from the figures, that a full step ($\alpha = 1$) is taken and that F_{1_1} and F_{1_2} increase from -256 to -172.4725, and -256 to -94.9038, respectively.

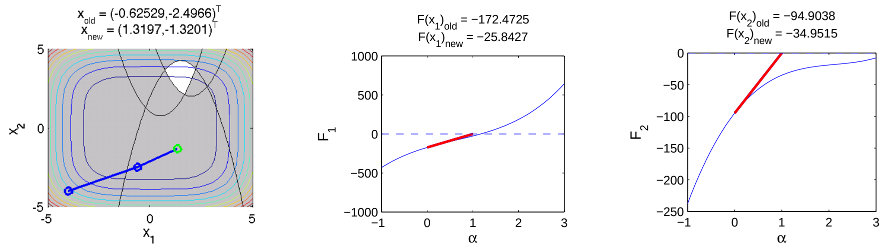


(a) The path. (b) Newton step in F_{1_1}. (c) Newton step in F_{1_2}.

Figure 6.1: The Newton step at iteration 1. $L(\boldsymbol{x}_0, \boldsymbol{y}_0) = 512$ and $L(\boldsymbol{x}_1, \boldsymbol{y}_1) = 410.9783$.

Iteration 2

Having taken the second step the position is $(\boldsymbol{x}_2, \boldsymbol{y}_2) = ([1.3197, -1.3201]^T, [0, 25.7498, 0]^T)$ as seen in figure 6.2(a). The Lagrangian function value is $L(\boldsymbol{x}_2, \boldsymbol{y}_2) = 103.4791$. The step size is $\alpha = 1$, F_{1_1} increases from -172.4725 to -25.8427 and F_{1_2} increases from -94.9038 to -34.9515 as seen in figures 6.2(b) and 6.2(c).

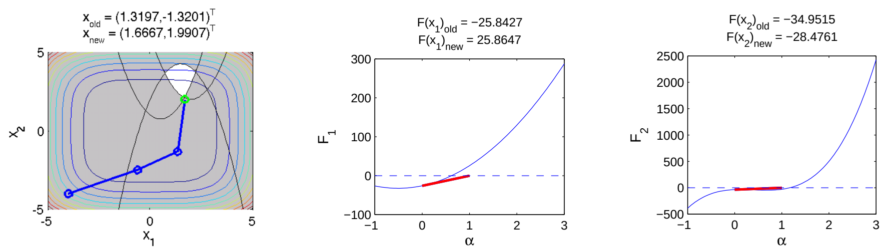


(a) The path. (b) Newton step in F_{1_1}. (c) Newton step in F_{1_2}.

Figure 6.2: The Newton step at iteration 2. $L(\boldsymbol{x}_1, \boldsymbol{y}_1) = 410.9783$ and $L(\boldsymbol{x}_2, \boldsymbol{y}_2) = 103.4791$.

Iteration 3

After the third step, the position is $(\boldsymbol{x}_3, \boldsymbol{y}_3) = ([1.6667, 1.9907]^T, [15.7893, 44.2432, 0]^T)$, see figure 6.3(a). The Lagrangian function value is $L(\boldsymbol{x}_3, \boldsymbol{y}_3) = 30.6487$. Again the step size is $\alpha = 1$, F_{1_1} increases from -25.8427 to 25.8647 and F_{1_2} increases from -34.9515 to -28.4761 as seen in figure 6.3(b) and 6.3(c).

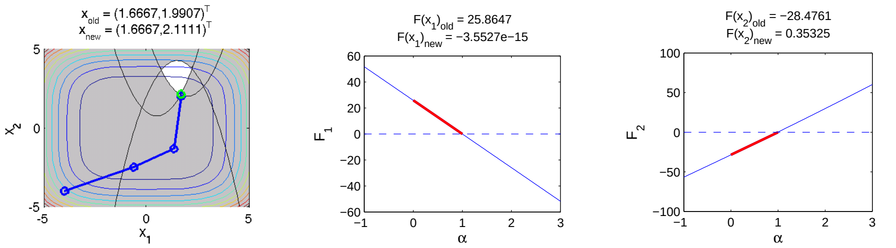


(a) The path. (b) Newton step in F_{1_1}. (c) Newton step in F_{1_2}.

Figure 6.3: The Newton step at iteration 3. $L(\boldsymbol{x}_2, \boldsymbol{y}_2) = 103.4791$ and $L(\boldsymbol{x}_3, \boldsymbol{y}_3) = 30.6487$.

Iteration 4

The fourth step takes us to $(\boldsymbol{x}_4, \boldsymbol{y}_4) = ([1.6667, 2.1111]^T, [2.1120, 35.1698, 0]^T)$, see figure 6.4(a). The Lagrangian function value is $L(\boldsymbol{x}_4, \boldsymbol{y}_4) = 27.5790$. The step size is $\alpha = 1$, F_{1_1} decreases from 25.8647 to $-3.55271e{-}15$ and F_{1_2} increases from -28.4761 to 0.3533 as seen in figures 6.4(b) and 6.4(c). Even though refinements can be made by taking more steps we stop the algorithm at the optimal position $(\boldsymbol{x}^*, \boldsymbol{y}^*) = (\boldsymbol{x}_4, \boldsymbol{y}_4) = ([1.6667, 2.1111]^T, [2.1120, 35.1698, 0]^T)$ where the optimal value is $f(\boldsymbol{x}^*) = 27.5790$.



(a) The path. (b) Newton step in F_{1_1}. (c) Newton step in F_{1_2}.

Figure 6.4: The Newton step at iteration 4. $L(\boldsymbol{x}_3, \boldsymbol{y}_3) = 30.6487$ and $L(\boldsymbol{x}_4, \boldsymbol{y}_4) = 27.5790$.

An interactive demo application `SQP_demo.m` is found in appendix D.5.

Chapter 7

Conclusion

In this thesis we have investigated the active set methods, together with the range and null space procedures which are used in solving QP's. We have also focused on refining the methods and procedures in order to gain efficiency and reliability. Now we will summarize the most important observations found in the thesis.

The primal active set method is the most intuitive method. However, it has two major disadvantages. Firstly, it requires a feasible starting point, which is not trivial to find. Secondly and most crucially, is the possibility of cycling. The dual active set method does not suffer from these drawbacks. The method easily computes the starting point itself, and furthermore convergence is guaranteed. On the other hand, the primal active set method has the advantage of only requiring the Hessian matrix to be positive semi definite.

The range space and the null space procedures are equally good. But, where the range space procedure is fast, the null space procedure is slow and vice versa. Thus, in practice the choice of method is problem specific. For problems consisting of a small number of active constraints in relation to the number of variables, the range space procedure is preferable. And for problems with a large number active constraints compared to the number of variables, the null space procedure is to be preferred. If the nature of the problem potentially allows a large number of constraints in comparison to the number of variables, then, to gain advantage of both procedures, it is necessary to shift dynamically between them. This can easily be done by comparing the number of active constraints against the number of variables e.g. for each iteration However, this requires a theoretically predefined relation pointing at when to shift between the range space and the null space procedures. This relation can as mentioned be found in theory, but in practice it also relies on the way standard Matlab functions are implemented, the architecture of the processing unit, memory access etc., and therefore finding this relation in practice is more complicated than first assumed.

By using Givens rotations, the factorizations used to solve the KKT system can be updated instead of completely recomputed. And as the active set methods solve a sequence of KKT systems, the total computational savings are significant. The null space procedure in particular has become more efficient. These updates have been further refined by distinguishing bounds, i.e. fixed variables, from general constraints. The greater fraction of active bounds compared to active general constraints, the smaller the KKT system gets and vice versa. Therefore, this particular update is of the utmost importance, when the QP contains potentially many active bounds.

The SQP method is useful in solving nonlinear constrained programs. It is founded in Newton steps. The SQP solver is based on a sequence of Newton steps, where each single step is solved as a QP. So a fast and reliable QP solver is essential in the SQP method. The QP solver which has been developed in the thesis, see appendix B, has proved successful in fulfilling this task.

7.1 Future Work

Dynamic Shift Implementation of dynamic shift between the range space and null space procedures would be interesting, because computational speed could be gained this way.

Low Level Language Our QP solver has been implemented in Matlab, and standard Matlab functions such as `chol` and `qr` have been used. In future works, implementation in Fortran or C++ would be preferable. This would make the performance tests of the different methods more reliable. Implementation in any low level programming language may be expected to improve general performance significantly. Furthermore, any theoretically computed performances may also be expected to hold in practice.

Precomputed Active Set The dual active set method requires the Hessian matrix $\boldsymbol{G}$ of the objective function to be positive definite, as it computes the starting point $\boldsymbol{x}_0$ by use of the inverse Hessian matrix: $\boldsymbol{x}_0 = -\boldsymbol{G}^{-1}\boldsymbol{g}$. The primal active set method using the null space procedure only requires the reduced Hessian matrix $\boldsymbol{Z}^T\boldsymbol{G}\boldsymbol{Z}$ to be positive definite. In many problems it is possible to find an active set which makes the reduced Hessian matrix positive definite even if the Hessian matrix is positive semi definite. In future works the LP solver which finds the starting point to the primal active set method should be designed so that it also finds the active set which makes the reduced Hessian matrix positive definite. This extension would give the primal active set method an advantage compared to the dual active set method.

Bibliography

[1] Li, W. and Swetits, J. J.
The Linear l1 Estimator and the Huber M-Estimator, SIAM Journal on Optimization, (1998).

[2] Gill, P E., Gould, N. I. M., Murray, W., Saunders, M. A., Wright, M. H.
A Weighted Gram-Schmidt Method for Convex Quadratic Pro- gramming. Mathematical Programming, 30, (1984).

[3] Gill, P. E. and Murray, W.
Numerically Stable Methods for Quadratic Programming. Mathematical Programming, 14, (1978).

[4] Golub, G. H. and Van Loan, C. F.
Matrix Computations, (1996).

[5] Wilkinson, J. H.
The Algebraic Eigenvalue Problem, (1965).

[6] Dennis, J. E. and Schnabel, R. B.
Numerical Methods for Unconstrained Optimization and Nonlinear Equations., (1996).

[7] Gill, P. E., Golub, G. H., Murray, W. and Saunders, M. A.
Methods for Modifying Matrix Factorizations. Mathematics of Computation, 28., (1974).

[8] Gill, P. E. and Murray, W. *Numerically Stable Methods for Quadratic Programming. Mathematical Programming, 14.*,(1978).

[9] Gill, P. E., Murray, W., Saunders, M. E. and Wright, M. H. *Procedures for Optimization Problems with a Mixture of Bounds and General Linear Constraints. ACM Transactions on Mathematical Software, 10.*,(1984).

[10] Goldfarb, D. and Idnani, A.
A numerically stable dual method for solving strictly convex quadratic programs, (1983).

[11] Schmid, C. and Biegler, L.
Quadratic programming methods for reduced hessian SQP, (1994).

[12] Schittkowski, K.
QL: A Fortran Code for Convex Quadratic Programming - Users Guide. Technical report, Department of Mathematics, University of Bayreuth, (2003).

[13] John Bagterp Jørgensen.
Quadratic Programming, (2005).

[14] Nocedal, J. and Wright, S. J.
Numerical Optimization, Springer Series in Operations Research, Second Edition, (2006).

[15] John Bagterp Jørgensen.
Lecture notes from course 02611 Optimization Algorithms and Data-Fitting, IMM, DTU, DK-2800 Lyngby, (november 2005).

[16] Powell, M. J. D.
A Fast Algorithm for Nonlinearly Constrained Optimization Calculations. In G. A. Watson, editor, Numerical Analysis, (1977).

[17] John Bagterp Jørgensen.
Lecture notes from course: Model Predictive Control, IMM, DTU, DK-2800 Lyngby, (february 2007).

[18] L. Eldén, L. Wittmeyer-Koch and H.B. Nielsen:
Introduction to Numerical Computation, published by Studentlitteratur(2002).

Appendix A

Quadruple Tank Process

The quadruple tank process Jørgensen [17] is a system of four tanks, which are connected through pipes as illustrated in figure A.1. Water from a main-tank is transported around the system and the flow is controlled by the pumps F_1 and F_2. The optimization problem is to stabilize the water level in tank 1 and 2 at some level, called set points illustrated as a red line. Values γ_1 and γ_2 of the two valves control how much water is pumped directly into tank 1 and 2 respectively. The valves are constant, and essential for the ease with which the process is controlled.

The dynamics of the quadruple tank process are described in the following differential equations

$$\frac{dh_1}{dt} = \frac{\gamma_1}{A_1}F_1 + \frac{a_3}{A_1}\sqrt{2gh_3} - \frac{a_1}{A_1}\sqrt{2gh_1} \tag{A.1a}$$

$$\frac{dh_2}{dt} = \frac{\gamma_2}{A_2}F_2 + \frac{a_4}{A_2}\sqrt{2gh_4} - \frac{a_2}{A_2}\sqrt{2gh_2} \tag{A.1b}$$

$$\frac{dh_3}{dt} = \frac{1-\gamma_2}{A_3}F_2 - \frac{a_3}{A_3}\sqrt{2gh_3} \tag{A.1c}$$

$$\frac{dh_4}{dt} = \frac{1-\gamma_1}{A_4}F_1 - \frac{a_4}{A_4}\sqrt{2gh_4} \tag{A.1d}$$

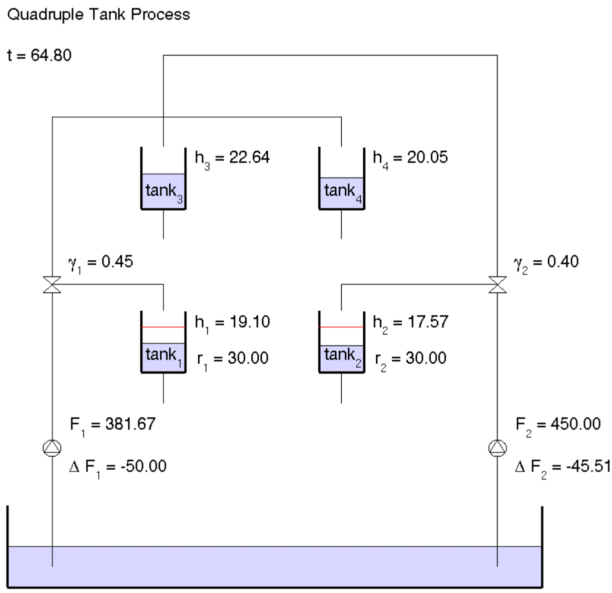


Figure A.1: Quadruple Tank Process.

where A_i is the cross sectional area, a_i is the area of outlet pipe, h_i is the water level of tank no. i, γ_1 and γ_2 are flow distribution constants of the two valves, g is acceleration of gravity and F_1 and F_2 are the two rate of flows. As the QP solver requires the constraints to be linear we need to linearize the equations in (A.1). Of course this linearization causes the model to be a much more coarse approximation, but as the purpose is to build a convex QP for testing, this is of no importance. The linearizations are

$$\frac{dh_1}{dt} = \frac{\gamma_1}{A_1}F_1 + \frac{a_3}{A_1}2gh_3 - \frac{a_1}{A_1}2gh_1 \tag{A.2a}$$

$$\frac{dh_2}{dt} = \frac{\gamma_2}{A_2}F_2 + \frac{a_4}{A_2}2gh_4 - \frac{a_2}{A_2}2gh_2 \tag{A.2b}$$

$$\frac{dh_3}{dt} = \frac{1-\gamma_2}{A_3}F_2 - \frac{a_3}{A_3}2gh_3 \tag{A.2c}$$

$$\frac{dh_4}{dt} = \frac{1-\gamma_1}{A_4}F_1 - \frac{a_4}{A_4}2gh_4. \tag{A.2d}$$

This system of equations is defined as the function

$$\frac{d}{dt}\boldsymbol{x}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t)), \quad \boldsymbol{x} = [h_1\ h_2\ h_3\ h_4]^T, \quad \boldsymbol{u} = [F_1\ F_2]^T \tag{A.3}$$

which is discretized using Euler

$$\frac{d}{dt}\boldsymbol{x}(t) \simeq \frac{\boldsymbol{x}(t_{k+1}) - \boldsymbol{x}(t_k)}{t_{k+1} - t_k} = \frac{\boldsymbol{x}_{k+1} - \boldsymbol{x}_k}{\Delta t} = f(\boldsymbol{x}_k, \boldsymbol{u}_k) \tag{A.4a}$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta t f(\boldsymbol{x}_k, \boldsymbol{u}_k) \tag{A.4b}$$

$$F(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{x}_{k+1}) = \boldsymbol{x}_k + \Delta t f(\boldsymbol{x}_k, \boldsymbol{u}_k) - \boldsymbol{x}_{k+1} = \boldsymbol{0}. \tag{A.4c}$$

The function $F(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{x}_{k+1}) = \boldsymbol{0}$ defines four equality constraints, one for each height in $\boldsymbol{x}$. As the time period is discretized into N time steps, this gives $4N$ equality constraints. Because each equality constraint is defined as two inequality constraints of identical value, as lower and upper bound, (A.4) defines $8N$ inequality constraints, called general constraints.

To make the simulation realistic we define bounds on each variable

$$\boldsymbol{u}_{min} \leq \boldsymbol{u}_k \leq \boldsymbol{u}_{max} \tag{A.5a}$$

$$\boldsymbol{x}_{min} \leq \boldsymbol{x}_k \leq \boldsymbol{x}_{max} \tag{A.5b}$$

which gives $2N(|\boldsymbol{u}| + |\boldsymbol{x}|) = 12N$ inequality constraints, called bounds. We have also defined restrictions on how much the rate of flows can change between two time steps

$$\Delta \boldsymbol{u}_{min} \leq \boldsymbol{u}_k - \boldsymbol{u}_{k-1} \leq \Delta \boldsymbol{u}_{max} \tag{A.6}$$

in addition this gives $2N|\boldsymbol{u}| = 4N$ inequality constraints, also general constraints.

The objective function which we want to minimize is

$$min \quad \frac{1}{2}\int((h_1(t) - r_1)^2 + (h_2(t) - r_2)^2)dt \tag{A.7}$$

where r_1 and r_2 are the set points. The exact details of how the system is set up as a QP can be found in either Jørgensen [17] or our MATLAB implementation `quad_tank_demo.m`. The quadruple tank process defines an inequality constrained convex QP of $(|\boldsymbol{u}|+|\boldsymbol{x}|)N = 6N$ variables and $(8+12+4)N = 24N$ inequality constraints consisting of $12N$ bounds and $12N$ general constraints.

Quadruple Tank Process by example
Now we will set up a test example of the quadruple tank problem. For this we use the following settings

$$\begin{aligned}
t &= [0,\ 360] \\
N &= 100 \\
\boldsymbol{u}_{min} &= [0,\ 0]^T \\
\boldsymbol{u}_{max} &= [500,\ 500]^T \\
\Delta\boldsymbol{u}_{min} &= [-50,\ -50]^T \\
\Delta\boldsymbol{u}_{max} &= [50,\ 50]^T \\
\boldsymbol{u}_0 &= [0,\ 0]^T \\
\gamma_1 &= 0.45 \\
\gamma_2 &= 0.40 \\
r_1 &= 30 \\
r_2 &= 30 \\
\boldsymbol{x}_{min} &= [0,\ 0,\ 0,\ 0]^T \\
\boldsymbol{x}_{max} &= [40,\ 40,\ 40,\ 40]^T \\
\boldsymbol{x}_0 &= [0,\ 0,\ 0,\ 0]^T.
\end{aligned}$$

This defines an inequality constrained convex QP with $6 * 100 = 600$ variables and $24 * 100 = 2400$ constraints. The solution to the problem is found by using our QP solver, see appendix B. The solution is illustrated in figure A.2, where everything is seen to be as expected. We have also written a program `quad_tank_plot.m` for visualizing the solution of the quadruple tank problem as an animation. In figure A.3 to A.8 we have illustrated the solution $\boldsymbol{x}_k^*$ and $\boldsymbol{u}_k^*$ for $k \in \{1, 3, 6, 10, 15, 20, 25, 30, 40, 60, 80, 100\}$ using `quad_tank_plot.m`.

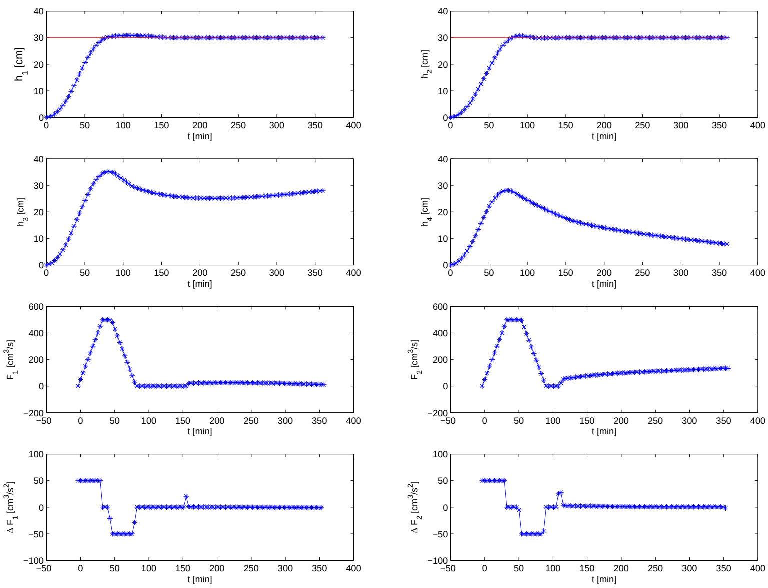
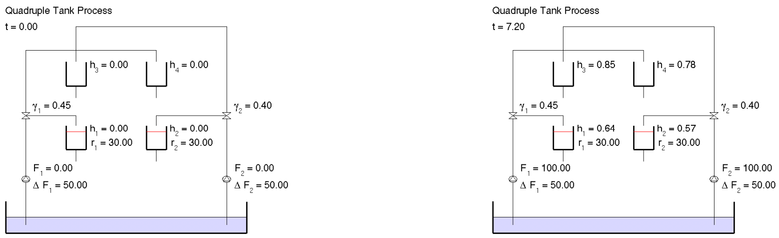


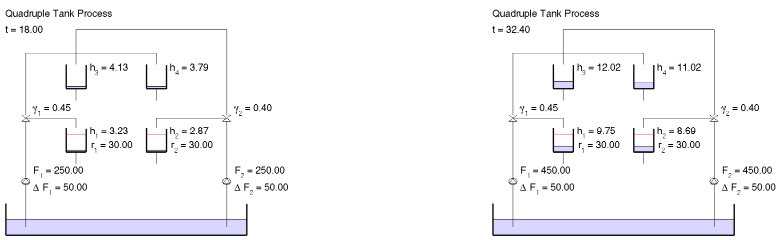
Figure A.2: The solution of the quadruple tank problem found by using our QP solver. It is seen that the water levels in tank 1 and 2 are stabilized around the setpoints. The water levels in tank 3 and 4, the two flows F_1 and F_2 and the difference in flow between time steps ΔF_1 and ΔF_2 are also plotted.

An interactive demo application `quad_tank_demo.m` is found in appendix D.5.



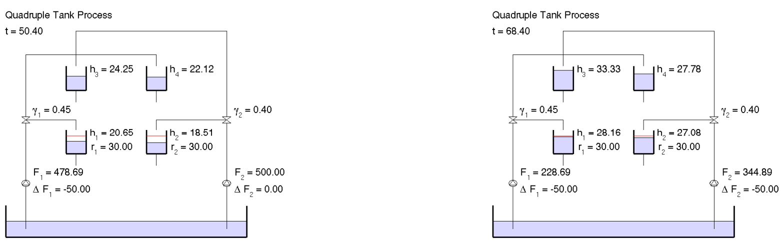
(a) $k = 1$ (b) $k = 3$

Figure A.3: discretization $k = 1$ and $k = 3$.



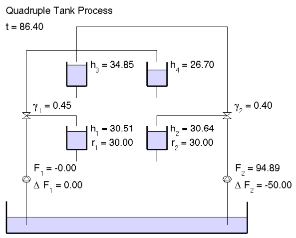
(a) $k = 6$ (b) $k = 10$

Figure A.4: discretization $k = 6$ and $k = 10$.



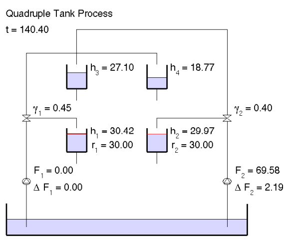
(a) $k = 15$ (b) $k = 20$

Figure A.5: discretization $k = 15$ and $k = 20$.



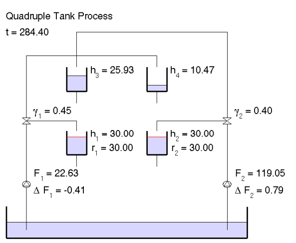
(a) $k = 25$ (b) $k = 30$

Figure A.6: discretization $k = 25$ and $k = 30$.



(a) $k = 40$ (b) $k = 60$

Figure A.7: discretization $k = 40$ and $k = 60$.



(a) $k = 80$ (b) $k = 100$

Figure A.8: discretization $k = 60$ and $k = 100$.

Appendix B

QP Solver Interface

Our QP solver is implemented in Matlab as `QP_solver.m`, and it is founded on the following structure

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

$$\text{s.t.} \quad l_i \leq x_i \leq u_i \qquad i = 1, 2, ..., n \tag{B.1b}$$

$$(b_l)_i \leq \boldsymbol{a}_i^T\boldsymbol{x} \leq (b_u)_i \qquad i = 1, 2, ..., m \tag{B.1c}$$

where f is the objective function. The number of bounds is $2n$ and the number of general constraints is $2m$. This means, that we have upper and lower limits on every bound and on every general constraint. The Matlab interface of the QP solver is constructed as follows

```
x = QP_solver(G, g, l, u, A, bl, bu, x).
```

The input parameters of the QP solver are described in table B.1.

`G` The Hessian matrix $\boldsymbol{G} \in \mathbb{R}^{n \times n}$ of the objective function.

`g` The linear term $\boldsymbol{g} \in \mathbb{R}^n$ of the objective function.

`l` The lower limits $\boldsymbol{l} \in \mathbb{R}^n$ of bounds.

`u` The upper limits $\boldsymbol{u} \in \mathbb{R}^n$ of bounds.

`A` The constraint matrix $\boldsymbol{A} = [\boldsymbol{a}_i^T]_{i=1,2,\ldots,m}$, so $\boldsymbol{A} \in \mathbb{R}^{m \times n}$.

`bl` The lower limits $\boldsymbol{b_l} \in \mathbb{R}^m$ of general constraints.

`bu` The upper limits $\boldsymbol{b_u} \in \mathbb{R}^m$ of general constraints.

`x` A feasible starting point $\boldsymbol{x} \in \mathbb{R}^n$ used in the primal active set method. If $\boldsymbol{x}$ is not given or empty, then the dual active set method is called within the QP solver.

Table B.1: The input parameters of the QP and the LP solver.

It is possible to define equalities, by means the lower and upper limits are equal, as $l_i = u_i$ and $(b_l)_i = (b_u)_i$ respectively. If any of the limits are unbounded, they must be defined as $-\infty$ for lower limits and ∞ for upper limits. If the QP solver is called with a starting point $\boldsymbol{x}$, then the primal active set method is called within the QP solver. The feasibility of the starting point is checked by the QP solver before the primal active set method is called.

It is possible to find a feasible starting point with our LP solver, which we implemented in Matlab as `LP_solver.m`. The LP solver is based on (B.1) and the Matlab function `linprog`. The Matlab interface of the LP solver is constructed as follows

```
x = LP_solver(l, u, A, bl, bu).
```

The input parameters of the LP solver are described in table B.1.

It must be mentioned, that both the QP and LP solver has some additional input and output parameters. These parameters are e.g. used for tuning and performance analysis of the solvers. For a further description of these parameters we refer to the respective Matlab help files.

Appendix C

Implementation

The algorithms discussed in the thesis have been implemented in Matlab version `7.3.0.298 (R2006b), August 03, 2006`. In the following, we have listed the implemented functions in sections.

C.1 Equality Constrained QP's

The null space procedure solves an equality constrained QP by using the null space of the constraint matrix, and is implemented in

`null_space.m` .

The range space procedure solves the same problem by using the range space of the constraint matrix, and is implemented in

`range_space.m` .

C.2 Inequality Constrained QP's

An inequality constrained convex QP can be solved by use of the primal active set method which is implemented in

`primal_active_set_method.m`

or the dual active set method

`dual_active_set_method.m` .

The active set methods have been integrated in a QP solver that sets up the QP with a set of bounds and a set of general constraints. An interface has been provided that offers different options to the user.

`QP_solver.m` .

If the user want to use the primal active set method in the QP solver, a feasible starting point must be calculated. This can be done by using the LP solver

`LP_solver.m` .

C.3 Nonlinear Programming

SQP solves a nonlinear program by solving a sequence of inequality constrained QP's. The SQP solver is implemented in

`SQP_solver.m` .

C.4 Updating the Matrix Factorizations

By updating the matrix factorizations, the efficiency of the null space procedure and the range space procedure can be increased significantly. The following implementations are used for updating the matrix factorizations used in the null space procedure and the range space procedure. All the updates are based on Givens rotations which are computed in

`givens_rotation_matrix.m` .

Update of matrix factorizations used in range space procedure are implemented in the following files

`range_space_update.m`

`qr_fact_update_app_col.m`

`qr_fact_update_rem_col.m` .

And for the null space procedure the matrix updates are implemented in

`null_space_update.m`

`null_space_update_fact_app_col.m`

`null_space_update_fact_rem_col.m` .

Further optimization of the matrix factorization is done by using updates based on fixed and free variables. These updates have only been implemented for the null space procedure and are found in

`null_space_updateFRFX.m`

`null_space_update_fact_app_general_FRFX.m`

`null_space_update_fact_rem_general_FRFX.m`

`null_space_update_fact_app_bound_FRFX.m`

`null_space_update_fact_rem_bound_FRFX.m`

C.5 Demos

For demonstrating the methods and procedures, we have implemented different demonstration functions. The QP solver is demonstrated in

`QP_demo.m`

which uses the plot function

`active_set_plot.m` .

Among other options the user can choose between the primal active set method and the dual active set method.

The QP solver is also demonstrated on the quadruple tank process in

`quad_tank_demo.m`

which uses the plot functions

`quad_tank_animate.m`

`quad_tank_plot.m` .

Besides having the possibility of adjusting the valves, pumps and the set points individually, the user can vary the size of the QP by the input N.

The SQP solver is demonstrated on a small two dimensional nonlinear program and the path is visualized at each iteration. The implementation is found in

`SQP_demo.m` .

C.6 Auxiliary Functions

`add2mat.m` .

`line_search_algorithm.m` .

Appendix D

Matlab-code

D.1 Equality Constrained QP's

null_space.m

```
function [x,u] = null_space(G,A,g,b)

% NULL_SPACE solves the equality constrained convex QP:
%                    min  1/2x'Gx+g'x         (G is required to be postive semi definite)
%                    s.t.    A'x = b          (A is required to have full column rank)
% where the number of variables is n and the number of constraints is m.
% The null space of the QP is used to find the solution.
%
%   Call
%       [x,u] = null_space(G,A,g,b)
%
%   Input parameters
%       G                 : is the Hessian matrix (nxn) of the QP.
%       A                 : is the constraint matrix (nxm): every column contains a from the
%                                    equality: a'x = b.
%       g                 : is the gradient (nx1) of the QP.
%       b                 : is the right hand side of the constraints.
%
%   Output parameters
%       x                 : the solution
%       mu                : the lagrangian multipliers
%
%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date        : 08. february 2007.

[n,m] = size(A);
if( m~=0 ) % for situations where A is empty
```

```
    [Q,R] = qr(A);
    Q1 = Q(:,1:m);
    Q2 = Q(:,m+1:n);
    R = R(1:m,:);
    py = R'\b;
    Q2t = Q2';
    gz = Q2t*(G*(Q1*py) + g);
    Gz = Q2t*G*Q2;
    L = chol(Gz)';
    pz = L\-gz;
    pz = L'\pz;
    x = Q1*py + Q2*pz;
    u = R\(Q1'*(G*x + g));
else
    x = -G\g;
    u = [];
end
```

range_space.m

```
function [x mu] = range_space(L,A,g,b)
% RANGE_SPACE solves the equality constrained convex QP:
%                   min  1/2x'Gx+g'x          (G is required to be postive definite)
%                   s.t.    A'x = b           (A is required to have full column rank)
% where the number of variables is n and the number of constraints is m.
% The range space of the QP is used to find the solution.
%
%   Call
%       [x,u] = range_space(L,A,g,b)
%   Input parameters
%       L               : is the cholesky factorization of the Hessian matrix (nxn) of the QP.
%       A               : is the constraint matrix (nxm): every column contains a from the
%                                equality: a'x = b.
%       g               : is the gradient (nx1) of the QP.
%       b               : is the right hand side of the constraints.
%   Output parameters
%       x               : the solution
%       mu              : the lagrangian multipliers

%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date        : 08. february 2007.
%   Reference   : ---------------------

Lt = L';
K = Lt\A;
H = K'*K;
w = Lt\g;
z = b+K'*w;
M = chol(H);
mu = M'\z;
mu = M\mu;
y = K*mu-w;
x = L\y;
```

D.2 Inequality Constrained QP's

primal_active_set_method.m

```
function [x,mu,info,perf] = primal_active_set_method(G,g,A,b,x,w_non,pbc,opts,
    trace)

% PRIMAL_ACTIVE_SET_METHOD Solving an inequality constrained QP of the
% form:
%   min  f(x) = 0.5*x'*G*x + g*x
%   s.t. A*x >= b,
% by solving a sequence of equality constrained QP's using the primal
% active set method. The method uses the range space procedure or the null
% space procedure to solve the KKT system. Both the range space and the
% null space procedures has been provided with factorization updates.
%
%   Call
%     x = primal_active_set_method(G, g, A, b, w_non, pbc)
%     x = primal_active_set_method(G, g, A, b, w_non, pbc, opts)
%     [x, mu, info, perf] = primal_active_set_method( ... )
%
% Input parameters
%   G     : The Hessian matrix of the objective function, size nxn.
%   g     : The linear term of the objective function, size nx1.
%   A     : The constraint matrix holding the constraints, size nxm.
%   b     : The right-hand side of the constraints, size mx1.
%   x     : Starting point, size nx1.
%   w_non : List of inactive constraints, pointing on constraints in A.
%   pbc   : List of corresponding constraints, pointing on constraints in
%           A. Can be empty.
%   opts  : Vector with 3 elements:
%           opts(1) = Tolerance used to stabilize the methods numerically.
%                     If |value| <= opts(1), then value is regarded as zero.
%           opts(2) = maximum no. of iteration steps.
%           opts(3) = 1 : Using null space procedure.
%                     2 : Using null space procedure with factorization
%                         update.
%                     3 : Using null space procedure with factorization
%                         update based on fixed and free variables. Can only
%                         be called, if the inequality constrained QP is
%                         setup on the form seen in QP_solver.
%     If opts is not given or empty, the default opts = [1e-8 1000 3].
%
% Output parameters
%   x    : The optimal solution.
%   mu   : The Lagrange multipliers at the optimal solution.
%   info : Performace information, vector with 3 elements:
%          info(1)   = final values of the objective function.
%          info(2)   = no. of iteration steps.
%          info(3)   = 1 : Feasible solution found.
%                      2 : No. of iteration steps exceeded.
%   perf : Performace, struct holding:
%          perf.x  : Values of x, size is nx(it+1).
%          perf.f  : Values of the objective function, size is 1x(it+1).
%          perf.mu : Values of mu, size is nx(it+1).
%          perf.c  : Values of c(x), size is mx(it+1).
%          perf.Wa : Active set, size is mx(it+1).
%          perf.Wi : Inactive set, size is mx(it+1).
%
%   By          : Carsten V\"olcker, s961572.
%                 Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization.
%                 M.Sc., IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor.
%                 Per Grove Thomsen, Professor.
%   Date        : 07. June 2007.

% the size of the constraint matrix, where the constraints are given columnwise
    ...
[n,m] = size(A);

nb = 2*n; % number of bounds
ngc = m - nb; % number of general constraints

% initialize...
z = zeros(m,1);
x0 = x;
f0 = objective(G,g,x0);
mu0 = z;
c0 = constraints(A,b,x0);
w_act0 = z;
w_non0 = (1:1:m)';
```

```


% initialize options...
tol = opts(1);
it_max = opts(2);
method = opts(3);
% initialize containers...
%trace = (nargout > 3);
perf = [];
if trace
    X = repmat(zeros(n,1),1,it_max);
    F = repmat(0,1,it_max);
    Mu = repmat(z,1,it_max);
    C = repmat(z,1,it_max);
    W_act = repmat(z,1,it_max);
    W_non = repmat(z,1,it_max);
end

% initialize counters...
it = 0;
Q = []; T = []; L = []; rem = [];        % both for null_space_update and for null_space_update_FXFR

if method == 3   % null space with FXFR-update
    nb1 = n*2;    % number of bounds
else
    nb1 = 0;
end

nab = 0;          % number of active bouns
P = eye(n);

w_act = [];
Q_old = []; T_old = []; L_old = []; rem = [];

% iterate...
stop = 0;
while ~stop
    it = it + 1;
    if it >= it_max
        stop = 2; % maximum no iterations exceeded
    end

    % call range/null space procedure...
    mu = z;

    if method == 1
        [p,mu_] = null_space(G,A(:,w_act),G*x+g,zeros(length(w_act),1));
    end

    if method == 2
        [p,mu_,Q,T,L] = null_space_update(G,A(:,w_act),G*x+g,zeros(length(w_act),1),Q,T,L,rem);
    end

    if method == 3
        Cr = G*x+g;
        A_ = A(:,w_act(nab+1:end));
        %         ajust C(:,r) to make it correspond to the factorizations of the
        %         Fixed variables (whenever -1 appears at variable i C(:,r)i should change sign)
        if nab % some bounds are in the active set
            u_idx = find(w_act > nb1/2 & w_act< nb1+1);
            var = n-nab+u_idx;
            Cr(var) = -Cr(var);
            A_(var,:) = -A_(var,:);
        end
        [p,mu_,Q,T,L] = null_space_updateFRFX(Q,T,L,G,A_,Cr,zeros(length(w_act),1),nab,rem-nab);
    end

    mu(w_act) = mu_;

    if norm(p) < tol
        if mu > -tol
            stop = 1; % solution found
        else
            % compute index j of bound/constraint to be removed...
            [dummy,rem] = min(mu_);
            [w_act w_non A P x nab G g] = remove_constraint(rem,A,w_act,w_non,x,P,nb1,nab,n,G,g,pbc,b);
        end
    else
        % compute step length and index j of bound/constraint to be appended...
        [alpha,j] = step_length(A,b,x,p,w_non,nb,n,tol);
        if alpha < 1
```

```
            % make constrained step...
            x = x + alpha*p;
            [w_act w_non A P x nab G g Q] = append_constraint(j,A,w_act,w_non,x,P,nb1,nab,n,G,g,Q,pbc,b); % r is index of A
        else
            % make full step...
            x = x + p;
        end
    end
    % collecting output in containers...
    if trace
        if nb1 % method 3 is used
            X(:,it) = P'*x;
        else
            X(:,it) = x;
        end
        F(it) = objective(G,g,x);
        Mu(:,it) = mu;
        C(:,it) = constraints(A,b,x);
        W_act(w_act,it) = w_act;
        W_non(w_non,it) = w_non;
    end
end

if nb1 % method 3 is used
    x = P'*x;
end

% building info...
info = [objective(G,g,x) it stop];

% building perf...
if trace
    X = X(:,1:it); X = [x0 X];
    F = F(1:it); F = [f0 F];
    Mu = Mu(:,1:it); Mu = [mu0 Mu];
    C = C(:,1:it); C = [c0 C];
    W_act = W_act(:,1:it); W_act = [w_act0 W_act];
    W_non = W_non(:,1:it); W_non = [w_non0 W_non];
    perf = struct('x',{X},'f',{F},'mu',{Mu},'c',{C},'Wa',{W_act},'Wi',{W_non});
end

function [alpha,j] = step_length(A,b,x,p,w_non,nb,n,tol)
alpha = 1; j = [];
for app = w_non
    if app > nb
        fv = 1:1:n; % general constraint
    else
        fv = mod(app-1,n)+1; % index of fixed variabel
    end
    ap = A(fv,app)'*p(fv);
    if ap < -tol
        temp = (b(app) - A(fv,app)'*x(fv))/ap;
        if -tol < temp & temp < alpha
            alpha = temp; % smallest step length
            j = app; % index j of bound to be appended
        end
    end
end

% function [w_act,w_non] = append_constraint(b,w_act,w_non,j,pbc)
% w_act = [w_act j]; % append constraint j to active set
% w_non = w_non(find(w_non ~= j)); % remove constraint j from nonactive set
% if ~isinf(b(pbc(j)))
%     w_non = w_non(find(w_non ~= pbc(j))); % remove constraint pbc(j) from nonactive set, if not unbounded
% end

% function [w_act,w_non] = remove_constraint(b,w_act,w_non,j,pbc)
% w_act = w_act(find(w_act ~= j)); % remove constraint j from active set
% w_non = [w_non j]; % append constraint j to nonactive set
% if ~isinf(b(pbc(j)))
%     w_non = [w_non pbc(j)]; % append constraint pbc(j) to nonactive set, if not unbounded
% end

function [w_act w_non C P x nab G g] = remove_constraint(wi,C,w_act,w_non,x,P,nb,nab,n,G,g,pbc,b) % wi is index of w_act
j = w_act(wi);

if j < nb+1                          % j is a bound and we have to reorganize the variables
    var1 = n-nab+1;
    var2 = n-nab+wi;

    temp = C(var1,:);
    C(var1,:) = C(var2,:);
```

```
    C(var2,:) = temp;

    temp = x(var1);
    x(var1) = x(var2);
    x(var2) = temp;

    temp = P(var1,:);
    P(var1,:) = P(var2,:);
    P(var2,:) = temp;

    temp = G(var1,var1);
    G(var1,var1) = G(var2,var2);
    G(var2,var2) = temp;

    temp = g(var1);
    g(var1) = g(var2);
    g(var2) = temp;
    nab = nab - 1;

    temp = w_act(wi);
    w_act(wi) = w_act(1);
    w_act(1) = temp;
    j = w_act(1);
end
w_act = w_act(find(w_act ~= j));                 % bound/ general constraint j is
     removed from active set
w_non = [w_non j];                               % bound/ general constraint j
     appended to nonactive set

if ~isinf(b(pbc(j)))
    w_non = [w_non pbc(j)];                      % append bound/constraint pbc(j)
         to nonactive set, if not unbounded
end

function [w_act w_non C P x nab G g Q] = append_constraint(j,C,w_act,w_non,x,P,
     nb,nab,n,G,g,Q,pbc,b) % j is index of C

if j < nb+1                                      % j is a bound and we have to
     reorganize the variables
    var1 = find(abs(C(:,j))==1);
    var2 = n-nab;

    temp = C(var1,:);
    C(var1,:) = C(var2,:);
    C(var2,:) = temp;

    temp = Q(var1,:);
    Q(var1,:) = Q(var2,:);
    Q(var2,:) = temp;

    temp = x(var1);
    x(var1) = x(var2);
    x(var2) = temp;

    temp = P(var1,:);
    P(var1,:) = P(var2,:);
    P(var2,:) = temp;

    temp = G(var1,var1);
    G(var1,var1) = G(var2,var2);
    G(var2,var2) = temp;

    temp = g(var1);
    g(var1) = g(var2);
    g(var2) = temp;
    nab = nab + 1;
    w_act = [j w_act];                           % j (is a bound) is appended to
         active set
else
    w_act = [w_act j];                           % j (is a general constraint)
         is appended to active set
end
w_non = w_non(find(w_non ~= j));                 % bound/ general constraint j
     is removed fom nonactive set
if ~isinf(b(pbc(j)))
    w_non = w_non(find(w_non ~= pbc(j)));        % remove bound/constraint pbc(j
         ) from nonactive set, if not unbounded
end

function f = objective(G,g,x)
f = 0.5*x'*G*x + g'*x;

function c = constraints(A,b,x)
c = A'*x - b;

function l = lagrangian(G,g,A,b,x,mu)
L = objective(G,g,A,b,x,mu) - mu(:)'*constraints(G,g,A,b,x,mu);
```

dual_active_set_method.m

```
function [x,mu,info,perf] = dual_active_set_method(G,g,C,b,w_non,pbc,opts,trace
    )

% DUAL_ACTIVE_SET_METHOD Solving an inequality constrained QP of the
% form:
%    min   f(x) = 0.5*x'*G*x + g*x
%    s.t.  A*x >= b,
% by solving a sequence of equality constrained QP's using the dual
% active set method. The method uses the range space procedure or the null
% space procedure to solve the KKT system. Both the range space and the
% null space procedures has been provided with factorization updates.
%
%    Call
%      x = dual_active_set_method(G, g, A, b, w_non, pbc)
%      x = dual_active_set_method(G, g, A, b, w_non, pbc, opts)
%      [x, mu, info, perf] = dual_active_set_method( ... )
%
% Input parameters
%    G      : The Hessian matrix of the objective function, size nxn.
%    g      : The linear term of the objective function, size nx1.
%    A      : The constraint matrix holding the constraints, size nxm.
%    b      : The right-hand side of the constraints, size mx1.
%    x      : Starting point, size nx1.
%    w_non : List of inactive constraints, pointing on constraints in A.
%    pbc    : List of corresponding constraints, pointing on constraints in
%             A. Can be empty.
%    opts  : Vector with 3 elements:
%            opts(1) = Tolerance used to stabilize the methods numerically.
%                      If |value| <= opts(1), then value is regarded as zero.
%            opts(2) = maximum no. of iteration steps.
%            opts(3) = 1 : Using null space procedure.
%                      2 : Using null space procedure with factorization
%                          update.
%                      3 : Using null space procedure with factorization
%                          update based on fixed and free variables. Can only
%                          be called, if the inequality constrained QP is
%                          setup on the form seen in QP_solver
%                      4 : Using range space procedure.
%                      5 : Using range space procedure with factorization
%                          update.
%      If opts is not given or empty, the default opts = [1e-8 1000 3].
%
% Output parameters
%    x     : The optimal solution.
%    mu    : The Lagrange multipliers at the optimal solution.
%    info : Performace information, vector with 3 elements:
%            info(1)   = final values of the objective function.
%            info(2)   = no. of iteration steps.
%            info(3)   = 1 : Feasible solution found.
%                        2 : No. of iteration steps exceeded.
%                        3 : Problem is infeasible.
%    perf : Performace, struct holding:
%            perf.x   : Values of x, size is nx(it+1).
%            perf.f   : Values of the objective function, size is 1x(it+1).
%            perf.mu : Values of mu, size is nx(it+1).
%            perf.c   : Values of c(x), size is mx(it+1).
%            perf.Wa : Active set, size is mx(it+1).
%            perf.Wi : Inactive set, size is mx(it+1).
%
%    By           : Carsten V\"olcker, s961572.
%                   Esben Lundsager Hansen, s022022.
%    Subject      : Numerical Methods for Sequential Quadratic Optimization.
%                   M.Sc., IMM, DTU, DK-2800 Lyngby.
%    Supervisor : John Bagterp Jørgensen, Assistant Professor.
%                   Per Grove Thomsen, Professor.
%    Date         : 07. June 2007.

% initialize options...
tol = opts(1);
it_max = opts(2);
method = opts(3);
[n,m] = size(C);
z = zeros(m,1);

% initialize containers...
%trace = (nargout > 3);
```

```
perf = [];
if trace
    X = repmat(zeros(n,1),1,it_max);
    F = repmat(0,1,it_max);
    Mu = repmat(z,1,it_max);
    Con = repmat(z,1,it_max);
    W_act = repmat(z,1,it_max);
    W_non = repmat(z,1,it_max);
end

if method == 3   % null space with FXFR-update
    nb = n*2;    % number of bounds
else
    nb = 0;
end
nab = 0;         % number of active bouns
P = eye(n);

x = -G\g;
mu = zeros(m,1);
w_act = [];

x0 = x;
f0 = objective(G,C,g,b,x,mu);
mu0 = mu;
con0 = constraints(G,C(:,w_non),g,b(w_non),x,mu);
w_act0 = z;
w_non0 = (1:1:m)';

Q = []; T = []; L = []; R = []; rem = [];        % both for range- and null_space_update and for null_space_update_FXFR
if method == 4 || method == 5 % range space or range space update
    chol_G = chol(G);
end
it_tot =0;
it = 0;
max_itr = it_max;
stop = 0;

while ~stop

    c = constraints(G,C(:,w_non),g,b(w_non),x,mu);
    if c >= -tol;%-1e-12%-sqrt(eps) % all elements must be >= 0
%         disp(['////// itr: ',int2str(it+1),' //////////////'])
%         disp('STOP: all inactive constraints >= 0')
        stop = 1;
    else
        % we find the most negative value of c
        [c_r,r] = min(c);
        r = w_non(r);
    end

    it = it + 1;
    if it >= max_itr % no convergence
        disp(['////// itr: ',int2str(it+1),' //////////////'])
        disp('STOP: it >= max_itr (outer while loop)' )
        stop  = 2;
    end

    it2 = 0;
    stop2 = max(0,stop);
    while ~stop2 %c_r < -sqrt(eps)
        it2 = it2 + 1;

        if method == 1
            [p,v] = null_space(G,C(:,w_act),-C(:,r),-zeros(length(w_act),1));
        end
        if method == 2
            [p,v,Q,T,L] = null_space_update(G,C(:,w_act),-C(:,r),zeros(length(w_act),1),Q,T,L,rem);
        end
        if method == 3
            Cr = C(:,r);
            A_ = C(:,w_act(nab+1:end));
            %         ajust C(:,r) to make it correspond to the factorizations of the
            %         Fixed variables (whenever -1 appears at variable i C(:,r)i should change sign)
            if nab % some bounds are in the active set
                u_idx = find(w_act > nb/2 & w_act< nb+1);
                var = n-nab+u_idx;
                Cr(var) = -Cr(var);
                A_(var,:) = -A_(var,:);
            end
            [p,v,Q,T,L] = null_space_updateFRFX(Q,T,L,G,A_,-Cr,zeros(length(w_act),1),nab,rem-nab);
        end
```

```
        if method == 4
            [p,v] = range_space(chol_G,C(:,w_act),-C(:,r),-zeros(length(w_act),1));
        end
        if method == 5
            [p,v,Q,R] = range_space_update(chol_G,C(:,w_act),-C(:,r),zeros(length(w_act),1),Q,R,rem);
        end

        if isempty(v)
            v = [];
        end

        arp = C(:,r)'*p;
        if abs(arp) <= tol % linear dependency
            if v >= 0 % solution does not exist
                disp(['///// itr: ',int2str(it+1),' ///////////// '])
                disp('STOP: v >= 0, PROBLEM IS INFEASIBLE!!!')
                stop = 3;
                stop2 = stop;
            else
                t = inf;
                for k = 1:length(v)
                    if v(k) < 0
                        temp = -mu(w_act(k))/v(k);
                        if temp < t
                            t = temp;
                            rem = k;
                        end
                    end
                end
                mu(w_act) = mu(w_act) + t*v;
                mu(r) = mu(r) + t;
                %           remove linear dependent constraint from A
                [w_act w_non C P x nab G g] = remove_constraint(rem,C,w_act,w_non,x,P,nb,nab,n,G,g,pbc,b); % rem is index of w_act
            end
        else
            % stepsize in dual space
            t1 = inf;
            for k = 1:length(v)
                if v(k) < 0
                    temp = -mu(w_act(k))/v(k);
                    if temp < t1
                        t1 = temp;
                        rem = k;
                    end
                end
            end
            % stepsize in primal space
            t2 = -constraints(G,C(:,r),g,b(r),x,mu)/arp;
            if t2 <= t1
                x = x + t2*p;
                mu(w_act) = mu(w_act) + t2*v;
                mu(r) = mu(r) + t2;
                % append constraint to active set
                [w_act w_non C P x nab G g Q] = append_constraint(r,C,w_act,w_non,x,P,nb,nab,n,G,g,Q,pbc,b); % r is index of C
            else
                x = x + t1*p;
                mu(w_act) = mu(w_act) + t1*v;
                mu(r) = mu(r) + t1;
                % remove constraint from active set
                [w_act w_non C P x nab G g] = remove_constraint(rem,C,w_act,w_non,x,P,nb,nab,n,G,g,pbc,b); % rem is index of w_act
            end
        end
        c_r = constraints(G,C(:,r),g,b(r),x,mu);
        if c_r > -tol
            stop2 = 1; % leave the inner while-loop but doesnt stop the algorithm
        end

        if it2 >= max_itr % no convergence (terminate the algorithm)
            disp(['///// itr: ',int2str(it+1),' ///////////// '])
            disp('STOP: it >= max_itr (inner while loop)')
            stop = 2;
            stop2 = stop;
        end

        % collecting output in containers...
        if trace
            if nb % method 3 is used
                X(:,it) = P'*x;
            else
                X(:,it) = x;
            end
```

```
            F(it) = objective(G,C,g,b,x,mu);
            Mu(:,it) = mu;
            Con(w_non,it) = constraints(G,C(:,w_non),g,b(w_non),x,mu);
            W_act(w_act,it) = w_act;
            W_non(w_non,it) = w_non;
        end
    end % while
    it_tot = it_tot + it2;
end % while
it_tot = it_tot + it;
if nb % method 3 is used
    x = P'*x;
    %     figure; spy(C(:,w_act)),pause
end

% building info...
info = [objective(G,C,g,b,x,mu) it_tot stop];
% building perf...
if trace
    X = X(:,1:it); X = [x0 X];
    F = (1:it); F = [f0 F];
    Mu = Mu(:,1:it); Mu = [mu0 Mu];
    Con = Con(:,1:it); Con = [con0 Con];
    W_act = W_act(:,1:it); W_act = [w_act0 W_act];
    W_non = W_non(:,1:it); W_non = [w_non0 W_non];
    perf = struct('x',{X},'f',{F},'mu',{Mu},'c',{Con},'Wa',{W_act},'Wi',{W_non});
end

function c = constraints(G,C,g,b,x,mu)
c = C'*x - b;

function [w_act w_non C P x nab G g] = remove_constraint(wi,C,w_act,w_non,x,P,nb,nab,n,G,g,pbc,b) % wi is index of w_act
j = w_act(wi);
if j < nb+1                          % j is a bound and we have to reorganize the variables
    var1 = n-nab+1;
    var2 = n-nab+wi;

    temp = C(var1,:);
    C(var1,:) = C(var2,:);
    C(var2,:) = temp;

    temp = x(var1);
    x(var1) = x(var2);
    x(var2) = temp;

    temp = P(var1,:);
    P(var1,:) = P(var2,:);
    P(var2,:) = temp;

    temp = G(var1,var1);
    G(var1,var1) = G(var2,var2);
    G(var2,var2) = temp;

    temp = g(var1);
    g(var1) = g(var2);
    g(var2) = temp;
    nab = nab - 1;

    temp = w_act(wi);
    w_act(wi) = w_act(1);
    w_act(1) = temp;
    j = w_act(1);
end
w_act = w_act(find(w_act ~= j));          % bound/ general constraint j is removed from active set
w_non = [w_non j];                        % bound/ general constraint j appended to nonactive set

if ~isempty(pbc)
    if ~isinf(b(pbc(j)))
        w_non = [w_non pbc(j)];           % append bound/constraint pbc(j) to nonactive set, if not unbounded
    end
end

function [w_act w_non C P x nab G g Q] = append_constraint(j,C,w_act,w_non,x,P,nb,nab,n,G,g,Q,pbc,b) % j is index of C
if j < nb+1                               % j is a bound and we have to reorganize the variables
    var1 = find(abs(C(:,j))==1);
    var2 = n-nab;

    temp = C(var1,:);
    C(var1,:) = C(var2,:);
```

```
    C(var2,:) = temp;

    temp = Q(var1,:);
    Q(var1,:) = Q(var2,:);
    Q(var2,:) = temp;

    temp = x(var1);
    x(var1) = x(var2);
    x(var2) = temp;

    temp = P(var1,:);
    P(var1,:) = P(var2,:);
    P(var2,:) = temp;

    temp = G(var1,var1);
    G(var1,var1) = G(var2,var2);
    G(var2,var2) = temp;

    temp = g(var1);
    g(var1) = g(var2);
    g(var2) = temp;
    nab = nab + 1;
    w_act = [j w_act];                    % j (is a bound) is appended to active set
else
    w_act = [w_act j];                    % j (is a general constraint) is appended to active set
end
w_non = w_non(find(w_non ~= j));          % bound/ general constraint j is removed fom nonactive set

if ~isempty(pbc)
    if ~isinf(b(pbc(j)))
        w_non = w_non(find(w_non ~= pbc(j)));     % remove bound/constraint pbc(j) from nonactive set, if not unbounded
    end
end

function f = objective(G,C,g,b,x,mu)
f = 0.5*x'*G*x + g'*x;
```

QP_solver.m

```
function [x,info,perf] = QP_solver(H,g,l,u,A,bl,bu,x,opts)

% QP_SOLVER Solving an inequality constrained QP of the form:
%    min  f(x) = 0.5*x'*H*x + g*x
%    s.t. l  <=  x  <= u
%         bl <= A*x <= bu,
% using the primal active set method or the dual active set method. The
% active set methods uses the range space procedure or the null space
% procedure to solve the KKT system. Both the range space and the null
% space procedures has been provided with factorization updates. Equality
% constraints are defined as l = u and bl = bu respectively.
%
% Call
%    x = QP_solver(H, g, l, u, A, bl, bu)
%    x = QP_solver(H, g, l, u, A, bl, bu, x, opts)
%    [x, info, perf] = QP_solver( ... )
%
% Input parameters
%    H     : The Hessian matrix of the objective function.
%    g     : The linear term of the objective function.
%    l     : Lower limits of bounds. Set as Inf, if unbounded.
%    u     : Upper limits of bounds. Set as -Inf, if unbounded.
%    A     : The constraint matrix holding the general constraints as rows.
%    bl    : Lower limits of general constraints. Set as Inf, if unbounded.
%    bu    : Upper limits of general constraints. Set as -Inf, if unbounded.
%    x     : Starting point. If x is not given or empty, then the dual active
%            set method is used, otherwise the primal active set method is
%            used.
%    opts  : Vector with 3 elements:
%            opts(1) = Tolerance used to stabilize the methods numerically.
%                      If |value| <= opts(1), then value is regarded as zero.
%            opts(2) = maximum no. of iteration steps.
%            Primal active set method:
%            opts(3) = 1 : Using null space procedure.
%                      2 : Using null space procedure with factorization
%                          update.
%                      3 : Using null space procedure with factorization
```

```matlab
%                   update based on fixed and free variables.
%               If opts(3) > 3, then opts(3) is set to 3 automatically.
%             Dual active set method:
%             opts(3) = 1 : Using null space procedure.
%                       2 : Using null space procedure with factorization
%                           update.
%                       3 : Using null space procedure with factorization
%                           update based on fixed and free variables.
%                       4 : Using range space procedure.
%                       5 : Using range space procedure with factorization
%                           update.
%      If opts is not given or empty, the default opts = [1e-8 1000 3].
%
% Output parameters
%   x     : The optimal solution.
%   info  : Performace information, vector with 3 elements:
%             info(1)    = final values of the objective function.
%             info(2)    = no. of iteration steps.
%             Primal active set method:
%             info(3)    = 1 : Feasible solution found.
%                          2 : No. of iteration steps exceeded.
%             Dual active set method:
%             info(3)    = 1 : Feasible solution found.
%                          2 : No. of iteration steps exceeded.
%                          3 : Problem is infeasible.
%   perf  : Performace, struct holding:
%             perf.x   : Values of x , size is nx(it+1).
%             perf.f   : Values of the objective function, size is 1x(it+1).
%             perf.mu  : Values of mu, size is nx(it+1).
%             perf.c   : Values of c(x), size is (n+n+m+m)x(it+1).
%             perf.Wa  : Active set, size is (n+n+m+m)x(it+1).
%             perf.Wi  : Inactive set, size is (n+n+m+m)x(it+1).
%      Size (n+n+m+m)x(it+1) is refering to indices i_l = 1:n, i_u = (n+1):2
%      n, i_bl = (2n+1):(2n+m) and i_bu = (2n+m+1):(2n+2m).
%
%   By          : Carsten V\"olcker, s961572.
%                 Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization.
%                 M.Sc., IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor.
%                 Per Grove Thomsen, Professor.
%   Date        : 07. June 2007.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Tune input and gather information                                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Tune...
l = l(:); u = u(:);
bl = bl(:); bu = bu(:);
g = g(:);
% Gather...
[m,n] = size(A);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set options                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if nargin < 9 | isempty(opts)
    tol = 1e-8;
    it_max = 1000;
    method = 3;
    opts = [tol it_max method];
else
    opts = opts(:)';
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check nargin/nargout                                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
error(nargchk(7,9,nargin))
error(nargoutchk(1,3,nargout))
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check input/output                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check H...
sizeH = size(H);
if sizeH(1) ~= n | sizeH(2) ~= n
    error(['Size of A is ',int2str(m),'x',int2str(n),', so H must be of size ',
         int2str(n),'x',int2str(n),'.'])
end
Hdiff = H - H';
if norm(Hdiff(:),inf) > eps*norm(H(:),inf) % relative check of biggest absolute
      value in Hdiff
    error('H must be symmetric.')
end
[dummy,p] = chol(H);
if p
    error('H must be positive definite.')
end
```

```
% Check g...
sizeg = size(g);
if sizeg(1) ~= n | sizeg(2) ~= 1
    error(['Size of A is ',int2str(m),'x',int2str(n),', so g must be a vector of ',int2str(n),' elements.'])
end
% Check l and u...
%l(40,1) = inf; % ???
sizel = size(l);
if sizel(1) ~= n | sizel(2) ~= 1
    error(['Size of A is ',int2str(m),'x',int2str(n),', so l must be a vector of ',int2str(n),' elements.'])
end
sizeu = size(u);
if sizeu(1) ~= n | sizeu(2) ~= 1
    error(['Size of A is ',int2str(m),'x',int2str(n),', so u must be a vector of ',int2str(n),' elements.'])
end
for i = 1:n
    if l(i,1) > u(i,1)
        error(['l(',int2str(i),') must be smaller than or equal to u(',int2str(i),').'])
    end
end
% Check bl and bu...
sizebl = size(bl);
if sizebl(1) ~= n | sizebl(2) ~= 1
    error(['Size of A is ',int2str(m),'x',int2str(n),', so bl must be a vector of ',int2str(m),' elements.'])
end
sizebu = size(bu);
if sizebu(1) ~= n | sizebu(2) ~= 1
    error(['Size of A is ',int2str(m),'x',int2str(n),', so bu must be a vector of ',int2str(m),' elements.'])
end
for i = 1:m
    if bl(i) > bu(i)
        error(['bl(',int2str(i),') must be smaller than or equal to bu(',int2str(i),').'])
    end
end
% Check x...
if nargin > 7 & ~isempty(x)
    %opts(1) = 1e-20; % ???
    feasible = 1;
    sizex = size(x);
    if sizex(1) ~= n | sizex(2) ~= 1
        error(['Size of A is ',int2str(m),'x',int2str(n),', so x must be a vector of ',int2str(n),' elements.'])
    end
    i_l = find(x - l < -opts(1)); i_u = find(x - u > opts(1));
    i_bl = find(A*x - bl < -opts(1)); i_bu = find(A*x - bu > opts(1));
    if ~isempty(i_l)
        disp(['Following bound(s) violated, because x - l < ',num2str(-opts(1)),': '])
        fprintf(['\b',int2str(i_l'),'.\n'])
        feasible = 0;
    end
    if ~isempty(i_u)
        disp(['Following bound(s) violated, because x - u > ',num2str(opts(1)),': '])
        fprintf(['\b',int2str(i_u'),'.\n'])
        feasible = 0;
    end
    if ~isempty(i_bl)
        disp(['Following general constraint(s) violated, because A*x - bl < ',num2str(-opts(1)),': '])
        fprintf(['\b',int2str(i_bl'),'.\n'])
        feasible = 0;
    end
    if ~isempty(i_bu)
        disp(['Following general constraint(s) violated, because A*x - bu > ',num2str(opts(1)),': '])
        fprintf(['\b',int2str(i_bu'),'.\n'])
        feasible = 0;
    end
    if ~feasible
        error('Starting point for primal active set method is not feasible.')
    end
end
% Check opts...
if length(opts) ~= 3
    error('Options must be a vector of 3 elements.')
end
i = 1;
if ~isreal(opts(i)) | isinf(opts(i)) | isnan(opts(i)) | opts(i) < 0
    error('opts(1) must be positive.')
```

```matlab
end
i = 2;
if ~isreal(opts(i)) | isinf(opts(i)) | isnan(opts(i)) | opts(i) < 0 | mod(opts(i),1)
    error('opts(2) must be a positive integer.')
end
i = 3;
if ~isreal(opts(i)) | isinf(opts(i)) | isnan(opts(i)) | opts(i) < 1 | 5 < opts(i) | mod(opts(i),1)
    error('opts(3) must be an integer in range 1 <= value <= 5.')
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize                                                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
I = eye(n);
At = A';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Organize bounds and constraints                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Convert input structure l <= I*x <= u and bl <= A*x <= bu to C*x >= b, where
    C = [I -I A -A] = [B A] (A = [A -A]) and b = [l -u bl -bu]...
B = [I -I]; % l <= I*x <= u --> I*x >= l & -I*x >= -u
A = [At -At]; % bl <= A*x <= bu --> A*x >= bl & -A*x >= -bu
C = [B A];
b = [l; -u; bl; -bu];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Build inactive set and corresponding constraints                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize inactive set...
w_non = 1:1:2*(n+m);
% Remove unbounded constraints from inactive set...
w_non = w_non(find(~isinf(b)));
% Indices of corresponding constraints...
cc = [(n+1):1:(2*n) 1:1:n (2*n+m+1):1:2*(n+m) 2*n+1:1:2*n+m]; % w_non = [i_l
    i_u i_bl i_bu] -> cc = [i_u i_l i_bu i_bl]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Startup info                                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Disregarded constraints...
%l(40,1) = -inf; % ???
i_l = find(isinf(l)); i_u = find(isinf(u));
i_bl = find(isinf(bl)); i_bu = find(isinf(bu));
if ~isempty(i_l)
    disp('Following constraint(s) disregarded, because l is unbounded:')
    disp(['i = [',int2str(i_l'),']'])
end
if ~isempty(i_u)
    disp('Following constraint(s) disregarded, because u is unbounded:')
    disp(['i = [',int2str(i_u'),']'])
end
if ~isempty(i_bl)
    disp('Following constraint(s) disregarded, because bl is unbounded:')
    disp(['i = [',int2str(i_bl'),']'])
end
if ~isempty(i_bu)
    disp('Following constraint(s) disregarded, because bu is unbounded:')
    disp(['i = [',int2str(i_bu'),']'])
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Call primal active set or dual active set method                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
trace = (nargout > 2); % building perf
if nargin < 8 | isempty(x)
    disp('Calling dual active set method.')
    [x,mu,info,perf] = dual_active_set_method(H,g,C,b,w_non,cc,opts,trace);
else
    disp('Starting point is feasible.')
    disp('Calling primal active set method.')
    if opts(3) > 3
        opts(3) = 3;
    end
    [x,mu,info,perf] = primal_active_set_method(H,g,C,b,x,w_non,cc,opts,trace);
end
% Display info...
if info(1) == 2
    disp('No solution found, maximum number of iteration steps exceeded.')
end
if info(1) == 3
    disp('No solution found, problem is unfeasible.')
end
disp('QPsolver terminated.')
```

LP_solver.m

```
function [x,f,A,b,Aeq,beq,l,u] = LP_solver(l,u,A,bl,bu)

% LP_SOLVER Finding a feasible point with respect to the constraints of an
% inequality constrained QP of the form:
%
%   min   f(x) = 0.5*x'*H*x + g*x
%   s.t.  l  <=  x  <= u
%         bl <= A*x <= bu,
%
% using the Matlab function linprog. Equality constraints are defined as
% l = u and bl = bu respectively.
%
% Call
%   x = LP_solver(l, u, A, bl, bu)
%   x = LP_solver(l, u, A, bl, bu, opts)
%   [x, f, A, b, Aeq, beq, l, u] = LP_solver( ... )
%
% Input parameters
%   l     : Lower limits of bounds. Set as Inf, if unbounded.
%   u     : Upper limits of bounds. Set as -Inf, if unbounded.
%   A     : The constraint matrix holding the general constraints as rows.
%   bl    : Lower limits of general constraints. Set as Inf, if unbounded.
%   bu    : Upper limits of general constraints. Set as -Inf, if unbounded.
%   opts  : Vector with 2 elements:
%           opts(1) = Tolerance deciding if constraints are equalities.
%                     If |bu - bl| <= opts(1), then constraint is regarded
%                     as an equality.
%           opts(2) = pseudo-infinity, can be used to replace (+-)Inf with a
%                     real value regarding unbounded variables and general
%                     constraints.
%     If opts is not given or empty, the default opts = [0 inf].
%
% Output parameters
%   x                 : Feasible point.
%   f,A,b,Aeq,beq,l,u : Output structured for further use in linprog.
%
%   By          : Carsten V\"olcker, s961572.
%                 Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization.
%                 M.Sc., IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor.
%                 Per Grove Thomsen, Professor.
%   Date        : 07. June 2007.

find_equality_constraints = 1; % see initialization of constraints below
equality_tol = 1e-8; % see initialization of constraints below
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Tune input and gather information                                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Tune...
l = l(:); u = u(:);
bl = bl(:); bu = bu(:);
% Gather...
[m,n] = size(A);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set options                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if nargin < 6 | isempty(opts)
    equality_tol = 1e-8;
    pseudoinf = 1e8;
else
    opts = opts(:)';
    % Check opts...
    if length(opts) ~= 2
        error('Options must be a vector of 2 elements.')
    end
    i = 1;
    if ~isreal(opts(i)) | isinf(opts(i)) | isnan(opts(i)) | opts(i) < 0
        error('opts(1) must be positive.')
    end
    i = 2;
    if ~isreal(opts(i)) | isinf(opts(i)) | isnan(opts(i)) | opts(i) < 0
        error('opts(2) must be positive.')
    end
    equality_tol = opts(1);
    pseudoinf = opts(2);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check nargin/nargout                                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
error(nargchk(5,6,nargin))
error(nargoutchk(1,8,nargout))
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check input                                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check l and u...
%l(40,1) = inf; % ???
sizel = size(l);
if sizel(1) ~= n | sizel(2) ~= 1
    error(['Size of A is ',int2str(m),'x',int2str(n),', so l must be a vector of ',int2str(n),' elements.'])
end
sizeu = size(u);
if sizeu(1) ~= n | sizeu(2) ~= 1
    error(['Size of A is ',int2str(m),'x',int2str(n),', so u must be a vector of ',int2str(n),' elements.'])
end
for i = 1:n
    if l(i,1) > u(i,1)
        error(['l(',int2str(i),') must be smaller than or equal to u(',int2str(i),').'])
    end
end
% Check bl and bu...
sizebl = size(bl);
if sizebl(1) ~= n | sizebl(2) ~= 1
    error(['Size of A is ',int2str(m),'x',int2str(n),', so bl must be a vector of ',int2str(m),' elements.'])
end
sizebu = size(bu);
if sizebu(1) ~= n | sizebu(2) ~= 1
    error(['Size of A is ',int2str(m),'x',int2str(n),', so bu must be a vector of ',int2str(m),' elements.'])
end
for i = 1:m
    if bl(i) > bu(i)
        error(['bl(',int2str(i),') must be smaller than or equal to bu(',int2str(i),').'])
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize input for linprog                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Replace +/-inf with pseudo inf (linprog requirement)...
if pseudoinf ~= inf
    l(find(l == -inf)) = -pseudoinf;
    u(find(u == inf)) = pseudoinf;
    bl(find(bl == -inf)) = -pseudoinf;
    bu(find(bu == inf)) = pseudoinf;
end
% Objective function f defined as a vector -> linprog is using inf-norm...
f = ones(n,1);
% Initialize constraints...
if 1
    % Find indices of equality and inequality constraints...
    in = 1:1:m; % indices of all constraints
    eq = find(abs(bu - bl) <= equality_tol)'; % indices of equality constraints
    for i = eq
        in = in(find(in ~= i)); % remove indices of equality constraints
    end
    % Split constraints into equality and inequality constraints...
    A_eq = A(:,eq); A_in = A(:,in);
    bl_eq = bl(eq); bl_in = bl(in);
    bu_eq = bu(eq); bu_in = bu(in);
    A = [-A_in'; A_in']; % constraint matrix of inequality constraints
    b = [-bl_in; bu_in]; % inequality constraints
    Aeq = A_eq'; % constraint matrix of equality constraints
    beq = (bl_eq + bu_eq)/2; % equality constraints
else
    % all constraints initialized as inequalities
    A = [-A'; A']; % onstraint matrix of inequality constraints
    b = [-bl; bu]; % inequality constraints
    Aeq = []; % no equality constraints
    beq = []; % no equality constraints
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Find feasible point and display user info                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Find feasible point using linprog with default settings...
disp('Calling linprog.')
[x,dummy,exitflag] = linprog(f,A,b,Aeq,beq,l,u,[],optimset('Display','off'));
% Replace pseudo limit with +/-inf...
if pseudoinf ~= inf
    b(find(b == -pseudoinf)) = -inf;
    b(find(b == pseudoinf)) = inf;
    l(find(l == -pseudoinf)) = -inf;
    u(find(u == pseudoinf)) = inf;
```

```
end
% Display info...
if exitflag ~= 1
    disp(['No feasible point found, exitflag = ',int2str(exitflag),', see "help
        linprog".'])
end
for i = 1:length(x)
    if x(i) > pseudoinf
        disp(['Feasible point regarded as infinite, x(',int2str(i),') > ',
            num2str(pseudoinf),'.'])
    end
    if x(i) < -pseudoinf
        disp(['Feasible point regarded as infinite, x(',int2str(i),') < ',
            num2str(-pseudoinf),'.'])
    end
end
disp('LPsolver terminated.')
```

D.3 Nonlinear Programming

SQP_solver.m

```
function [x, info, perf] = SQP_solver(modfun, modsens, costfun, costsens, x0, pi0, opts, varargin)
% SQP_SOLVER Solves a nonlinear program of the form
%
%   min  f(x)
%   s.t. h(x) >= 0
%
% Where f: R^n -> R, and h: R^n ->R^m, meaning that n is the number of variables and m
% is the number of constraints. SQP solves the program by use of the Lagrangian function
% L(x,y) = f(x)-y'h(x) which means that it is the following system that is
% solved
%
% nabla_x(L(x,y)) = nabla(f(x))-nabla(h(x))y = 0
% nabla_y(L(x,y)) = -h(x) = 0.
%
% Newtons method is used to approximate the solution. Each Newton step is calculated by
% solving a QP defined as
%
% min 0.5*delta_x'[nabla^2_xx(L(x,y))]delta_x + [nabla(f(x))]'delta_x
%  s.t. nabla(h(x))'delta_x >= -h(x)
%
% This means that the solution can only be found if nabla^2_xx(L(x,y)) is
% positive definite. An BFGS-update has been provided which approximates nabla^2_xx(L(x,y)).
% The solution is found by solving a sequence of these QPs. The
% dual active set method is used for solving the QP's.
%
% Call
%   [x, info, perf] = SQP_solver(@modfun, @modsens, @costfun, @costsens, x0, pi0, opts)
%
% Input parameters
%   @modfun     : functions that defines: h(x)          : R^n ->R^m
%   @modsens    : functions that defines: nabla(h(x))   : R^n ->R^(nxm)
%   @costfun    : functions that defines: f(x)          : R^n -> R
%   @costsens   : functions that defines: nabla(f(x))   : R^n -> R^n
%   x0          : starting_quess
%   pi0         : lagrange multipliers for the constraints.
%                       (could be a zero-vector of length m).
%   opts            :
%   opts  : Vector with 3 element:
%           opts(1) = Tolerance used to stabilize the methods numerically.
%                     If |value| < opts(1), then value is regarded as zero.
%           opts(2) = maximum no. of iteration steps.
%           opts(3) = 1 : Using null space procedure.
%                   = 2 : Using null space procedure with factorization
%                         update.
%     If opts(3) > 2, then opts(3) is set to 2 automatically.
%     If opts is not given or empty, the default opts = [1e-8 1000 2].
%
% Output parameters
%   x     : The optimal solution.
%   info  : Performace information, vector with 3 elements:
%           info(1)   = final values of f.
%           info(2)   = no. of iteration steps.
%           info(3)   = 1 : Feasible solution found.
%                       2 : No. of iteration steps exceeded.
%   perf  : Performace, struct holding:
%           perf.x        : Values of x from each iteration of SQP. Size is nxit.
%           perf.f        : Values of f(x) from each iteration of SQP. Size is 1xit.
%           perf.itQP     : Number of iterations from the dual active set method
%                           each time a QP is solved. Size is 1xit.
%           perf.stopQP   : reason why the dual active set method has
%                           terminated each time a QP is solved. Size
%                           is 1xit.
%   perf.stopQP(i) = 1: solution of QP has been found successfully.
%   perf.stopQP(i) = 2: solution of QP has not been found successfully as
%                       iteration number has exceeded max_iteration number.
%   perf.stopQP(i) = 3: solution of QP has not been found as the QP is infeasible.
%
%   By          : Carsten V\"olcker, s961572.
%                 Esben Lundsager Hansen, s022022.
```

```
%   Subject     : Numerical Methods for Sequential Quadratic Optimization.
%                 M.Sc., IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor.
%                 Per Grove Thomsen, Professor.
%   Date        : 07. June 2007.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if nargin < 7 | isempty(opts)
    tol = 1e-8;
    it_max = 1000;
    method = 2;
    opts = [tol it_max method];
else
    if opts(3) > 2
        opts(3) = 2;
    end
    opts = opts(:)';
end

f0 = feval(costfun, x0, varargin{:});
g0 = feval(modfun, x0, varargin{:});
c  = feval(costsens, x0, varargin{:});
A  = feval(modsens, x0, varargin{:});
W  = eye(length(x0));
w_non = (1:1:length(g0));

stop = 0;
tol = opts(1);
it_max = opts(2);
itr = 0;
n = length(x0);
xinit = x0;
finit = f0;

% initialize containers...
trace = (nargout > 2);
if trace
    X_ = repmat(zeros(n,1),1,it_max); % x of SQP
    F = repmat(0,1,it_max); % function value of SQP
    It = repmat(0,1,it_max); % no. iterations of QP
    Stop = repmat(0,1,it_max); % stop of QP
end
max_itr = it_max;

while ~stop
    X(:,itr+1) = x0;
    itr = itr+1;
    if(itr > max_itr)
        stop = 2;
    end

    [delta_x, mu,info] = dual_active_set_method(W,c,A,-g0,w_non,[],opts);

    if (abs(c'*delta_x) + abs(mu'*g0)) < tol
        disp('solution has been found')
        stop = 1;
    else

        if itr == 1
            sigma = abs(mu);
        else
            for i=1:length(mu)
                sigma(i) = max(abs(mu(i)), 0.5*(sigma(i)+abs(mu(i))));
            end
        end

        [alpha,x,f,g] = line_search_algorithm(modfun,costfun,f0,g0,c,x0,delta_x
            ,sigma,1e-4);

        pii = pi0 + alpha*(mu-pi0);

        nabla_L0 = c-A*pii;
        c  = feval(costsens, x, varargin{:});
        A  = feval(modsens, x, varargin{:});
        nabla_L = c-A*pii;
        s = x - x0;
        y = nabla_L - nabla_L0;
        sy = s'*y;
        sWs = s'*W*s;
        if(sy >= 0.2*sWs)
            theta = 1;
        else
            theta = (0.8*sWs)/(sWs-sy);
        end
        Ws = W*s;
        sW = s'*W;
        r = theta*y+(1-theta)*Ws;
```

```
        W = W-(Ws*sW)/sWs+(r*r')/(s'*r);
        x0 = x;
        pi0 = pii;
        f0 = f;
        g0 = g;
    end

    % collecting output in containers...
    if trace
        X_(:,itr) = x0;
        F(itr) = f0;
        It(itr) = info(2);
        Stop(itr) = info(3);
    end
end

info = [f0 itr stop]; % SQP info
x = x0;
% building perf...
if trace
    X_ = X_(:,1:itr); X_ = [xinit X_];
    F = F(1:itr); F = [finit F];
    It = It(1:itr); It = [0 It];
    Stop = Stop(1:itr); Stop = [0 Stop];
    perf = struct('x',{X},'f',{F},'itQP',{It},'stopQP',{Stop});
end
```

D.4 Updating the Matrix Factorizations

givens_rotation_matrix.m

```
function [c,s] = givens_rotation_matrix(a,b)

% GIVENS_ROTATION_MATRIX: calculates the elements c and s which are used to
% introduce one zero in a vector of two elements
%
%   Call
%       [c s] = givens_rotation_matrix(a,b)
%
%   Input parameters
%       a and b are the two elements of the vector where we want to
%       introduce one zero.
%
%   Output parameters
%       c and s is used to construct the givens_rotation_matrix Qgivens: [c -s;
%     s c].
%       Now one zero is introduced: Qgivens*[a b]' = [gamma 0],
%       where gamma is the length of [a b] is abs(gamma)
%
%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor : John Bagterp Jørgensen, Assistant Professor & Per Grove
%    Thomsen, Professor.
%   Date        : 31. october 2006.
%   Reference   : ---------------------



% if(b==0)
%     c = 1;
%     s = 0;
% else
%     if(abs(b)>abs(a))
%         tau = -a/b;
%         s = 1/sqrt(1+tau*tau);
%         c = tau*s;
%     else
%         tau = -b/a;
%         c = 1/sqrt(1+tau*tau);
%         s = tau*c;
%     end
% end
 G = givens(a,b);
 c = G(1,1);
 s = G(2,1);
```

range_space_update.m

```
function [x,u,Q,R] = range_space_update(L,A,g,b,Q,R,col_rem)
% RANGE_SPACE_UPDATE uses the range-space procedure for solving a QP problem:
    min f(x)=0.5*x'Gx+g'x st: A'x=b,
% where A contains m constraints and the system has n variables.
    RANGE_SPACE_UPDATE contains methods for
% updating the factorizations using Givens rotations.

%       *** when solving an inequality constrained QP, a seqence of equality
%       constrained QPs are solved. The difference between two of these
%       following equality constrained QP is one appended constraint at the
%       last index of A, or a constraint removed at index col_rem of A.
%
%   Call
%       [x,u,Q,R] = range_space_update(L,A,g,b,Q,R,col_rem)
%

%   Input parameters
%       L                : is the Cholesky factorization of the Hessian matrix G
%     of f(x). L is nxn
%       A                : is the constraint matrix. The constraints are columns
%     in A. A is nxm
%       g                : contains n elements
%       b                : contains m elements
```

```
%       Q and R : is the QR-factorization of the QP which has just been solved
%       (if not the first iteration) in the sequence descibed in ***.
%       col_rem          : is the index at which a constraint has been removed from A.
%
%       Q, R and col_rem can be empty [] which means that The QP
%       is the first one in the sequence (see ***).

%   Output parameters
%       x                : is the optimized point
%       u                : is the corresponding Lagrangian Multipliers
%       Q and R : is the QR-factorization of A

%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date        : 11. february 2007.
%   Reference   : ---------------------

[nA,mA] = size(A);
[nR,mR] = size(R);
K = L\A;
w = L\g;
z = b+(w'*K)';
if isempty(Q) && isempty(R)
%    disp('complete factorization');
    [Q,R] = qr(K);
elseif mR < mA % new column has been appended to A
%    disp('append update');
    [Q, R] = qr_fact_update_app_col(Q, R, K(:,end));
elseif mR > mA % column has been removed from A at index col_rem
%    disp('remove update ');
    [Q, R] = qr_fact_update_rem_col(Q, R, col_rem);
end
u = R(1:length(z),:)'\z;
u = R(1:length(z),:)\u;
y = K*u-w;
x = L'\y;
```

qr_fact_update_app_col.m

```
function [Q,R] = qr_fact_update_app_col(Q,R,col_new)
% QR_FACT_UPDATE_APP_COL updates the qr-factorization when a single column is
%   appended at index m+1. And the factorization from before adding the column is known
%   :(Q_old and R_old)
%
%   Call
%       [q r] = qr_fact_update_app_col(Q_old, R_old, col_new)
%
%   Input parameters
%       Q_old is the Q part of the QR-factorization from the former
%           matrix A ( the matrix we want to append one column at index m+1).
%       R_old is the R part from the QR-factorization from the former
%           matrix A ( the matrix we want to append one column at index m+1).
%       col_new is the column we want to append
%
%   Output parameters
%       Q is the updated Q-matrix
%       R is the updated R-matrix (everything but the upper mxm matrix is zeros)
%
%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date        : 31. october 2006.
%   Reference   : ---------------------

[n m] = size(R);
sw = (col_new'*Q)';
for j = n:-1:m+2
    i = j-1;
     [c s] = givens_rotation_matrix(sw(i),sw(j));
     e1 = sw(i)*c - sw(j)*s;
     sw(j) = sw(i)*s + sw(j)*c;
     sw(i) = e1;
```

```
    v1 = Q(:,i)*c - Q(:,j)*s;
    Q(:,j) = Q(:,i)*s + Q(:,j)*c;
    Q(:,i) = v1;
end
R = [R sw];
```

qr_fact_update_rem_col.m

```
function [Q,R] = qr_fact_update_rem_col(Q,R,col_index)
% QR_FACT_UPDATE_REM_COL updates the qr-factorization when a single column is
%  removed.
%
%   Call
%       [q r] = qr_fact_update_rem_col(Q_old, R_old, col_new)
%
%   Input parameters
%       Q_old is the Q part from the QR-factorization from the former
%           matrix A ( the matrix we want to remove one column ).
%       R_old is the R part from the QR-factorization from the former
%          matrix A ( the matrix we want to remove one column ).
%       col_index is the index of the column we want to remove
%
%   Output parameters
%       Q is the updated Q-matrix
%       R is the updated R-matrix (everything but the upper mxm matrix is zeros )
%
%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date        : 31. october 2006.
%   Reference   : --------------------

[n m] = size(R);
t = m - col_index;
for i = 1:1:t
    j = i+1;
    [c s] = givens_rotation_matrix(R(col_index+i-1,col_index+i), R(col_index+j-1,col_index+i));
    v1 = R(col_index+i-1,col_index+1:end)*c - R(col_index+j-1,col_index+1:end)*s;
    R(col_index+j-1,col_index+1:end) = R(col_index+i-1,col_index+1:end)*s + R(col_index+j-1,col_index+1:end)*c;
    R(col_index+i-1,col_index+1:end) = v1;
    q1 = Q(:,col_index+i-1)*c - Q(:,col_index+j-1)*s;
    Q(:,col_index+j-1) = Q(:,col_index+i-1)*s + Q(:,col_index+j-1)*c;
    Q(:,col_index+i-1) = q1;
end
R = [R(:,1:col_index-1) R(:,col_index+1:end)];
```

null_space_update.m

```
function [x,u,Q_new,T_new,L_new] = null_space_update(G,A,g,b,Q_old,T_old,L_old,col_rem)

% NULL_SPACE_UPDATE uses the null-space procedure for solving a QP problem: min f(x)=0.5*x'Gx+g'x st: A'x=b,
% where A contains m constraints and the system has n variables. NULL_SPACE_UPDATE contains methods for
% updating the factorizations using Givens rotations.

%       *** when solving an inequality constrained QP, a seqence of equality
%       constrained QPs are solved. The difference between two of these
%       following equality constrained QP is one appended constraint at the
%       last index of A, or a constraint removed at index col_rem of A.
%
%   Call
%       [x,u,Q_new,T_new,L_new] = null_space_update(G,A,g,b,Q_old,T_old,L_old,col_rem)
%

```

```
%   Input parameters
%       G                 : is the Hessian matrix of f(x). G is nxn
%       A                 : is the constraint matrix. The constraints are columns in A. A is nxm
%       g                 : contains n elements
%       b                 : contains m elements
%       Q_old and T_old : is the QT-factorization of the QP which has just been solved
%       (if not the first iteration) in the sequence descibed in ***. The
%       T part of the QT-factorization is lower triangular
%       L_old             : is the Cholesky factorization of the reduced Hessian
%       matrix of the QP just solved (see ***).
%       col_rem           : is the index at which a constraint has been removed from A.
%
%       Q_old, T_old, L_old and col_rem can be empty [] which means that The QP
%       is the first one in the sequence (see ***).

%   Output parameters
%       x                 :is the optimized point
%       u                 :is the corresponding Lagrangian Multipliers
%       Q_new and T_new :is the QT-factorization of A
%       L_new             :is the Cholesky factorization of the reduced Hessian matrix.

%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date        : 08. february 2007.

[nA,mA] = size(A);
[nT,mT] = size(T_old);

dimNulSpace = nA-mA;

Q_new = Q_old;
T_new = T_old;
L_new = L_old;
if isempty(Q_old) && isempty(T_old) && isempty(L_old)
    %disp('complete factorization');
    [Q,R] = qr(A);
    Itilde = flipud(eye(nA));
    T_new = Itilde*R;
    Q_new = Q*Itilde;
    Q1 = Q_new(:,1:dimNulSpace);
    Gz = Q1'*G*Q1;
    L_new = chol(Gz)';
elseif mT < mA % new column has been appended to A
    % disp('append update');
    [Q_new, T_new, L_new] = null_space_update_fact_app_col(Q_old, T_old, L_old, A(:,end));
elseif mT > mA% column has been removed from A at index col_rem
    % disp('remove update ');
    [Q_new, T_new, L_new] = null_space_update_fact_rem_col(Q_old, T_old, L_old, G, col_rem);
end

Q1 = Q_new(:,1:dimNulSpace);
Q2 = Q_new(:,dimNulSpace+1:nA);
T_newMark = T_new(dimNulSpace+1:end,:);
py = T_newMark'\b;
gz = -((G*(Q2*py) + g)'*Q1)';
z = L_new\gz;
pz = L_new'\z;
x = Q2*py + Q1*pz;
u = ((G*x + g)'*Q2)';
u = T_newMark\u;
```

null_space_update_fact_app_col.m

```
function [Q, T, L] = null_space_update_fact_app_col(Q, T, L, col_new)
% NULL_SPACE_UPDATE_FACT_APP_COL updates the QT-factorization of A when a
% single column col_new is appended to A as the last column. The resulting
% constraint matrix is Abar = [A col_new]. The corresponding QP problem has a reduced Hessian
% matrix redH and the cholesky factorization of redH is L_old.

%   Call
```

```
%       [Q, T, L] = null_space_update_fact_app_col(Q, T, L, col_new)

%   Input parameters
%       Q and T          : is the QT-factorization of A
%       L                : is the cholesky factorization of the reduced Hessian matrix of the corresponding QP problem.
%       col_new          : is the column that is appended to A: Abar = [A col_new]

%   Output parameters
%       Q and T          :is the QT-factorization of Abar
%       L                :is the Cholesky factorization of the reduced Hessian matrix of the new QP problem.

%   By           : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject      : Numerical Methods for Sequential Quadratic Optimization,
%                  Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date         : 08. february 2007.
%   Reference    : ---------------------

[n,m] = size(T);
dimNullSpace = n - m;
wv = (col_new'*Q)';
for i = 1:dimNullSpace-1
    j = i+1;
    [s,c] = givens_rotation_matrix(wv(i),wv(j));
    temp = wv(i)*c + wv(j)*s;
    wv(j) = wv(j)*c - wv(i)*s;
    wv(i) = temp;
    temp = Q(:,i)*c + Q(:,j)*s;
    Q(:,j) = Q(:,j)*c - Q(:,i)*s;
    Q(:,i) = temp;
    temp = L(i,:)*c + L(j,:)*s;
    L(j,:) = L(j,:)*c - L(i,:)*s;
    L(i,:) = temp;
end
for i = 1:dimNullSpace-1
    j = i+1;
    [c,s] = givens_rotation_matrix(L(i,i),L(i,j));
    temp = L(:,i)*c - L(:,j)*s;
    L(:,j) = L(:,j)*c + L(:,i)*s;
    L(:,i) = temp;
end
T = [T wv];
L = L(1:dimNullSpace-1,1:dimNullSpace-1);
```

null_space_update_fact_rem_col.m

```
function [Q, T, L] = null_space_update_fact_rem_col(Q, T, L, G, col_rem)
% NULL_SPACE_UPDATE_FACT_REM_COL updates the QT-factorization of A when a
% column is removed from A at column-index col_rem. The new Constraint matrix is called Abar.
% The corresponding QP problem has a reduced Hessian matrix redH and the cholesky factorization
% of redH is L.

%   Call
%       [Q, T, L] = null_space_update_fact_rem_col(Q, T, L, G, col_rem)

%   Input parameters
%       Q and T : is the QT-factorization of A
%       L                : is the cholesky factorization of the reduced Hessian matrix of the corresponding QP problem.
%       G                : is the Hessian matrix of the QP problem.
%       col_rem          : is the column-index at which a column has been removed from A

%   Output parameters
%       Q and T      :is the QT-factorization of Abar
%       L            :is the Cholesky factorization of the reduced Hessian matrix of the new QP problem.

%   By           : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject      : Numerical Methods for Sequential Quadratic Optimization,
%                  Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date         : 08. february 2007.
```

```
%   Reference   : ----------------------

[n,m] = size(T);
dimNulSpace = n-m;
j = col_rem;
mm = m-j;
nn = mm+1;


for i=1:1:mm
    idx1 = nn-i;
    idx2 = idx1+1;
    [s,c] = givens_rotation_matrix(T(dimNulSpace+idx1,j+i),T(dimNulSpace+idx2,j+i));
    temp = T(dimNulSpace+idx1,j+1:end)*c + T(dimNulSpace+idx2,j+1:end)*s;
    T(dimNulSpace+idx2,j+1:end) = -T(dimNulSpace+idx1,j+1:end)*s + T(dimNulSpace+idx2,j+1:end)*c;
    T(dimNulSpace+idx1,j+1:end) = temp;
    temp = Q(:,dimNulSpace+idx1)*c + Q(:,dimNulSpace+idx2)*s;
    Q(:,dimNulSpace+idx2) = -Q(:,dimNulSpace+idx1)*s + Q(:,dimNulSpace+idx2)*c;
    Q(:,dimNulSpace+idx1) = temp;
end

T = [T(:,1:j-1) T(:,j+1:end)];
z = Q(:,dimNulSpace+1);
l = L\((G*z)'*Q(:,1:dimNulSpace))';
delta = sqrt(z'*G*z-l'*l);
L = [L zeros(dimNulSpace,1);l' delta];
```

null_space_updateFRFX.m

```
function [x,u,Q_fr,T_fr,L_fr] = null_space_updateFRFX(Q_fr,T_fr,L_fr,G,A,g,b,dim_fx,col_rem)

% NULL_SPACE_UPDATE_FRFX uses the same procedure as NULL_SPACE_UPDATE for solving  f(x)=0.5*x'Gx+g'x st: A'x=b,
%(so please take a look at it). The difference is that NULL_SPACE_UPDATE_FRFX takes advantage of the fact that some of the active constraints
% are bounds (usually). An active bound correspond to one fixed variable. This means that x can be devided into [x_free, x_fixed]'
% where x_fixed are those variables which are fixed. The part of the
% factorizations which correspond to the fixed variables can not be changes
% (as they are fixed) and this means that we are only required to
% refactorize the part which correspond to the free variables.

%       *** when solving an inequality constrained QP, a seqence of equality
%       constrained QPs are solved. The difference between two of these
%       following equality constrained QP is one appended constraint or one
%       removed constaint
%
%   Call
%       [x,u,Q_fr,T_fr,L_fr] = null_space_updateFRFX(Q_fr,T_fr,L_fr,G,A,g,b,dim_fx,col_rem)
%
%   Input parameters
%       G               : is the Hessian matrix of f(x). G is nxn
%       A               : is the constraint matrix which only contains active general constraints (the bound-constraints has been removed).
%                         The dimension of A is nxm_fr (n is number
%                         of variables and m_fr is the number of
%                         active general constraints)
%       g               : is the gradient of f(x) and the dimension is nx1
%       b               : contains the max values of the constraints (both
%                         general and an bound constraints) and therefore the dimension is
%                         (m_fr+mfx)x1.
%       Q_fr and T_fr are the free part of the QT-factorization of the part of the QP which has just been solved
%       (if not the first iteration) in the sequence descibed in ***. The
%       T part of the QT-factorization is lower triangular
%       L_old           : is the Cholesky factorization of the reduced Hessian
%       matrix of the QP just solved (see ***).
%       col_rem         : is the index at which a constraint has been removed from A (if a constraint has been appended this
%                         variable is unused.
%
%       Q_old, T_old, L_old and col_rem can be empty [] which means that The QP
%       is the first one in the sequence (see ***).
%       dim_fx          : number of fixed variables
%
```

```
%   Output parameters
%       x                   :is the solution
%       u                   :is the corresponding Lagrangian Multipliers
%       Q_fr and T_fr       :is the QT-factorization of A corresponding to the
%                                free variables.
%       L_fr                :is the Cholesky factorization of the reduced
%                                Hessian matrix.
%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date        : 08. february 2007.
%   Reference   : ---------------------

[nT mT] = size(T_fr);
[nA mA] = size(A);
dim_fx_old = nA-nT;

if isempty(A) % nothing to factorize
%     disp('A is empty')
    C = eye(dim_fx);
    C = [zeros(length(g)-dim_fx,dim_fx); C];
    [x u] = null_space(G,C,g,b);
    Q_fr = []; T_fr = []; L_fr = []; A_fx = [];

elseif isempty(T_fr) || ((mA == mT) && (dim_fx == dim_fx_old)) % complete factorization
   % disp('complete factorization')
    A_fr = A(1:end-dim_fx,:);
    [n m] = size(A_fr);
    G_frfr = G(1:n,1:n);
    dns = n-m;
    [Q,R] = qr(A_fr);
    Itilde = flipud(eye(n));
    T_fr = Itilde*R;
    Q_fr = Q*Itilde;
    Qz = Q_fr(:,1:dns);
    Gz = Qz'*G_frfr*Qz;
    L_fr = chol(Gz)';
    A_fx = A(end-dim_fx+1:end,:);
    [x u] = help_fun(Q_fr,T_fr,L_fr,A_fx,G,g,b);

elseif mA > mT % one general constraint has been appended
%     disp('append general constraint')
    [Q_fr, T_fr, L_fr] = null_space_update_fact_app_general_FRFX(Q_fr, T_fr, L_fr, A(:,end));
    dim_fr = size(T_fr,1);
    A_fx = A(dim_fr+1:end,:);
    [x u] = help_fun(Q_fr,T_fr,L_fr,A_fx,G,g,b);
elseif mA < mT % one general constraint has been removed at indx col_rem
%     disp('remove general constraint')
    dim_fr = size(T_fr,1);
    G_frfr = G(1:dim_fr,1:dim_fr);
    [Q_fr, T_fr, L_fr] = null_space_update_fact_rem_general_FRFX(Q_fr, T_fr, L_fr,G_frfr,col_rem);
    A_fx = A(dim_fr+1:end,:);
    [x u] = help_fun(Q_fr,T_fr,L_fr,A_fx,G,g,b);

elseif dim_fx > dim_fx_old % one bound has been appended
%      disp('append bound')
    [Q_fr, T_fr, L_fr] = null_space_update_fact_app_bound_FRFX(Q_fr, T_fr, L_fr);
    dim_fr = size(T_fr,1);
    A_fx = A(dim_fr+1:end,:);
    [x u] = help_fun(Q_fr,T_fr,L_fr,A_fx,G,g,b);

elseif dim_fx < dim_fx_old % one bound has been removed
%     disp('remove bound')
    [nT mT] = size(T_fr);
    dns = nT-mT;
    T_fr = T_fr(dns+1:end,:);
    T_fr = [T_fr; A(nT+1,:)];
    [Q_fr, T_fr, L_fr] = null_space_update_fact_rem_bound_FRFX(Q_fr, T_fr, L_fr, G);
    A_fx = A(nT+2:end,:);
    [x u] = help_fun(Q_fr,T_fr,L_fr,A_fx,G,g,b);
end

function [x_new u_new] = help_fun(Q_fr,T_fr,L_fr,A_fx,G,g,b)
% disp('help_fun')
[nT,mT] = size(T_fr);
dns = nT-mT;
dim_fr = nT;
dim_fx = length(g)-dim_fr;
Q1 = Q_fr(:,1:dns);
Q2 = Q_fr(:,dns+1:nT);
```

```
T_fr = T_fr(dns+1:end,:);
b_fr = b(dim_fx+1:end);
x_fx = b(1:dim_fx);
if dim_fx
    temp = (x_fx'*A_fx)';
    b_fr = b_fr-temp;
end
py = T_fr'\b_fr;
G_frfr = G(1:dim_fr,1:dim_fr);
g_fr = g(1:dim_fr);
gz = -((G_frfr*(Q2*py) + g_fr)'*Q1)';
z = L_fr\gz;
pz = L_fr'\z;
x_fr = Q2*py + Q1*pz;
%compute Lagrangian multipliers
c = G*[x_fr; x_fx] + g;
c_fr = c(1:dim_fr);
c_fx = c(dim_fr+1:end);
Y_fr = Q_fr(1:dim_fr,dns+1:dim_fr);
u_I = T_fr\(c_fr'*Y_fr)';
u_B = c_fx-A_fx*u_I;
x_new = [x_fr; x_fx];
u_new = [u_B; u_I];
```

null_space_update_fact_app_general_FRFX.m

```
function [Q_fr, T_fr, L_fr] = null_space_update_fact_app_general_FRFX(Q_fr, T_fr, L_fr, col_new)

% NULL_SPACE_UPDATE_FACT_APP_GENERAL_FRFX updates the QT-factorization of A when a
% general constraint: col_new is appended to A as the last column. The resulting
% constraint matrix is Abar = [A col_new]. The corresponding QP problem has a reduced Hessian
% matrix redH and the cholesky factorization of redH is L_fr. It is only
% the part corresponding to the free variables which are updated (the fixed
% part are not changing)
%   Call
%       [Q_fr, T_fr, L_fr] = null_space_update_fact_app_general_FRFX(Q_fr, T_fr, L_fr, col_new)
%
%   Input parameters:
%       Q_fr and T_fr   : is the QT-factorization of A (the part
%                         corresponding to the free variables)
%       L_fr            : is the cholesky factorization of the reduced Hessian matrix of the corresponding QP problem.
%       col_new         : is the general constraint that is appended to A: Abar = [A col_new]

%   Output parameters
%       Q_fr and T_fr   :is the QT-factorization of Abar(the part
%                         corresponding to the free variables)
%       L_fr            :is the Cholesky factorization of the reduced Hessian matrix of the new QP problem.

%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date        : 08. february 2007.
%   Reference   : ---------------------

[n,m] = size(T_fr);
dns = n-m;
Z_fr = Q_fr(:,1:dns);
Y_fr = Q_fr(:,dns+1:end);
T_fr = T_fr(dns+1:end,:);
a_fr = col_new(1:n);
wv = (a_fr'*Q_fr)';
w = wv(1:dns);
v = wv(dns+1:end);
for i =1:length(w)-1
    j = i+1;
    [s,c] = givens_rotation_matrix(w(i),w(j));

    temp = w(i)*c + w(j)*s;
    w(j) = -w(i)*s + w(j)*c;
    w(i) = temp;
```

```
     temp = Z_fr(:,i)*c + Z_fr(:,j)*s;
     Z_fr(:,j) = -Z_fr(:,i)*s + Z_fr(:,j)*c;
     Z_fr(:,i) = temp;

     temp = L_fr(i,:)*c + L_fr(j,:)*s;
     L_fr(j,:) = -L_fr(i,:)*s + L_fr(j,:)*c;
     L_fr(i,:) = temp;
end
gamma = w(end);
T_fr = [zeros(1,size(T_fr,2)) gamma;T_fr v];
L_fr = L_fr(1:end-1,:);
[nn mm] = size(L_fr);
for i=1:1:nn
     j=i+1;
     [c,s] = givens_rotation_matrix(L_fr(i,i),L_fr(i,j));

     temp = L_fr(:,i)*c - L_fr(:,j)*s;
     L_fr(:,j) = L_fr(:,i)*s + L_fr(:,j)*c;
     L_fr(:,i) = temp;
end
Q_fr = [Z_fr Y_fr];
T_fr = [zeros(dns-1,size(T_fr,2));T_fr];
L_fr = L_fr(:,1:dns-1);
```

null_space_update_fact_rem_general_FRFX.m

```
function [Q_new, T_new, L_new] = null_space_update_fact_rem_general_FRFX(Q_fr, T_fr, L_fr, G_frfr, j)

% NULL_SPACE_UPDATE_FACT_REM_GENERAL_FRFX updates the QT-factorization corresponding to the free variables of A when a
% general constraint is removed from A at column-index j. The new Constraint matrix is called Abar.
% The corresponding QP problem has a reduced Hessian matrix redH and the cholesky factorization
% of redH is L_fr.

%   Call
%       [Q_new, T_new, L_new] = null_space_update_fact_rem_col(Q_fr, T_fr, L_fr, G_frfr, col_rem)

%   Input parameters
%       Q_fr and T_fr     : is the QT-factorization of A (the part
%                           corresponding to the free variables)
%       L_fr              : is the cholesky factorization of the reduced Hessian matrix of the corresponding QP problem (the part
%                           corresponding to the free variables).
%       G_frfr            : is the Hessian matrix of the QP problem (the part
%                           corresponding to the free variables).
%       col_rem           : is the column-index at which a general constraint has been removed from A

%   Output parameters
%       Q_new and T_new  :is the QT-factorization of Abar(the part
%                          corresponding to the free variables).
%       L_new            :is the Cholesky factorization of the reduced Hessian matrix of the new QP problem (the part
%                          corresponding to the free variables).

%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date        : 08. february 2007.
%   Reference   : ---------------------

[n,m] = size(T_fr);
dns = n-m;
T_fr = T_fr(dns+1:end,:);
T11 = T_fr(m-j+2:m,1:j-1);
N = T_fr(1:m-j+1,j+1:end);
M = T_fr(m-j+2:end,j+1:end);
Q1 = Q_fr(:,1:dns);
Q21 = Q_fr(:,dns+1:n-j+1);
Q22 = Q_fr(:,n-j+2:end);
[nn mm] = size(N);

for i=1:1:mm
```

```
    idx1 = nn-i;
    idx2 = idx1+1;
    [s,c] = givens_rotation_matrix(N(idx1,i),N(idx2,i));

    temp = N(idx1,:)*c + N(idx2,:)*s;
    N(idx2,:) = -N(idx1,:)*s +N(idx2,:)*c;
    N(idx1,:) = temp;

    temp = Q21(:,idx1)*c + Q21(:,idx2)*s;
    Q21(:,idx2) = -Q21(:,idx1)*s + Q21(:,idx2)*c;
    Q21(:,idx1) = temp;
end
N = N(2:end,:);
T_new = [zeros(nn-1,j-1) N; T11 M];
T_new = [zeros(dns+1,m-1); T_new];
Q_new = [Q1 Q21 Q22];
z = Q_new(:,dns+1);
l = L_fr\((G_frfr*z)'*Q1)';
delta = sqrt(z'*G_frfr*z-l'*l);
L_new = [L_fr zeros(dns,1);l' delta];
```

null_space_update_fact_app_bound_FRFX.m

```
function [Q_fr, T_fr, L_fr] = null_space_update_fact_app_bound_FRFX(Q_fr, T_fr, L_fr)

% NULL_SPACE_UPDATE_FACT_APP_BOUND_FRFX updates the QT-factorization of A when a
% bound is appended to the constraint matrix. The corresponding QP problem has a reduced Hessian
% matrix redH and the cholesky factorization of redH is L_fr. The
% QT-factorization correspond to the the general constraint matrix and only
% the part corresponding to the free variables.

%   Call
%       [Q_fr, T_fr, L_fr] = null_space_update_fact_app_bound_FRFX(Q_fr, T_fr, L_fr)

%   Input parameters
%       Q_fr and T_fr  : is the QT-factorization of A, (A is the general
%                        constraint matrix and only the part
%                        corresponding to the free variables)
%       L_fr           : is the cholesky factorization of the reduced Hessian matrix of the corresponding QP problem.

%   Output parameters
%       Q_fr and T_fr  :is the QT-factorization of the general constraint
%                        matrix for the part corresponding to the free variables.
%       L_fr           :is the Cholesky factorization of the reduced Hessian matrix of the new QP problem.

%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date        : 08. february 2007.
%   Reference   : ---------------------
[n,m] = size(T_fr);
dns = n-m;
q = Q_fr(end,:)';
TL = zeros(n);
TL(1:dns,1:dns) = L_fr;
TL(:,dns+1:end) = T_fr;
for i =1:length(q)-1
    j = i+1;
    [s,c] = givens_rotation_matrix(q(i),q(j));

    temp = q(i)*c + q(j)*s;
    q(j) = -q(i)*s + q(j)*c;
    q(i) = temp;

    temp = Q_fr(:,i)*c + Q_fr(:,j)*s;
    Q_fr(:,j) =- Q_fr(:,i)*s + Q_fr(:,j)*c;
    Q_fr(:,i) = temp;

    temp = TL(i,:)*c + TL(j,:)*s;
    TL(j,:) = -TL(i,:)*s + TL(j,:)*c;
    TL(i,:) = temp;
```

```
end

Q_fr = Q_fr(1:end-1,1:end-1);
T_fr = TL(1:end-1,dns+1:end);
L_new = TL(1:dns-1,1:dns);
[nn mm] = size(L_new);
for i=1:1:nn
    j=i+1;
    [c,s] = givens_rotation_matrix(L_new(i,i),L_new(i,j));

    temp = L_new(:,i)*c - L_new(:,j)*s;
    L_new(:,j) = L_new(:,i)*s + L_new(:,j)*c;
    L_new(:,i) = temp;
end
L_fr = L_new(:,1:end-1);
```

null_space_update_fact_rem_bound_FRFX.m

```
function [Q_fr,T_fr,L_fr] = null_space_update_fact_rem_bound_FRFX(Q_fr, T_fr,
    L_fr, G)

% NULL_SPACE_UPDATE_FACT_REM_BOUND_FRFX updates the QT-factorization of the
    general constraint matrix(and only the part
% corresponding to the free variables) when a bound is removed.
% The corresponding QP problem has a reduced Hessian matrix redH and the
    cholesky factorization
% of redH is L_old.

%   Call
%         [Q_fr,T_fr,L_fr] = null_space_update_fact_rem_bound_FRFX(Q_fr, T_fr,
    L_fr, G)

%   Input parameters
%       Q_fr and T_fr   : is the QT-factorization of the constraint matrix (and
     only the part corresponding to the free variables).
%       L_fr            : is the cholesky factorization of the reduced Hessian
    matrix of the corresponding QP problem.
%       G               : is the Hessian matrix of the QP problem.

%   Output parameters
%       Q_fr and T_fr   :is the QT-factorization of the new general
%                        constraint matrix (and only the part
    corresponding to the free variables)
%       L_fr            :is the Cholesky factorization of the reduced Hessian
    matrix of the new QP problem.

%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor : John Bagterp Jørgensen, Assistant Professor & Per Grove
    Thomsen, Professor.
%   Date        : 08. february 2007.
%   Reference   : ---------------------

n = size(Q_fr,1);
m = size(T_fr,1)-1;
dns = n-m;
G_frfr = G(1:n,1:n);
Z_fr = Q_fr(:,1:dns);
Y_fr = Q_fr(:,dns+1:end);
[nn mm] = size(T_fr);
Y_fr = [Y_fr zeros(size(Y_fr,1),1); zeros(1,size(Y_fr,2)) 1];
for i=1:1:mm
    idx1 = nn-i;
    idx2 = idx1+1;
    [s,c] = givens_rotation_matrix(T_fr(idx1,i),T_fr(idx2,i));

    temp = T_fr(idx1,:)*c + T_fr(idx2,:)*s;
    T_fr(idx2,:) = -T_fr(idx1,:)*s +T_fr(idx2,:)*c;
    T_fr(idx1,:) = temp;

    temp = Y_fr(:,idx1)*c + Y_fr(:,idx2)*s;
    Y_fr(:,idx2) =- Y_fr(:,idx1)*s + Y_fr(:,idx2)*c;
    Y_fr(:,idx1) = temp;
end
T_fr = T_fr(2:end,:);
Z_fr = [Z_fr; zeros(1,size(Z_fr,2))];
Q_fr = [Z_fr Y_fr];
T_fr = [zeros(size(Z_fr,2)+1,size(T_fr,2));T_fr];
Z_fr_bar = Q_fr(:,1:n-m+1);
```

```
Z_fr = Z_fr_bar(1:end-1,1:end-1);
z = Z_fr_bar(1:end-1,end);
rho = Z_fr_bar(end,end);
h = G(1:n,n+1);
omega = G(n+1,n+1);
l = L_fr\((G_frfr*z+rho*h)'*Z_fr)';
delta = sqrt(z'*(G_frfr*z+2*rho*h)+omega*rho*rho-l'*l);
L_fr = [L_fr zeros(size(L_fr,1),1); l' delta];
```

D.5 Demos

QP_demo.m

```
function QP_demo(method,funtoplot)

% QP_DEMO Interactive demonstration of the primal active set and
% the dual active set methods.
%
% Call
%   QP_demo(method,funtoplot)
%
% Input parameter
%   method    : 'primal'  : Demonstrating the primal active set method.
%               'dual'    : Demonstrating the dual active set method.
%   funtoplot : 'objective'  : Plotting the objective function.
%               'lagrangian' : Plotting the Lagrangian function.
%
%   By         : Carsten V\"olcker, s961572.
%                Esben Lundsager Hansen, s022022.
%   Subject    : Numerical Methods for Sequential Quadratic Optimization.
%                M.Sc., IMM, DTU, DK-2800 Lyngby.
%   Supervisor : John Bagterp Jørgensen, Assistant Professor.
%                Per Grove Thomsen, Professor.
%   Date       : 07. June 2007.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check nargin/nargout                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
error(nargchk(2,2,nargin))
error(nargoutchk(0,0,nargout))
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check input                                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% check method...
if ~strcmp(method,'primal') & ~strcmp(method,'dual')
    error('Method must be ''primal'' or ''dual''.')
end
% check funtoplot...
ftp = 0; % plot objective function
if ~strcmp(funtoplot,'objective') & ~strcmp(funtoplot,'lagrangian')
    error('Funtoplot must be ''objective'' or ''lagrangian''.')
elseif strcmp(funtoplot,'lagrangian')
    ftp = 1; % plot lagrangian
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Setup and run demo                                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Setup demo...
G = [1 0;0 1];
g = [0 0]';
A = [0.5 1;
    0 1;
    2 1.75;
    3 -1;
    1 0];
b = [3 1 8.5 3 2.2]';
% Run demo...
if strcmp(method,'primal')
    primal_active_set_demo(G,g,A',b,ftp)
else
    dual_active_set_demo(G,g,A',b,ftp)
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auxilery function(s)                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function primal_active_set_demo(G,g,A,b,ftp)
% initialize...
[n,m] = size(A);
At = A';
w_non = 1:1:m;
w_act = [];
mu = zeros(m,1);
% initialize options...
tol = sqrt(eps);
it_max = 100;
% initialize counters and containers...
it = 0;
X = repmat(zeros(n,1),1,it_max);
% plot...
x = active_set_plot(G,At,g,b,[],mu,w_act,[-4 8;-4 8],[20 20 50 100 tol ftp]);
X(:,1) = x;
```

```matlab
% check feasibility of x...
i_b = find(At*x - b < -tol);
if ~isempty(i_b)
    disp(['Following constraint(s) violated, because A*x < b: '])
    fprintf(['\b',int2str(i_b'),'.\n'])
    error('Starting point for primal active set method is not feasible, run demo again.')
end
% iterate...
stop = 0;
while ~stop
    it = it + 1;
    if it >= it_max
        disp('No. or iterations steps exeeded.')
        stop = 2; % maximum no iterations exceeded
    end
    % call range/null space procedure...
    mu = zeros(m,1);
    [p,mu_act] = null_space_demo(G,A(:,w_act),G*x+g,zeros(length(w_act),1));
    mu(w_act) = mu_act;
    % plot...
    active_set_plot(G,At,g,b,X(:,1:it),mu,w_act,[-4 8;-4 8],[20 20 50 100 tol ftp]);
    disp('Press any key to continue...')
    pause
    % check if solution found...
    if norm(p) <= tol
        if mu >= -tol
            stop = 1; % solution found
            disp('Solution found by primal active set method, demo terminated.')
        else
            % compute index j of bound/constraint to be removed...
            [dummy,j] = min(mu);
            w_act = w_act(find(w_act ~= j)); % remove constraint j from active set
            w_non = [w_non j]; % append constraint j to nonactive setfunction
        end
    else
        % compute step length and index j of bound/constraint to be appended...
        alpha = 1;
        for app = w_non
            ap = At(app,:)*p; % At(app,:) = A(:,app)'
            if ap < -tol
                temp = (b(app) - At(app,:)*x)/ap;
                if -tol < temp & temp < alpha
                    alpha = temp; % smallest step length
                    j = app; % index j of bound to be appended
                end
            end
        end
        if alpha < 1
            % make constrained step...
            x = x + alpha*p;
            w_act = [w_act j]; % append constraint j to active set
            w_non = w_non(find(w_non ~= j)); % remove constraint j from nonactive set
        else
            % make full step...
            x = x + p;
        end
    end
    X(:,it+1) = x;
%     % plot...
%     if ~stop
%         %disp('Press any key to continue...')
%         %pause
%         active_set_plot(G,At,g,b,X(:,1:it+1),mu,w_act,[-4 8;-4 8],[20 20 50 100 tol ftp]);
%         disp('Press any key to continue...')
%         pause
%     end
end

function dual_active_set_demo(G,g,A,b,ftp)
% initialize...
[n,m] = size(A);
C = A;
w_non = 1:1:m;
w_act = [];
x = -G\g; x = x(:);
mu = zeros(m,1);
% initialize options...
tol = sqrt(eps);
max_itr = 100;
% initialize counters and containers...
it = 0;
```

```
it_draw = 1;
X = repmat(zeros(n,1),1,max_itr);
% plot...
active_set_plot(G,C',g,b,x,mu,w_act,[-4 8;-4 8],[20 20 50 100 tol ftp]);
disp('Press any key to continue...')
pause
X(:,1) = x;
% iterate...
stop = 0;
while ~stop
    c = constraints(G,C(:,w_non),g,b(w_non),x,mu);
    if c >= -tol;
        stop = 1;
        %disp('STOP: all inactive constraints >= 0')
        disp('Solution found by dual active set method, demo terminated.')
    else
        % we find the least negative value of c
        c_r = max(c(find(c < -sqrt(eps))));
        r = find(c == c_r);
        r = r(1);
    end
    it = it + 1;
    if it >= max_itr
        disp('No. or iterations steps exeeded (outer loop).')
        stop = 3; % maximum no iterations exceeded
    end
    % iterate...
    it2 = 0;
    stop2 = max(0,stop);
    while ~stop2
        it2 = it2 + 1;
        if it2 >= max_itr
            disp('No. or iterations steps exeeded (inner loop).')
            stop = 3;
            stop2 = stop;
        end
        % call range/null space procedure...
        [p,v] = null_space_demo(G,C(:,w_act),-C(:,r),zeros(length(w_act),1));
        if isempty(v)
            v = [];
        end
        arp = C(:,r)'*p;
        if abs(arp) <= tol % linear dependency
            if v >= 0 % solution does not exist
                disp('Problem is infeasible, demo terminated.')
                stop = 2;
                stop2 = stop;
            else
                t = inf;
                for k = 1:length(v)
                    if v(k) < 0
                        temp = -mu(w_act(k))/v(k);
                        if temp < t
                            t = temp;
                            rem = k;
                        end
                    end
                end
                mu(w_act) = mu(w_act) + t*v;
                mu(r) = mu(r) + t;
                w_act = w_act(find(w_act ~= w_act(rem)));
            end
        else
            % stepsize in dual space...
            t1 = inf;
            for k = 1:length(v)
                if v(k) < 0
                    temp = -mu(w_act(k))/v(k);
                    if temp < t1
                        t1 = temp;
                        rem = k;
                    end
                end
            end
            % stepsize in primal space...
            t2 = -constraints(G,C(:,r),g,b(r),x,mu)/arp;
            if t2 <= t1
                x = x + t2*p;
                mu(w_act) = mu(w_act) + t2*v;
                mu(r) = mu(r) + t2;
                w_act = [w_act r];
            else
                x = x + t1*p;
                mu(w_act) = mu(w_act) + t1*v;
                mu(r) = mu(r) + t1;
                w_act = w_act(find(w_act ~= w_act(rem)));
            end
```

```
        end
        c_r = constraints(G,C(:,r),g,b(r),x,mu);
        if c_r > -tol
            stop2 = 1; % leaves the inner while-loop but does not stop the
                 algorithm
        end
        it_draw = it_draw + 1;
        X(:,it_draw) = x;
        %plot...
        if ~ stop
            active_set_plot(G,C',g,b,X(:,1:it_draw),mu,w_act,[-4 8;-4 8],[20 20
                 50 100 tol ftp]);
            disp('Press any key to continue...')
            pause
        end
    end % while
end % while

function [x,mu] = null_space_demo(G,A,g,b)
% initialize...
[n m] = size(A);
% QR factorization of A so that A = [Y Z]*[R 0]'...
[Q,R] = qr(A); % matlab's implementation
Y = Q(:,1:m);
Z = Q(:,m+1:n);
R = R(1:m,:);
Zt = Z';
% Solve for the range space component py...
py = R'\b;
% Compute the reduced gradient...
gz = Zt*(G*(Y*py) + g);
% Compute the reduced Hessian and compute its Cholesky factorization...
Gz = Zt*G*Z;
L = chol(Gz)';
% Solve for the null space component pz...
pz = L\-gz;
pz = L'\pz;
% Compute the solution...
x = Y*py + Z*pz;
% Compute the Lagrange multipliers...
mu = R\(Y'*(G*x + g));

function f = objective(G,A,g,b,x,mu)
f = x'*G*x + g'*x;

function c = constraints(G,A,g,b,x,mu)
c = A'*x - b;

function l = lagrangian(G,A,g,b,x,mu)
L = objective(G,A,g,b,x,mu) - mu'*constraints(G,A,g,b,x,mu);
```

active_set_plot.m

```
function [x,w_act] = active_set_plot(G,A,g,b,x,mu,w_act,D,opts)

% ACTIVE_SET_PLOT Plotting the objective or the Lagrangian function and the
% constraints with feasible regions. The constraints must on the form
% A*x >= b. Can only plot for three dimensions.
%
% Call
%    active_set_plot(G, A, g, b, x, mu, wa, D)
%    active_set_plot(G, A, g, b, x, mu, wa, D, opts)
%    [x,wa] = active_set_plot( ... )
%
% Input parameters
%   G      : The Hessian of the obejctive function.
%   A      : The constraint matrix of size mx2, where m is the number of
%            constraints.
%   g      : Coefficients of linear term in objective function.
%   b      : Righthandside of constraints.
%   x      : Starting point. If x is a matrix of size 2xn, n = 1,2,3,...,
%            then the iteration path is plottet. If x is empty, the user will
%            be asked to enter a starting point.
%   mu     : The Lagrangian multipliers. If mu is empty, all multipliers will
%            be set to zero.
%   wa     : Working set listing the active constraints. If wa is empty, then
%            a constraint will be found as active, if x is within a range of
%            opts(5) to that constraint.
%   D      : Domain to be plottet, given as [x1(1) x1(2); x2(1) x2(2)].
%   opts   : Vector with six elements.
```

```
%            opts(1:2) : Number of grid points in the first and second
%                        direction.
%            opts(3)   : Number of contour levels.
%            opts(4)   : Number of linearly spaced points used for plotting
%                        the constraints.
%            opts(5)   : A constraint will be found as active, if x is
%                        within a range of opts(5) to that constraint.
%            opts(6)   : 0: Plotting the contours of the objective function.
%                        1: Plotting the contours of the Lagrangian function.
%   If opts not, then the default opts = [20 20 50 100 sqrt(eps) 0].
%
% Output parameters
%   x : Same as input x. If input x is empty, then the starting point
%          entered by the user.
%   w : Same as input w_act. If input w_act is empty, then the list of
%          active constraint found upon the input/entered starting point.
%
% By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
% In course   : Numerical Methods for Sequential Quadratic Optimization,
%               Master Thesis, IMM, DTU, DK-2800 Lyngby.
% Supervisor  : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen,
%               Professor.
% Date        : 28th January 2007.

% checking input...
error(nargchk(8,9,nargin))
A = A';
[n,m] = size(A);
if isempty(mu)
    mu = zeros(m,1);
end
[u,v] = size(D);
if u ~= 2 | v ~= 2
    error('The domain must be a matrix of size 2x2.')
end
if nargin > 8
    [u,v] = size(opts(:));
    if u ~= 6 | v ~= 1
        error('Opts must be a vector of length 6.')
    end
end

% default opts...
if nargin < 9 | isempty(opts)
    opts = [20 20 20 20 sqrt(eps) 0];%[20 20 50 100 sqrt(eps) 0];
end

% function to plot...
fun = @objective;
if opts(6)
    fun = @lagrangian;
end

% internal parameters...
fsize = 12; % font size

% plot the contours of the objective or the Lagrangian function...
figure(1), clf
contplot(fun,G,A,g,b,mu,D,opts)
xlabel('x_1','FontSize',fsize)
ylabel('x_2','FontSize',fsize)
hold on

% plot the constraints...
if nargout & isempty(x)
    constplot(@constraints,G,A,g,b,mu,D,w_act,m,opts,fsize)
    %title('x = ( , ), f(x) = , W_a = [], \mu = []','FontSize',fsize)
    % ask user to enter starting point...
    while isempty(x)
        disp('Left click on plot to select starting point or press any key to
             enter starting point in console.')
        [u,v,but] = ginput(1);
        if but == 1
            x = [u v];
        else
            while isempty(x) | length(x) ~= 2 | sum(isnan(x)) | sum(isinf(x)) |
                    sum(~isreal(x)) | ischar(x)
                x = input('Enter starting point [x1 x2]: ');
            end
        end
    end
    x = x(:);
    figure(1)
    % find active constraints...
    if nargout > 1
        w_act = find(abs(A(2,:)'*x(2) + feval(@constraints,G,A(1,:),g,b,x(1),mu
             )) <= opts(5))'; % A(2)*x2 + (A(1)*x1 - b) <= eps
```

```
        if w_act
            constplot(@constraints,G,A,g,b,mu,D,w_act,m,opts,fsize)
        end
        title(['x = (',num2str(x(1,end),2),', ',num2str(x(2,end),2),'), f(x) = ',num2str(objective(G,A,g,b,x(:,end),mu),2),', W_a = [',int2str(w_act),'], \mu = [',num2str(mu',2),']'],'FontSize',fsize)
    end
else
    %if isempty(w_act)
    %    w_act = find(abs(A(2,:)'*x(2) + feval(@constraints,G,A(1,:),g,b,x(1),mu)) <= opts(5))'; % A(2)*x2 + (A(1)*x1 - b) <= eps
    %end
    constplot(@constraints,G,A,g,b,mu,D,w_act,m,opts,fsize)
    title(['x = (',num2str(x(1,end),2),', ',num2str(x(2,end),2),'), f(x) = ',num2str(objective(G,A,g,b,x(:,end),mu),2),', W_a = [',int2str(w_act),'], \mu = [',num2str(mu',2),']'],'FontSize',fsize)
end

% plot the path...
pathplot(x)
hold off

function contplot(fun,G,A,g,b,mu,D,opts)
[X1,X2] = meshgrid(linspace(D(1,1),D(1,2),opts(1)),linspace(D(2,1),D(2,2),opts(2)));
F = zeros(opts(1:2));
for i = 1:opts(1)
    for j = 1:opts(2)
        F(i,j) = norm(feval(fun,G,A,g,b,[X1(i,j);X2(i,j)],mu),2);
    end
end
contour(X1,X2,F,opts(3))

function constplot(fun,G,A,g,b,mu,D,w_act,m,opts,fsize)
fcolor = [.4 .4 .4]; falpha = .4; % color and alpha values of faces marking unfeasable region
bcolor = [.8 .8 .8]; % background color of constraint numbering
x1 = linspace(D(1,1),D(1,2),opts(4));
x2 = linspace(D(2,1),D(2,2),opts(4));
C = zeros(m,opts(4));
for j = 1:opts(4)
    for i = 1:m
        if A(2,i) % if A(2) ~= 0
            C(i,j) = -feval(fun,G,A(1,i),g,b(i),x1(j),mu)/A(2,i); % x2 = -(A(1)*x1 - b)/A(2)
        else
            C(i,j) = b(i)/A(1,i); % A(2) = 0 => x1 = b/A(1), must be plottet reversely as (C(i,:),x2)
        end
    end
end
for i = 1:m
    if any(i == w_act)
        lwidth = 1; color = [1 0 0]; % linewidth and color of active constraints
    else
        lwidth = 1; color = [0 0 0]; % linewidth and color of inactive constraints
    end
    if A(2,i) % if A(2) ~= 0
        if A(2,i) > 0 % if A(2) > 0
            fill([D(1,1) D(1,2) D(1,2) D(1,1)],[C(i,1) C(i,end) min(D(2,1),C(i,end)) min(D(2,1),C(i,1))],fcolor,'FaceAlpha',falpha)
        else
            fill([D(1,1) D(1,2) D(1,2) D(1,1)],[C(i,1) C(i,end) max(D(2,2),C(i,end)) max(D(2,2),C(i,1))],fcolor,'FaceAlpha',falpha)
        end
        plot(x1,C(i,:),'-','LineWidth',lwidth,'Color',color)
        if C(i,1) < D(1,1)% | C(i,1) < D(2,1)
            text(-feval(fun,G,A(2,i),g,b(i),D(2,1),mu)/A(1,i),D(2,1),int2str(i),'Color','k','EdgeColor',color,'BackgroundColor',bcolor,'FontSize',fsize) % x1 = -(A(2)*x2 - b)/A(1)
        else
            if C(i,1) > D(2,2)
                text(-feval(fun,G,A(2,i),g,b(i),D(2,1),mu)/A(1,i),D(2,1),int2str(i),'Color','k','EdgeColor',color,'BackgroundColor',bcolor,'FontSize',fsize) % x1 = -(A(2)*x2 - b)/A(1)
            else
                text(D(1,1),C(i,1),int2str(i),'Color','k','EdgeColor',color,'BackgroundColor',bcolor,'FontSize',fsize)
            end
        end
    else
        if A(1,i) > 0 % if A(1) > 0
            fill([D(1,1) C(i,1) C(i,end) D(1,1)],[D(2,1) D(2,1) D(2,2) D(2,2)],fcolor,'FaceAlpha',falpha)
        else
```

```
            fill([C(i,1) D(1,2) D(1,2) C(i,end)],[D(2,1) D(2,1) D(2,2) D(2,2)],fcolor,'FaceAlpha',falpha)
        end
        plot(C(i,:),x2,'-','LineWidth',lwidth,'Color',color)
        text(C(i,1),D(2,1),int2str(i),'Color','k','EdgeColor',color,'BackgroundColor',bcolor,'FontSize',fsize)
    end
end

function pathplot(x)
lwidth = 2; msize = 6;
plot(x(1,1),x(2,1),'ob','LineWidth',lwidth,'Markersize',msize) % starting position
plot(x(1,:),x(2,:),'LineWidth',lwidth) % path
plot(x(1,end),x(2,end),'og','LineWidth',lwidth,'Markersize',msize) % current position

function f = objective(G,A,g,b,x,mu)
f = 0.5*x'*G*x + g'*x;

function c = constraints(G,A,g,b,x,mu)
c = A'*x - b;

function l = lagrangian(G,A,g,b,x,mu)
l = objective(G,A,g,b,x,mu) - mu'*constraints(G,A,g,b,x,mu);
```

quad_tank_demo.m

```
function quad_tank_demo(t,N,r,F,dF,gam,w,pd)

% QUAD_TANK_DEMO Demonstration of the quadruple tank process. The water
% levels in tank 1 and 2 are controlled according to the set points. The
% heights of all four tanks are 50 cm. The workspace is saved as
% 'quadruple_tank_process.mat' in current directory, so it is possible to
% run the animation again by calling quad_tank_animate without recomputing
% the setup.
% NOTE: A new call of quad_tank_demo will overwrite the saved workspace
%       'quadruple_tank_process.mat'. The file must be deleted manually.
%
% Call
%   quad_tank_demo(t,N,r,F,dF,gam,pd)
%
% Input parameters
%   t   : [min] Simulation time of tank process. 1 <= t <= 30. Default is
%         5. The time is plottet as seconds. The last discrete point is not
%         animated/plottet.
%   N   : Discretization of t. 5 <= N <= 100, must be an integer. Default
%         is 10. Number of variables is 6*N and number of constraints is
%         24*N.
%   r   : [cm] Set points of tank 1 and 2. 0 <= r(i) <= 50. Default is
%         [30 30].
%   F   : [l/min] Max flow rates of pump 1 and 2. 0 <= F(i) <= 1000.
%         Default is [500 500].
%   dF  : [l/min^2] Min/max change in flow rates of pump 1 and 2. -100 <=point
%         dF(i) <= 100. Default is [pump1 pump2] = [-50 50 -50 50].
%   gam : Fraction of flow from pump 1 and 2 going directly to tank 1 and
%         2. 0 <= gam(i) <= 1. Default is [0.45 0.40].
%   w   : Setting priority of controlling water level in tank 1 and 2
%         relative to one another. 1 <= w(i) <= 1000. Default is [1 1].
%   pd  : 1: Using primal active set method, 2: Using dual active set
%         method. Default is 2.
%     If input parameters are empty, then default values are used.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check nargin/nargout                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
error(nargchk(7,8,nargin))
error(nargoutchk(0,0,nargout))
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check input                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check t...
if isempty(t)
    t = 5*60; % t = min*sec
else
    t = check_input(t,1,30,1)*60; % t = min*sec
end
% Check N...
if isempty(N)
    N = 10;
```

```
else
    if mod(N,1)
        error('N must be an integer.')
    end
    N = check_input(N,5,100,1);
end
% Check r...
if isempty(r)
    r = [30 30];
else
    r = check_input(r,0,50,2);
end
% Check F...
if isempty(F)
    F = [500 500];
else
    F = check_input(F,0,1000,2);
end
% Check dF...
if isempty(dF)
    dF = [-50 50 -50 50];
else
    dF = check_input(dF,-100,100,4);
end
% Check gam...
if isempty(gam)
    gam = [0.45 0.4];
else
    gam = check_input(gam,0,1,2);
end
% Check w...
if isempty(w)
    w = [1 1];
else
    w = check_input(w,1,1000,2);
end
% Check pd...
if nargin < 8 | isempty(pd)
    pd = 2;
else
    if mod(pd,1)
        error('pd must be an integer.')
    end
    pd = check_input(pd,1,2,1);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Startup info                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if N >= 30
    cont = 'do';
    while ~strcmp(lower(cont),'y') & ~strcmp(lower(cont),'n')
        cont = input('N >= 30, so computational time will be several minutes, do you want to continue? y/n [y]: ','s');
        if isempty(cont)
            cont = 'y';
            fprintf('\b')
            disp('y')
        end
        %disp(' ')
    end
    if cont == 'n'
        disp('Simulation terminated by user.')
        return
    end
end
disp(['N = ',int2str(N),', so number of variables is ',int2str(6*N),' and number of constraints is ',int2str(24*N),'.'])
disp(['t = ',num2str(t,2),', gam = [',num2str(gam,2),'], w = [',num2str(w,2),'].'])
disp(['r = [',num2str(r),'], F = [',num2str(F,2),'], dF = [',num2str(dF,2),'].'])
disp('Computing simulation, please wait...')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Setup demo                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% UI                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% physics...
g = 5; % gravity is small due to linearized system
% time span and number of sampling points...
tspan = [0 t];%360];
% weights matrices...
Q = [w(1) 0; 0 w(2)]; % weight matrix, used in Q-norm, setting priority of h1 and h2 relative to each other
Hw = 1e6; % weighing h1 and h2 (= Hw) in relation to h3, h4, u1 and u2 (= 1)
% pump 1...
```

```
Fmin1 = 0; Fmax1 = F(1); % minmax flows
dFmin1 = dF(1); dFmax1 = dF(2); % minmax rate of change in flow
F10 = 0; % initial value
% pump 2...
Fmin2 = 0; Fmax2 = F(2); % minmax flows
dFmin2 = dF(3); dFmax2 = dF(4); % minmax rate of change in flow
F20 = 0; % initial value
% valve 1...
gam1 = gam(1);
% valve 2...
gam2 =  gam(2);
% tank 1...
r1 = r(1); % set point
hmin1 = 0; hmax1 = 50; % minmax heights
h10 = 0; % initial value
% tank 2...
r2 = r(2); % set point
hmin2 = 0; hmax2 = 50; % minmax heights
h20 = 0; % initial value
% tank 3...
hmin3 = 0; hmax3 = 50; % minmax heights
h30 = 0; % initial value
% tank 4...
hmin4 = 0; hmax4 = 50; % minmax heights
h40 = 0; % initial value

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initiate variables                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% pumps...
umin = [Fmin1 Fmin2]';
umax = [Fmax1 Fmax2]';
dumin = [dFmin1 dFmin2]';
dumax = [dFmax1 dFmax2]';
% valves...
gam = [gam1 gam2];
% tanks...
bmin = [hmin1 hmin2 hmin3 hmin4]';
bmax = [hmax1 hmax2 hmax3 hmax4]';
% set points...
r = [r1 r2]';
% initial values...
x0 = [h10 h20 h30 h40]';
%u0 = [F10 F20]';
u0minus1 = [F10 F20]';
Q = Hw*Q; % weight matrix, used in Q-norm, setting priority of h1 and h2 relative to each other
dt = (tspan(2) - tspan(1))/N;

nx = length(x0);
nu = length(u0minus1);

a1 = 1.2272;
a2 = 1.2272;
a3 = 1.2272;
a4 = 1.2272;
A1 = 380.1327;
A2 = 380.1327;
A3 = 380.1327;
A4 = 380.1327;

Ac = 2*g*[-a1/A1 0 a3/A1 0;
     0 -a2/A2 0 a4/A2;
     0 0 -a3/A3 0;
     0 0 0 -a4/A4];

Bc = [gam1/A1 0;
     0 gam2/A2;
     0 (1 - gam2)/A3;
     (1 - gam1)/A4 0];

Cc = [1 0 0 0;
     0 1 0 0];

% %###############################################################
% % build the object function Hessian and gradient:
% %###############################################################

Qx = dt*Cc'*Q*Cc;                       % should this have non-zero elements in the diogonal??
Qx = add2mat(Qx,eye(2),3,3,'rep');     % => ASSURES THAT HESSIAN IS POSITIVE DEFINITE
Qu = eye(nu);                          % only purpose is to make dimensions fit and to remain positive definite
%Qu = zeros(nu);
qx = -dt*Cc'*Q*r;                       % qk in text
```

```
qu = zeros(2,1);                                % only purpose is to make dimensions fit

H = zeros(N*(nx+nu));                   % Hessian
g = zeros(N*(nx+nu),1)   ;              % gradient
for i = 1:N
    %     if i > floor(N/2)
    %         r(2) = 20
    %         qx = -dt*Cc'*Q*r
    %     end
    j = 1+(i-1)*(nu+nx);
    H = add2mat(H,Qu,j,j,'rep');
    g = add2mat(g,qu,j,1,'rep');
    H = add2mat(H,Qx,j+nu,j+nu,'rep');
    g = add2mat(g,qx,j+nu,1,'rep');
end

% ###########################################################################
% % Build A_c (general constraint matrix), upper and lower bounds for
% % general constraints (bl and bu) and upper and lower bounds for
% % variables (u and l)
% % #########################################################################

Ix = eye(nx);
Iu = eye(nu);
A = Ix + dt*Ac;
B = dt*Bc;
Ax0 = A*x0;
zerox = zeros(nx,1);
zerou = zeros(nu,1);

n   = N*(nx+nu);          % number of variables
m   = N*(nx+nu);          % number of general constraints
A_c = zeros(m,n);         % new A matrix (carsten) (general constraitns are rows => we will transpose it later)
l   = zeros(n,1);         % lower bounds for variables
u   = zeros(n,1);         % upper bounds for variables
bl  = zeros(m,1);         % lower bounds for general constraints
bu  = zeros(m,1);         % upper bounds for general constraints

row = 1;
col = 1;
A_c = add2mat(A_c,B,row,col,'rep');
A_c = add2mat(A_c,-Ix,row,col+nu,'rep');
bl = add2mat(bl,-Ax0,row,1,'rep');
bu = add2mat(bu, Ax0,row,1,'rep');
for i =1:N-1
    row = 1+i*nx;              % start row for new k
    col = 3+(i-1)*(nx+nu);     % start column for new k
    A_c = add2mat(A_c,A,row,col,'rep');
    A_c = add2mat(A_c,B,row,col+nx,'rep');
    A_c = add2mat(A_c,-Ix,row,col+nx+nu,'rep');
    bl = add2mat(bl, zerox,row,1,'rep');
    bu = add2mat(bu, zerox,row,1,'rep');
end

row = N*nx+1;
A_c = add2mat(A_c,Iu,row,1,'rep');
bl = add2mat(bl,dumin+u0minus1,row,1,'rep');
bu = add2mat(bu,dumax-u0minus1,row,1,'rep');
for i =1:N-1
    row = row+nu;
    col = 1+(i)*(nu+nx);

    A_c = add2mat(A_c, Iu,row,col,'rep');
    A_c = add2mat(A_c,-Iu,row,col-(nx+nu),'rep');
    bl = add2mat(bl,dumin,row,1,'rep');
    bu = add2mat(bu,dumax,row,1,'rep');
end

for i =0:N-1
    row = 1+i*(nx+nu);
    l = add2mat(l,umin,row,1,'rep');
    u = add2mat(u,umax,row,1,'rep');
    l = add2mat(l,bmin,row+nu,1,'rep');
    u = add2mat(u,bmax,row+nu,1,'rep');
end

if pd == 1
    x = LP_solver(l,u,A_c,bl,bu);
else
    x = [];
end
[x,info] = QP_solver(H,g,l,u,A_c,bl,bu,x);
disp('Performance information of active set method:')
info

```

```
% ####################################################################
% plots of quad-tank process
% ####################################################################
output = x;
% making t...
t = tspan(1):dt:tspan(2);
% making h, F, df...
u0 = output(1:2);
x1 = output(3:6);
nul0 = B*u0-Ix*x1+Ax0;
nul0 = nul0'*nul0;
x_k = x1;
h = x_k(1:2);
heights(:,1)=x0;
heights(:,2)=x_k;
flow(:,1) = u0minus1;
flow(:,2) = u0;
for k = 1:N-1
    ks = 3+(k-1)*(nx+nu);
    u_k = output(ks+4:ks+5);
    x_k_plus = output(ks+6:ks+9);

    nul_k = A*x_k+B*u_k-Ix*x_k_plus;
    nul_k = nul_k'*nul_k;

    x_k = x_k_plus;
    %    h = x_k(1:2)
    heights(:,k+2)=x_k;
    flow(:,k+2) = u_k;
end
dF = [diff(flow(1,:)); diff(flow(2,:))];
% plot...
fsize = 10;
figure(1), clf
subplot(4,2,1)
plot(t,heights(1,:),'-o')
hold on
plot(t,r(1)*ones(1,N+1),'r')
plot(t,hmax1*ones(1,N+1),'k')
xlabel('t [s]','FontSize',fsize), ylabel('h_1 [cm]','FontSize',fsize)%, legend('h_1','r_1')
axis([tspan(1) tspan(2) 0 50])
subplot(4,2,2)
plot(t,heights(2,:),'-o')
hold on
plot(t,r(2)*ones(1,N+1),'r')
plot(t,hmax2*ones(1,N+1),'k')
xlabel('t [s]','FontSize',fsize), ylabel('h_2 [cm]','FontSize',fsize)%, legend('h_2','r_2')
axis([tspan(1) tspan(2) 0 50])
subplot(4,2,3)
plot(t,heights(3,:),'-o')
xlabel('t [s]','FontSize',fsize), ylabel('h_3 [cm]','FontSize',fsize)
hold on
plot(t,hmax3*ones(1,N+1),'k')
axis([tspan(1) tspan(2) 0 50])
subplot(4,2,4)
plot(t,heights(4,:),'-o')
xlabel('t [s]','FontSize',fsize), ylabel('h_4 [cm]','FontSize',fsize)
hold on
plot(t,hmax4*ones(1,N+1),'k')
axis([tspan(1) tspan(2) 0 50])
subplot(4,2,5)
plot(t(1:end)-dt,flow(1,:),'-o')
xlabel('t [s]','FontSize',fsize), ylabel('F_1 [cm^3/s]','FontSize',fsize)
axis([t(1)-dt tspan(2) 0 Fmax1])
subplot(4,2,6)
plot(t(1:end)-dt,flow(2,:),'-o')
xlabel('t [s]','FontSize',fsize), ylabel('F_2 [cm^3/s]','FontSize',fsize)
%stairs(flow(2,:))
axis([t(1)-dt tspan(2) 0 Fmax2])
subplot(4,2,7)
plot(t(1:end-1)-dt,dF(1,:),'-o')
xlabel('t [s]','FontSize',fsize), ylabel('\Delta F_1 [cm^3/s^2]','FontSize',fsize)
axis([t(1)-dt tspan(2) dFmin1 dFmax1])
subplot(4,2,8)
plot(t(1:end-1)-dt,dF(2,:),'-o')
xlabel('t [s]','FontSize',fsize), ylabel('\Delta F_2 [cm^3/s^2]','FontSize',fsize)
axis([t(1)-dt tspan(2) dFmin2 dFmax2])
hold off
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Animate demo                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
save quadruple_tank_process
quad_tank_animate
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auxilery function(s)                                                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function v = check_input(v,l,u,n)
v = v(:)';
m = length(v);
if m ~= n
    error([num2str(inputname(1)),' must be a vector of length ',int2str(n),'.'])
end
for i = 1:n
    if ischar(v(i)) | ~isreal(v(i)) | isinf(v(i)) | isnan(v(i)) | v(i) < l | u < v(i)
        error([num2str(inputname(1)),'(',int2str(i),') must be in range ',num2str(l),' <= value <= ',num2str(u),'.'])
    end
end
```

quad_tank_animate.m

```
% ##################################################################
% animation of quad-tank process
% ##################################################################
load('quadruple_tank_process')
output = x;
% making t...
t = tspan(1):dt:tspan(2);
% making h, F, df...
u0 = output(1:2);
x1 = output(3:6);
nul0 = B*u0-Ix*x1+Ax0;
nul0 = nul0'*nul0;
x_k = x1;
h = x_k(1:2);
heights(:,1)=x0;
heights(:,2)=x_k;
flow(:,1) = u0minus1;
flow(:,2) = u0;
for k = 1:N-1
    ks = 3+(k-1)*(nx+nu);
    u_k = output(ks+4:ks+5);
    x_k_plus = output(ks+6:ks+9);

    nul_k = A*x_k+B*u_k-Ix*x_k_plus;
    nul_k = nul_k'*nul_k;

    x_k = x_k_plus;
    %    h = x_k(1:2)
    heights(:,k+2)=x_k;
    flow(:,k+2) = u_k;
end
dF = [diff(flow(1,:)); diff(flow(2,:))];
for i = 1:length(t)-1
    figure(2)
    quad_tank_plot(t(i),heights(:,i),bmax,r,flow(:,i),dF(:,i),gam)
    M(i) = getframe;
end

% make movie for presentation...
reply = input('Do you want to make 4 tank demo movie.avi file? y/n [y]: ', 's');
if isempty(reply) | reply == 'y'
    fps = input('Specify fps in avi file? [15]: ');
    if isempty(fps)
        fps = 15;
    end
    disp('Making avi file, please wait...')
    movie2avi(M,'4_tank_demo_movie.avi','fps',fps)
    disp('Finished making avi file...')
end
```

SQP_demo.m

```
function SQP_demo

% SQP_DEMO Interactive demonstration of the SQP method. The example problem
% is the following nonlinear program:
%          min   f(x) = x1^4 + x2^4
%          s.t.  x2 >=  x1^2 -  x1 + 1
%                x2 >=  x1^2 - 4x1 + 6
%                x2 <= -x1^2 + 3x1 + 2
%
% Call
%    SQP_demo()
%
%    By          : Carsten V\"olcker, s961572.
%                  Esben Lundsager Hansen, s022022.
%    Subject     : Numerical Methods for Sequential Quadratic Optimization.
%                  M.Sc., IMM, DTU, DK-2800 Lyngby.
%    Supervisor  : John Bagterp Jørgensen, Assistant Professor.
%                  Per Grove Thomsen, Professor.
%    Date        : 07. June 2007.

close all
it_max = 1000;
method = 1;
tol = 1e-8;opts = [tol it_max method];

pi0 = [0 0 0]';  % because we have three nonlinear constraints.
plot_scene(@costfun);

disp('Left click on plot to select starting point or press any key enter starting point in console.')
[u,v,but] = ginput(1);
if but == 1
    x0 = [u v];
else
    while isempty(x) | length(x) ~= 2 | sum(isnan(x)) | sum(isinf(x)) | sum(~isreal(x)) | ischar(x)
        x0 = input('Enter starting point [x1 x2]: ');
    end
end
x0 = x0';
pathplot(x0)
fsize = 12;
fsize_small = 10;
f0 = costfun(x0);
g0 = modfun(x0);
c  = costsens(x0);
A  = modsens(x0);
W  = eye(length(x0));
w_non = (1:1:length(g0));
plotNewtonStep = 1;
stop = 0;
tol = opts(1);
max_itr = opts(2);
itr = 0;
while ~stop
    disp('Press any key to continue...')
    pause
    X(:,itr+1) = x0;
    if plotNewtonStep
        W = hessian(x0,pi0);
    else
        hold on
        pathplot(X)
    end
    itr = itr+1;
    if(itr > max_itr)
        stop = 1;
    end

    [delta_x, mu,dummy] = dual_active_set_method(W,c,A,-g0,w_non,[],opts,0);

    if (abs(c'*delta_x) + abs(mu'*g0)) < tol
        disp('solution has been found')
        stop = 1;
    else

        if itr == 1
            sigma = abs(mu);
        else
            for i=1:length(mu)
                sigma(i) = max(abs(mu(i)), 0.5*(sigma(i)+abs(mu(i))));
            end
        end

        [alpha,x,f,g] = line_search_algorithm(@modfun,@costfun,f0,g0,c,x0,delta_x,sigma,1e-4);
```

```matlab

        pii = pi0 + alpha*(mu-pi0);

        % here the newton step is plotted
        if plotNewtonStep
            t_span = linspace(-1,3,100);
            for i =1:length(t_span)
                x_hat = x0+t_span(i)*delta_x;
                pi_hat = pi0+t_span(i)*(mu-pi0);
                nabla_fx = costsens(x_hat);
                nabla_hx = modsens(x_hat);
                y_val(:,i) = nabla_fx - nabla_hx*pi_hat;
            end

            subplot(1,3,1)
            hold off
            plot_scene(@costfun);
            X_temp = X;
            X_temp(:,itr+1) = x;%0+delta_x;
            x_fin = x;%x0+alpha*delta_x;
            pathplot(X_temp)
            title({['x_{old} = (',num2str(x0(1)),',',num2str(x0(2)),')^T'];['x_{new} = (',num2str(x_fin(1)),',',num2str(x_fin(2)),')^T']},'FontSize',fsize_small);
            xlabel('x_1' ,'FontSize',fsize), ylabel('x_2','FontSize',fsize)

            subplot(1,3,2)
            hold off
            plot(t_span,y_val(1,:));

            nabla_fx = costsens(x0);
            nabla_hx = modsens(x0);
            startPos = nabla_fx - nabla_hx*pi0;
            startPos_y = startPos(1);
            startPos_x = 0;

            endPos_x = 1;
            endPos_y = 0;

            hold on
            plot([startPos_x endPos_x],[startPos_y endPos_y],'LineWidth',2) % path
            plot([startPos_x alpha],[startPos_y (1-alpha)*startPos_y],'LineWidth',2,'color','r') % path
            plot([t_span(1) t_span(end)],[0 0],'--') % y=0

            pi_fin = pi0+alpha*(mu-pi0);
            nabla_fx_fin = costsens(x_fin);
            nabla_hx_fin = modsens(x_fin);
            endvalue = nabla_fx_fin - nabla_hx_fin*pi_fin;

            title({['F(x_1)_{old} = ',num2str(startPos_y)];['F(x_1)_{new} = ',num2str(endvalue(1))]},'FontSize',fsize_small);
            xlabel('\alpha' ,'FontSize',fsize), ylabel('F_1','FontSize',fsize)

            subplot(1,3,3)
            hold off
            plot(t_span,y_val(2,:));
            startPos_y = startPos(2);
            hold on
            plot([startPos_x endPos_x],[startPos_y endPos_y],'LineWidth',2) % path
            plot([startPos_x alpha],[startPos_y (1-alpha)*startPos_y],'LineWidth',2,'color','r') % path
            plot([t_span(1) t_span(end)],[0 0],'--') % y=0
            title({['F(x_2)_{old} = ',num2str(startPos_y)];['F(x_2)_{new} = ',num2str(endvalue(2))]},'FontSize',fsize_small);
            xlabel('\alpha' ,'FontSize',fsize), ylabel('F_2','FontSize',fsize)
        end
        nabla_L0 = c-A*pii;
        c  = costsens(x);
        A  = modsens(x);
        nabla_L = c-A*pii;
        s = x - x0;
        y = nabla_L - nabla_L0;
        sy = s'*y;
        sWs = s'*W*s;
        if(sy >= 0.2*sWs)
            theta = 1;
        else
            theta = (0.8*sWs)/(sWs-sy);
        end
        Ws = W*s;
        sW = s'*W;
        r = theta*y+(1-theta)*Ws;
        W = W-(Ws*sW)/sWs+(r*r')/(s'*r);
        x0 = x;
```

```
        pi0 = pii;
        f0 = f;
        g0 = g;
    end
end

function pathplot(x)
lwidth = 2; msize = 6; fsize = 12;
plot(x(1,1),x(2,1),'ob','LineWidth',lwidth,'Markersize',msize) % starting position
plot(x(1,:),x(2,:),'LineWidth',lwidth) % path
plot(x(1,:),x(2,:),'ob','LineWidth',lwidth,'Markersize',msize) % path
plot(x(1,end),x(2,end),'og','LineWidth',lwidth,'Markersize',msize) % current position
title(['x = (',num2str(x(1,end)),' , ',num2str(x(2,end)),')'],'FontSize',fsize)



function fx = costfun(x)
% The function to be minimized
fx = x(1)*x(1)*x(1)*x(1)+x(2)*x(2)*x(2)*x(2); % : cost = (X1.^4 + X2.^4);

function dx = costsens(x)
% The gradient of the cost function
dx =[4*x(1)*x(1)*x(1); 4*x(2)*x(2)*x(2)]; % : gradient of (X1.^4 + X2.^4);


function fx = modfun(x)
% The constraints
c1 = -x(1)^2+x(1)+x(2)-1;       % x2 >= x1.^2 - x1 + 1
c2 = -x(1)^2+4*x(1)+x(2)-6;     % x2 >= x1^2 - 4x1 + 6
c3 = -x(1).^2+3*x(1)-x(2)+2;    % x2 <= -x1^2 + 3x1 + 2
fx = [c1 c2 c3]';

function dfx = modsens(x)
% gradient of the constraints

dc1 = [-2*x(1)+1 1]';   % x2 >= x1.^2 - x1 + 1
dc2 = [-2*x(1)+4 1]';   % x2 >= x1^2 - 4x1 + 6
dc3 = [-2*x(1)+3 -1]';  % x2 <= -x1^2 + 3x1 + 2
dfx = [dc1 dc2 dc3];

function H = hessian(x,mu)

HessCtr1 = [-2 0; 0 0];            % x2 >= x1.^2 - x1 + 1
HessCtr2 = [-2 0; 0 0];            % x2 >= x1^2 - 4x1 + 6
HessCtr3 = [-2 0; 0 0];             % x2 <= -x1^2 + 3x1 + 2

HessCost = [12*x(1)*x(1) 0;
    0 12*x(2)*x(2)];      % : cost = (X1.^4 + X2.^4);

H = HessCost-(mu(1)*HessCtr1)-(mu(2)*HessCtr2)-(mu(3)*HessCtr3);

function plot_scene(costfun, varargin)
fcolor = [.4 .4 .4]; falpha = .4; % color and alpha values of faces marking unfeasable region
plot_left    = -5;
plot_right   =  5;
plot_buttom  = -5;
plot_top     =  5;
plotdetails = 30;
linspace_details = 100;
contours = 10;
ctr1 = 1;
ctr2 = 1;
ctr3 = 0;
ctr4 = 0;
ctr5 = 1;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% constraints defined for z=0
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
x_ctr = linspace(plot_left,plot_right,linspace_details);
y_ctr1 = x_ctr.^2-x_ctr+1;          ctr1_geq = 1;    % x2 >= x1.^2 - x1 + 1
y_ctr2 = x_ctr.^2 - 4*x_ctr + 6;    ctr2_geq = 1;    % x2 >= x1^2 - 4x1 + 6
y_ctr3 = sin(x_ctr) + 3;            ctr3_geq = 0;    % x2 <= sin(x1) + 3
y_ctr4 = cos(x_ctr) + 2;            ctr4_geq = 1;    % x2 >= cos(x1) + 2

y_ctr5 = -x_ctr.^2+3*x_ctr+2;       ctr5_geq = 0;    % x2 <= -x1^2 + 3x1 + 2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% plot the cost function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
delta = dist(plot_left,plot_right)/plotdetails;
[X1,X2] = meshgrid(plot_left:delta:plot_right); %create a matrix of (X,Y) from vector
for i = 1:length(X1)
    for j = 1:length(X2)
```

```
        cost(i,j) = feval(costfun, [X1(i,j) X2(i,j)]);%, varargin{:});
    end
end
%figure(1)
% mesh(X1,X2,cost)
%figure
contour(X1,X2,cost,contours)
hold on

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% plot the constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
buttom_final =[];
top_final =[];
[top_final, buttom_final] = plot_ctr(x_ctr, y_ctr1, ctr1_geq, ctr1,plot_left, plot_right, plot_buttom, plot_top, top_final, buttom_final, fcolor, falpha);
[top_final, buttom_final] = plot_ctr(x_ctr, y_ctr2, ctr2_geq, ctr2,plot_left, plot_right, plot_buttom, plot_top, top_final, buttom_final, fcolor, falpha);
[top_final, buttom_final] = plot_ctr(x_ctr, y_ctr3, ctr3_geq, ctr3,plot_left, plot_right, plot_buttom, plot_top, top_final, buttom_final, fcolor, falpha);
[top_final, buttom_final] = plot_ctr(x_ctr, y_ctr4, ctr4_geq, ctr4,plot_left, plot_right, plot_buttom, plot_top, top_final, buttom_final, fcolor, falpha);

[top_final, buttom_final] = plot_ctr(x_ctr, y_ctr5, ctr5_geq, ctr5,plot_left, plot_right, plot_buttom, plot_top, top_final, buttom_final, fcolor, falpha);

if ~isempty(top_final) && isempty(buttom_final)
    fill([plot_left x_ctr plot_right],[plot_buttom top_final plot_buttom], fcolor,'FaceAlpha',falpha)
end
if ~isempty(buttom_final) && isempty(top_final)
    fill([plot_left x_ctr plot_right],[plot_top buttom_final plot_top],fcolor,'FaceAlpha',falpha)
end
if ~isempty(buttom_final) && ~isempty(top_final)
    temp = top_final;
    top_final = max(top_final, buttom_final);
    fill([x_ctr],[top_final],fcolor,'FaceAlpha',falpha)
    fill([plot_left x_ctr plot_right],[plot_buttom temp plot_buttom],fcolor,'FaceAlpha',falpha)
end

function [top_fin, buttom_fin] = plot_ctr(x_span, y_span, geq, plott, left, right, buttom, top, top_fin, buttom_fin, color, alpha)
if plott
    plot(x_span,y_span,'bla')
    if geq
        if isempty(top_fin) % first call
            top_fin = y_span;
        else
            top_fin = max(top_fin,y_span);
        end
    else
        if isempty(buttom_fin)
            buttom_fin = y_span;
        else
            buttom_fin = min(buttom_fin,y_span);
        end
    end
end
```

D.6 Auxiliary Functions

add2mat.m

```
% ADD2MAT Add/subtract/replace elements of two matrices of different sizes.
%
% Syntax:
%    NEWMATRIX = ADD2MAT(MATRIX1,MATRIX2,INITATROW,INITATCOLUMN,ADDSTYLE)
%
% Description:
%    Addition or subtraction between or replacement of elements in MATRIX1
%    by MATRIX2. MATRIX1 and MATRIX2 can be of different sizes, as long as
%    MATRIX2 fits inside MATRIX1 with respect to the initial point. MATRIX2
%    operates on MATRIX1 starting from the initial point
%    (INITATROW,INITATCOLUMN) in MATRIX1.
%
%    ADDSTYLE = 'add': Building NEWMATRIX by adding MATRIX2 to elements in
%                      MATRIX1.
%    ADDSTYLE = 'sub': Building NEWMATRIX by subtracting MATRIX2 from elements
%                      in MATRIX1.
%    ADDSTYLE = 'mul': Building NEWMATRIX by elementwise multiplication of
%    MATRIX2
%                      and elements in MATRIX1.
%    ADDSTYLE = 'div': Building NEWMATRIX by elementwise division of MATRIX2
%                      and elements in MATRIX1.
%    ADDSTYLE = 'rep': Building NEWMATRIX by replacing elements in MATRIX1 with
%                      MATRIX2.
%
% Example:
%    >> A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
%
%    A =
%
%      1      2      3      4
%      5      6      7      8
%      9     10     11     12
%     13     14     15     16
%
%    >> b = [1 1 1]
%
%    b =
%
%         1      1      1
%
%    >> B = diag(b)
%
%    B =
%
%         1      0      0
%         0      1      0
%         0      0      1
%
%    >> C = add2mat(A,B,2,2,'rep')
%
%    C =
%
%      1      2      3      4
%      5      1      0      0
%      9      0      1      0
%     13      0      0      1
%
% See also DIAG2MAT, DIAG, CAT.

% ---------------------------------------------------------------
% ADD2MAT Version 3.0
% Made by Carsten V(oe)lcker, <s961572@student.dtu.dk>
% in MATLAB Version 6.5 Release 13
% ---------------------------------------------------------------

function matrix3 = add2mat(matrix1,matrix2,initm,initn,addstyle)

if nargin < 5
    error('Not enough input arguments.')
end
if ~isnumeric(matrix1)
    error('MATRIX1 must be a matrix.')
end
if ~isnumeric(matrix2)
    error('MATRIX2 must be a matrix.')
end
if ~isnumeric(initm) || length(initm) ~= 1
    error('INITATROW must be an integer.')
```

```
end
if ~isnumeric(initn) || length(initn) ~= 1
    error('INITATCOLUMN must be an integer.')
end
if ~isstr(addstyle)
    error('ADDSTYLE not defined.')
end
[m1,n1] = size(matrix1);
[m2,n2] = size(matrix2);
if m2 > m1 || n2 > n1
    error(['MATRIX2 with dimension(s) ',int2str(m2),'x',int2str(n2),' does not fit inside MATRIX1 with dimension(s) ' ...
        ,int2str(m1),'x',int2str(n1),'.'])
end
if initm > m1 || initn > n1
    error(['Initial point (',int2str(initm),',',int2str(initn),') exceeds dimension(s) ',int2str(m1),'x',int2str(n1),...
        ' of MATRIX1.'])
end
if initm+m2-1 > m1 || initn+n2-1 > n1
    error(['With initial point (',int2str(initm),',',int2str(initn),'), dimension(s) ',int2str(m2),'x',int2str(n2),...
        ' of MATRIX2 exceeds dimension(s) ',int2str(m1),'x',int2str(n1),' of MATRIX1.'])
end
switch addstyle
    case 'add'
        matrix1(initm:initm+m2-1,initn:initn+n2-1) = matrix1(initm:initm+m2-1,initn:initn+n2-1)+matrix2;
        matrix3 = matrix1;
    case 'sub'
        matrix1(initm:initm+m2-1,initn:initn+n2-1) = matrix1(initm:initm+m2-1,initn:initn+n2-1)-matrix2;
        matrix3 = matrix1;
    case 'mul'
        matrix1(initm:initm+m2-1,initn:initn+n2-1) = matrix1(initm:initm+m2-1,initn:initn+n2-1).*matrix2;
        matrix3 = matrix1;
    case 'div'
        matrix1(initm:initm+m2-1,initn:initn+n2-1) = matrix1(initm:initm+m2-1,initn:initn+n2-1)./matrix2;
        matrix3 = matrix1;
    case 'rep'
        matrix1(initm:initm+m2-1,initn:initn+n2-1) = matrix2;
        matrix3 = matrix1;
end
```

line_search_algorithm.m

```
function [alpha,x,f,g] = line_search_algorithm(modfun,costfun,f0,g0,c,x0,delta_x,sigma,c1,varargin)

% LINE_SEARCH_ALGORITHM implemented according to Powells l1-Penalty
% function
%
%   By          : Carsten V\"olcker, s961572 & Esben Lundsager Hansen, s022022.
%   Subject     : Numerical Methods for Sequential Quadratic Optimization,
%                 Master Thesis, IMM, DTU, DK-2800 Lyngby.
%   Supervisor  : John Bagterp Jørgensen, Assistant Professor & Per Grove Thomsen, Professor.
%   Date        : 08. february 2007.

n0  = sigma'*abs(g0);
T0  = f0+n0;
dT0 = c'*delta_x-n0;
alpha1 = 1;%alpha_val;%1;

x = x0+alpha1*delta_x;
f = feval(costfun, x, varargin{:});
g = feval(modfun, x, varargin{:});
T1 = f+sigma'*abs(g);

if T1 <= T0+c1*dT0
    alpha = alpha1;
    return
end

alpha_min = dT0/(2*(T0+dT0-T1));
alpha2 = max(0.1*alpha1, alpha_min); % skal 0.1 være c1 i stedet for??

```

```
x = x0+alpha2*delta_x;
f = feval(costfun, x, varargin{:});
g = feval(modfun, x, varargin{:});
T2 = f+sigma'*abs(g);

if T2 <= T0+c1*alpha2*dT0
    alpha = alpha2;
    return
end

stop = 0;
max_itr = 100;
itr = 0;
while ~stop
    itr = itr+1;
    if itr > max_itr
        disp('line search (itr > mat_itr)');
        stop = 1;
    end

    ab = 1/(alpha1-alpha2)*[1/(alpha1*alpha1) -1/(alpha2*alpha2);-alpha2/(alpha1*alpha1) alpha1/(alpha2*alpha2)]*[T1-dT0*alpha1-T0; T2-dT0*alpha2-T0];
    a = ab(1);
    b = ab(2);
    if( abs(a)<eps )
        alpha_min = -dT0/b;
    else
        alpha_min = (-b+(sqrt(b*b-3*a*dT0)))/3*a;
    end

    if( alpha_min <= 0.1*alpha2 )
        alpha = 0.1*alpha2;
    else
        if (alpha_min >= 0.5*alpha2)
            alpha = 0.5*alpha2;
        else
            alpha = alpha_min;
        end
    end

    x = x0+alpha*delta_x;
    f = feval(costfun, x, varargin{:});
    g = feval(modfun, x, varargin{:});
    T_alpha = f+sigma'*abs(g);

    if T_alpha <= T0+c1*alpha*dT0
        return
    end
    alpha1 = alpha2;
    alpha2 = alpha;
    T1 = T2;
    T2 = T_alpha;
end
```