NTNU – Trondheim
Norwegian University of
Science and Technology

Efficient Implementation of Dual First Order Algorithms

Application in Embedded MPC

Sverre Kvamme

Master of Science in Cybernetics and Robotics
Submission date: December 2014
Supervisor: Morten Hovd, ITK
Co-supervisor: Ion Necoara, UPB

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

The aim of the diploma project is to study the implementation challenges of numerical optimization algorithms that appear in embedded model predictive control.

A detailed review of some of the most important numerical optimization algorithms for solving embedded model predictive control problems will be given. A selection of these algorithms should be implemented in the C programming language, and performance comparisons should be made. Furthermore, a toolbox containing these algorithms should be created. The toolbox should have a Matlab interface for easy and efficient analysing and testing.

The work will be carried out at the University Politehnica of Bucharest, in the group of Associate Prof. Ion Necoara.

Preface

This Thesis completes my master's degree in engineering cybernetics at NTNU. The Thesis was written at University Politehnica of Bucharest (UPB), Romania, during the fall of 2014. The project proved to be both exiting and educational, as well as challenging at times. Over all, my stay in Bucharest ended up being a very nice experience.

The concrete result of the project ended up to be a versatile and efficient QP solver toolbox [1]. In collaboration with professor Ion Necoare, an attempt to publish a paper on this software, at the 54th IEEE Conference on Decision and Control, December 2015, Osaka, Japan, will be made.

I would like to thank my supervisor Morten Hovd for making the collaboration with UPB possible, and I would like to thank my supervisor at UPB, Ion Necoara, for his good help and dedicated support throughout the project.

Moreover, I would like to give my appreciation to the SEE Grant for partially founding my stay in Romania.

Furthermore, I would like to thank, Andrei Patrascu, with whom I shared an office with at UPB, and also the other good friends I made at UPB during my stay in Romania: Florin Stoican, Dragos Clipici, Paul Irofti, and Claudiu Dinicu.

Finally, I would like to thank my family for supporting me during all the years of university studies, and my lovely Ellen-Kristin Raasok for always being there for me throughout the ups and downs of this project.

Bucharest, 2014-12-20

Sverre Kvamme

Abstract

In this Thesis, numerical implementation of optimization algorithms for convex quadratic problems that appear in model predictive control for embedded linear systems, are examined. Different versions of dual first order methods are introduced and their complexity estimates are presented. The methods are implemented in the efficient programming language C, and optimized for low iteration complexity and low memory footprint. Extensive numerical simulations are conducted to test their performance and robustness, both against each other and against a commercial solver. Furthermore, a toolbox called *DuQuad* [1], that contains the implemented algorithms, is developed. The toolbox has a dynamic MATLAB interface which make the process of testing, comparing, and analysing the algorithms simple. The algorithms are implemented using only basic arithmetic and logical operations and are suitable to run on low cost hardware. It is shown that if an approximate solution is sufficient for a given application, there exists problems where some of the implemented algorithms obtain the solution faster than the state-of-the-art commercial solver.

Sammendrag

Denne rapporten omhandler implementering av optimeringsalgoritmer for konvekse kvadratiske problemer man finner i MPC for innebygde linære systemer. Forskjellige versjoner av dual første ordens metoder er introdusert, og estimater av deres kompleksitet er presentert. Disse metodene er implementert i det effektive programmeringsspråket C, og optimert i forhold til kompleksitet og lavt minnebruk. Omfattende numeriske simuleringer er utført for å teste deres ytelse og robusthet, både i forhold til hverandre og i forhold til et kommersielt optimeringsverktøy. En toolbox kalt DuQuad [1], som inneholder alle de implementerte algoritmene, er utviklet. Dette verktøyet har et dynamisk MATLAB brukergrensesnitt som gjør testing, sammenlikning og analysering av algoritmene enklere. Alle algoritmene er implementert ved å bruke enkle aritmetiske og logiske operasjoner, og er egnet til å kjøre på lavytelses hardware. Det er vist at om en tilnærmet løsning er tilstrekkelig for en gitt applikasjon, eksisterer det problemer hvor noen av de implementerte metodene finner en løsning raskere enn et ledende kommersielt optimeringsverktøy.

Contents

1. Introduction

1.1. Background and Motivation for the project

Model predictive control (MPC) is a highly relevant subject within the field of engineering automation. Because MPC can be a computationally demanding algorithm, it has commonly been utilized in slow processes with large time constant. However, as the computational capabilities of processors have improved drastically over the last decades, MPC schemes have also been introduced in faster real-time systems. As the solvers for the optimization problem in MPC now are being tailored for different applications, MPC on resource constrained platforms, e.g. microcontrollers, can perform on faster and faster real-time systems.

The optimization problems for linear systems that arises from condensed and sparse MPC formulations are often convex quadratic programs (QPs). Today, many solvers that are tailored for such optimization problems exists. However, most of these solvers are based on second order methods, which can offer fast rates of convergence in practice, but where the worst case complexity bounds are high. Furthermore, these methods have complex iterations, involving inversion of matrices, which are usually difficult to implement on embedded systems, where the unit demand simple computations. First order methods, on the other hand, can be implemented using less complex operations, and are often better suited in situations where low memory footprint and predictable behaviour are required.

Commercial dual first order QP solvers exist on the marked today. However, these are tailored to solve only specific cases of QP problems. The main motivation for this Master's Thesis is therefore to implement a toolbox containing different versions of dual first order optimization algorithms that can solve more general formulations of QPs.

1.2. Main Objectives

In addition to gain a deeper understanding within the field of MPC and convex optimization, the main objectives of this Master's Thesis can be summarised as follows:

- Study numerical first order optimization algorithms that solves convex quadratic problems that appear in MPC for embedded linear systems.
- Implement some of these algorithms in the efficient low-level programming language C, while optimizing for low iteration complexity and low memory footprint.
- Compare and analyse the robustness and convergence of these algorithms in practice.

- Finalize a complete SW program in the form of a toolbox that contains the implemented algorithms and has a simple user interface.

1.3. Project Overview

The project consists of two parts; one theoretical and one practical.

The theoretical part of the report starts in Chapter 2. First, a general introduction to MPC is provided, and an input constrained MPC problem is recast as a QP. Second, some concepts and properties of embedded MPC are reviewed. General optimization theory is studied in the third part, and the chapter ends with a section about convex quadratic programming.

The theoretical part continues in Chapter 3 where four different quadratic dual first order optimization algorithms are presented in detail.

The practical part of the project consists of implementing, testing and analysing the algorithms discussed in Chapter 3. These algorithms are all bundled into a toolbox, called *DuQuad*, which is discussed in Chapter 4. A presentation and discussion of the numerical simulations carried out in the project are found in Chapter 5.

A conclusion is given in Chapter 6, and a discussion of possible future work are given in Chapter 7. Theoretical bounds on convergence are stated in Appendix A, an alternative matrix multiplication method is presented in Appendix B, and a user manual for the DuQuad toolbox is given in Appendix C.

1.4. Notation

The notation throughout this report will be as follows:

- $[z]_{\mathcal{Z}}$ means that z is projected onto the set $\mathcal{Z}$.
- $[\,\cdot\,]_+$ means projection on a non-negative orthant $\mathbb{R}^n_+$.
- $[\,\cdot\,]_+$ and $[.]_{\mathcal{Z}}$ are used for spaces of different dimensions.
- $\|z\|$ denote the Euclidean norm, $\|z\| := \sqrt{z_1^2 + \cdots + z_n^2}$.
- $H \succeq 0$ means that the matrix H is positive semidefinite.
- $H \succ 0$ means that the matrix H is positive definite.
- $:=$ means defined as.
- $\lfloor \alpha \rfloor$ denotes the largest integer which is less than or equal to α.
- $\langle x, y \rangle := x^T y := \sum_{i=1}^n x_i y_i$, i.e. vector multiplication.
- n is the size of the hessian matrix H, which is an $n \times n$ matrix.
- m is the row size of the linear constraint matrix A, which is an $m \times n$ matrix.
- k is used to denote the iteration index in algorithms.

2. Literature Survey

This chapter contains control and optimization theory that the reader should be familiar with in order to better understand the following chapters in this Master's Thesis. The chapter is divided into four sections. In the first section, model predictive control is introduced, and a QP problem is derived from a general condensed MPC formulation. Second, different requirements and properties of embedded MPC are reviewed. In the third section, some concepts within optimization theory is presented. Finally, quadratic programming is discussed.

2.1. Model Predictive Control

This section will first give a brief introduction to *model predictive control* (MPC), and then show how a standard condensed MPC problem can be recast into a quadratic program. Some parts of this section is modified from the Project thesis of Kvamme [2].

2.1.1. Introduction

Model Predictive Control was first introduced in the 1970s and has been in use in chemical plants and oil refineries since the 1980s. In general, MPC is an optimal control strategy, based on numerical optimization, that can control more complex processes. The strategy is described by dynamical models, and has the advantage of controlling large multi-variable systems with constraints on inputs and states variables, while simultaneously optimizing the predicted future. Today MPC is still a very popular advanced control technology in chemical process industry, but the method is also emerging into several other application areas. Furthermore, as computers constantly are getting faster and more resourceful, MPC has lately been utilized in faster real-time systems, or *real-time computing* (RTC).

Most processes do not have a linear characteristic, i.e. they are nonlinear. However, these processes can be linearized around their equilibrium point to make a linear model. This model will not be exact, but in most cases good enough to control and keep the system stable. The linear model can also be obtained by system identification methods based on measured data. Therefore, the most widespread type of MPC is usually termed *linear MPC*. This type of MPC uses a linear *state-space* model for the prediction and a convex QP for the optimization.

In the beginning, MPC applications were optimized for a finite horizon. Nonetheless, during the 90s, theory was developed for linear process models to have an infinite horizon. It can be proven that an infinite horizon gives better theoretical properties, and is therefore often the preferred choice. For an infinite horizon, the *linear quadratic controller* (LQR) delivers an optimal performance and stabilizes the

system. On the other hand, the LQR does not handle constraints. This disadvantage of LQR is the main reason for the increased popularity of MPC. MPC gives the opportunity, in contrast to LQR, to handle constraints on both inputs and outputs of the system.

The MPC problem consists of an objective function and constraints. This is typically categorized as a convex or a non-convex optimization problem. In this Thesis, only convex optimization problems, where the optimization problem is a quadratic program (QP), is considered. A QP has a quadratic objective function and linear constraints.

2.1.2. Principle

The basic principle of MPC can be summarized as follows

- At a given point in time, based on the current state of the system, the objective function (from the state-space model of the plant), and the given constraints, find a sequence of optimal and feasible future inputs to the process.
- Apply only the first optimal input of the sequence to the process.
- Go to next timestep and repeat the process.

The length of the calculated optimal sequence of inputs is given by the length of the horizon N. The prediction horizon remains the same length despite the repetition of the optimization at future time instants, as shown in Figure 2.1. The strategy is called *receding horizon strategy*.

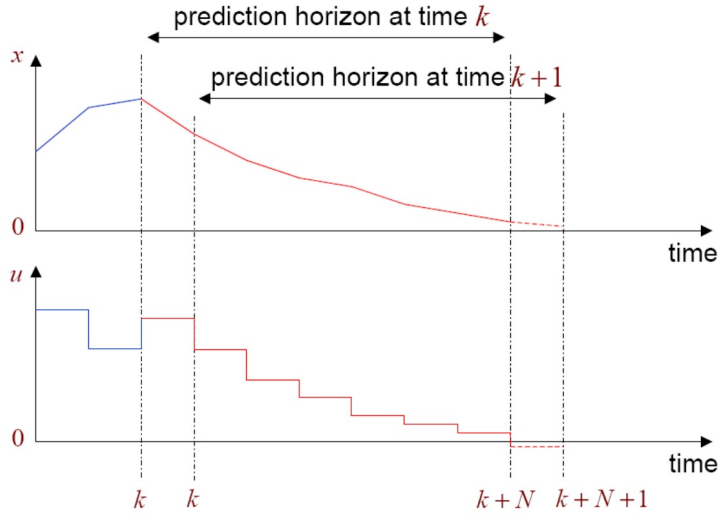


Figure 2.1: The receding horizon strategy. The length of the horizon, N, is constant. x is the state of the system while u is the input. Adapted from [3]

2.1.3. Defining the MPC Problem

A simple state space model of a plant is considered as an example for further discussion. It is assumed that the plant is time-invariant, linear, and in discrete time. Furthermore, the optimization problem is subjected to (s.t.) constraints on states and inputs, and have a quadratic cost criterium. A model of a plant that fulfill these conditions, and ignore disturbances on measurements and states, can be described as follows:

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k \\ y_k &= Cx_k, \end{aligned} \tag{2.1}$$

where $x_k \in \mathbb{R}^{n_x}$ is the state vector, $u_k \in \mathbb{R}^{n_u}$ is the input vector, and $y_k \in \mathbb{R}^{n_y}$ is the output/measurements vector. Furthermore, $A \in \mathbb{R}^{n_x \times n_x}$ is the state matrix, $B \in \mathbb{R}^{n_x \times n_u}$ is the input matrix, and $C \in \mathbb{R}^{n_y \times n_x}$ is the output matrix. The subscript $k+1$ refers ro the sample instant one sample interval after sample k. The state x_k and input u_k should be considered as *deviation variables*, i.e. they represent a error from some constant set of variables $\{\tilde{x}, \tilde{u}\}$. For a continuous process, $\{\tilde{x}, \tilde{u}\}$ will typically represent a stationary point where the process is in normal working-mode. There are no restrictions on the eigenvalues of A, i.e. the system could be stable or unstable.

Constraints

The system in equation 2.1 can be subjected to certain constraints on the states and inputs. The constraints on the states can be described as

$$Y_{lb} \leq Cx_k \leq Y_{ub}, \tag{2.2}$$

where Y_{lb}, $Y_{ub} \in \mathbb{R}^{n_x}$. Note that Cx_k also describes the measurements y. The constraints on the inputs u are given as

$$U_{lb} \leq u_k \leq U_{ub}, \tag{2.3}$$

where U_{lb}, $U_{ub} \in \mathbb{R}^{n_u}$. These constraints are also called box-constraints. Finally, constraints on the last state in an MPC problem can be included

$$T_{lb} \leq Tx_N \leq T_{ub}, \tag{2.4}$$

where T_{lb}, $T_{ub} \in \mathbb{R}^{n_x}$. This final, or terminal, state is explained in the following sections.

LQR and Dual Mode

The goal is to control the system in equation (2.1), i.e. $(x_k \to 0)$, while optimize the performance by minimizing the infinite horizon cost

$$J(x_0, u_k) = \sum_{k=0}^{\infty} (x_k^T Q x_k + u_k^T R u_k), \tag{2.5}$$

where future x_k are given by x_0 and u_k. Q and R are called tuning matrices. R has to be positive definite while Q only can be positive semidefinite[1]. Both matrices has to be symmetric.

In the case where there are no constraints for the system, the optimal solution that minimizes the function in equation (2.5) will be given by the the state feedback

$$u_k = -Kx_k, \tag{2.6}$$

where K is the *state feedback matrix*. This controller is called infinite horizon LQ controller, or the Linear Quadratic Regulation (LQR), and K is found from the Riccati equation [5]. Substituting for u_k, equation (2.5) can be rewritten in the following way:

$$\sum_{k=0}^{\infty} x_k^T Q x_k + u_k^T R u_k = \sum_{k=0}^{\infty} x_k^T (Q + K^T R K) x_k = x_0^T P x_0. \tag{2.7}$$

The LQR is optimal on the infinite horizon, multivariabel, robust, and easy to compute. However, as mentioned earlier, it has the drawback of not handling constraints.

In [2], it is shown how to use a method called *dual mode predictions* to divide the infinite horizon in two. Using this method the objective function is rewritten as

$$\sum_{k=0}^{\infty} x_k^T Q x_k + u_k^T R u_k = \sum_{k=0}^{N-1} \left\{ x_k^T Q x_k + u_k^T R u_k \right\} + x_N^T P x_N. \tag{2.8}$$

The first part of equation (2.8) is solved as a MPC problem with finite horizon N, while and the second part, with infinite horizon, is solved using LQR (or another feedback law that can guarantee stability). This scheme is illustrated in Figure 2.2.

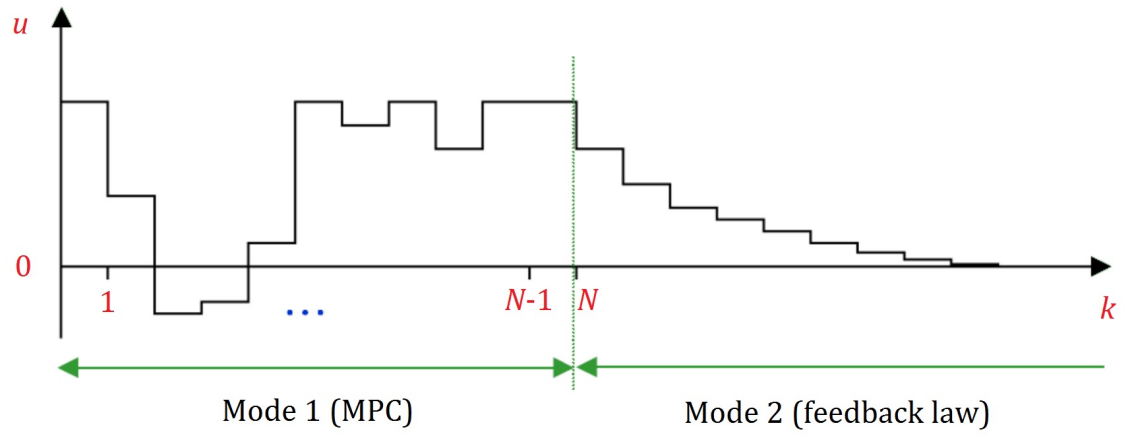


Figure 2.2: Dual mode input predictions

The logic behind this strategy is that the constraints are resolved before $k = N$,

[1]If a matrix is positive definite, all the eigenvalues are positive. If all eigenvalues are positive **or zero** the matrix is positive semidefinite [4].

and therefore the optimal solution of the LQR can be used for $k \geq N$ when no constraints are active. When combining the finite horizon MPC and the LQR, an MPC scheme with infinite horizon is obtained. By optimizing for on the infinite horizon, and only apply the first part of the optimal input to the system, feedback is achieved and the system can be kept stable.

Condensed MPC Problem

The column vectors x and u are defined as

$$x := \begin{bmatrix} x_1 & x_2 & \cdots & x_{N-1} & x_N \end{bmatrix}^T, \qquad u := \begin{bmatrix} u_0 & u_1 & \cdots & u_{N-2} & u_{N-1} \end{bmatrix}^T. \tag{2.9}$$

Given the result in equation (2.8), a general condensed MPC problem, which optimizes the input u and includes constraints on states and inputs, can be stated as follows:

$$\begin{aligned} \min_{u} \quad & \textstyle\sum_{k=0}^{N-1} \frac{1}{2} \left\{ x_k^T Q x_k + u_k^T R u_k \right\} + \frac{1}{2} x_N^T P x_N \\ \text{s.t.} \quad & Y_{lb} \leq C x_k \leq Y_{ub} \\ & U_{lb} \leq u_k \leq U_{ub} \\ & T_{lb} \leq T x_N \leq T_{ub} \\ & x_0 = \text{given}, \end{aligned} \tag{2.10}$$

where x_k is given by the state space in equation (2.1). In this problem formulation, P is referred to as the *terminal state weight* matrix and x_N is the terminal state. Constraints on the terminal state are also included in the problem. This condensed MPC formulation is also called input-only constrained MPC [6].

2.1.4. Condensed MPC Problem Recast as a Standard QP

In the following, the optimization problem in equation (2.10) will be recast as a QP on standard form. When the initial state x_0 is given as input to the problem, the next x_k can be calculated as follows:

$$\begin{aligned} x_1 &= Ax_0 + Bu_0, \\ x_2 &= Ax_1 + Bu_1 \\ &= A(Ax_0 + Bu_0) + Bu_1 \\ &= A^2 x_0 + ABu_0 + Bu_1, \\ x_3 &= Ax_2 + Bu_2 \\ &= A(A^2 x_0 + ABu_0 + Bu_1) + Bu_2 \\ &= A^3 x_0 + A^2 Bu_0 + ABu_1 + Bu_2, \\ &\vdots \\ x_k &= A^k x_0 + A^{k-1} Bu_0 + A^{k-2} Bu_1 + \cdots + Fu_{k-2} B + Bu_{k-1}. \end{aligned} \tag{2.11}$$

This means that future states x_k can be calculated from the initial state x_0 and the inputs u. By defining

$$\tilde{A} := \begin{bmatrix} A \\ A^2 \\ \vdots \\ A^{N-1} \\ A^N \end{bmatrix}, \qquad \tilde{B} := \begin{bmatrix} B & 0 & 0 & \dots & 0 \\ AB & B & 0 & \ddots & 0 \\ A^2B & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & AB & \ddots & \vdots \\ A^{N-1}B & \dots & A^2B & AB & B \end{bmatrix}, \tag{2.12}$$

x is defined by the function

$$\boxed{x = \tilde{A}x_0 + \tilde{B}u.} \tag{2.13}$$

Further, the weighting matrices Q, R, and P are placed as elements in the sparse matrices $\tilde{Q}$ and $\tilde{R}$, and C is placed in $\tilde{C}$ as follows:

$$\tilde{Q} := \begin{bmatrix} Q & 0 & \cdots & 0 & 0 \\ 0 & Q & \ddots & \vdots & \vdots \\ 0 & 0 & \ddots & 0 & 0 \\ \vdots & \vdots & \ddots & Q & 0 \\ 0 & 0 & \cdots & 0 & P \end{bmatrix}, \quad \tilde{R} := \begin{bmatrix} R & 0 & \cdots & 0 & 0 \\ 0 & R & \ddots & \vdots & \vdots \\ 0 & 0 & \ddots & 0 & 0 \\ \vdots & \vdots & \ddots & R & 0 \\ 0 & 0 & \cdots & 0 & R \end{bmatrix}, \quad \tilde{C} := \begin{bmatrix} C & 0 & \cdots & 0 & 0 \\ 0 & C & \ddots & \vdots & \vdots \\ 0 & 0 & \ddots & 0 & 0 \\ \vdots & \vdots & \ddots & C & 0 \\ 0 & 0 & \cdots & 0 & T \end{bmatrix}. \tag{2.14}$$

In addition, the constraints from equation (2.10) are placed as elements in the new vectors

$$\hat{lb} := \begin{bmatrix} Y_{lb} \\ \vdots \\ Y_{lb} \\ T_{lb} \end{bmatrix}, \quad \hat{ub} := \begin{bmatrix} Y_{ub} \\ \vdots \\ Y_{ub} \\ T_{ub} \end{bmatrix}, \quad lb := \begin{bmatrix} U_{lb} \\ \vdots \\ U_{lb} \end{bmatrix}, \quad ub := \begin{bmatrix} U_{ub} \\ \vdots \\ U_{ub} \end{bmatrix}. \tag{2.15}$$

By substituting x using equation (2.13), the objective function in (2.10) can be rewritten

$$\begin{aligned} \tfrac{1}{2}x^T\tilde{Q}x + \tfrac{1}{2}u^T\tilde{R}u &= \tfrac{1}{2}(\tilde{A}x_0 + \tilde{B}u)^T\tilde{Q}(\tilde{A}x_0 + \tilde{B}u) + \tfrac{1}{2}u^T\tilde{R}u \\ &= \tfrac{1}{2}u^T(\tilde{B}^T\tilde{Q}\tilde{B} + \tilde{R})u + x_0^T\tilde{A}^T\tilde{Q}\tilde{B}u + \tfrac{1}{2}x_0^T\tilde{A}^T\tilde{Q}\tilde{A}x_0, \end{aligned} \tag{2.16}$$

and the term Cx from the linear constraints

$$\tilde{C}x = \tilde{C}\tilde{A}x_0 + \tilde{C}\tilde{B}u. \tag{2.17}$$

Finally, by defining

$$H := \tilde{B}^T\tilde{Q}\tilde{B} + \tilde{R}, \qquad c := x_0^T\tilde{A}^T\tilde{Q}\tilde{B}, \qquad G := \tilde{C}\tilde{B}, \qquad g := \tilde{C}\tilde{A}x_0, \tag{2.18}$$

the QP which solves the condensed MPC problem in equation (2.10) is stated as follows:

$$\begin{aligned} \min_{u} \quad & \tfrac{1}{2} u^T H u + c^T u \\ \text{s.t.} \quad & \hat{lb} \leq Gu - g \leq \hat{ub} \\ & lb \leq u \leq ub. \end{aligned} \tag{2.19}$$

In Chapter 3, different algorithms for solving QPs on this form are discussed.

Sparse MPC

In the condensed MPC formulation, only the inputs u are optimized. However, when optimizing for both the the states x and the inputs u, the resulting optimization problem is called a *sparse* MPC problem. If defining $z = [x^T, u^T]$ it can be shown that a corresponding QP can be formulated as follows [7]:

$$\begin{aligned} \min_{z} \quad & \tfrac{1}{2} z^T \bar{H} z + \bar{c}^T z \\ \text{s.t.} \quad & \bar{G} z = \bar{g} \\ & \bar{lb} \leq z \leq \bar{ub}. \end{aligned} \tag{2.20}$$

For a more detailed deviation of the equations presented in this chapter, see Hovd [4] and Imsland [5]. Furthermore, different concepts such as MPC tracking, hard and soft constraints, preconditioning, tuning, and stability are discussed in Kvamme [2].

2.2. Embedded MPC

An *embedded system* is normally a computer system that is dedicated to perform one specific function, and is embedded as part of a larger mechanical or electrical system. Embedded systems usually operates with real-time computing constraints, e.g. operational deadlines from an event will occur to the system makes a response. In most cases, it will be overkill to utilize a powerful and expensive general-purpose computer to perform the function of an embedded system. As a consequence, modern embedded systems are often based on microcontrollers, which have constraints in computational efficiency and memory capacity.

Usually, MPC relies on complex optimization algorithms that require high computational efficiency, when dealing with a large number of constraints and a long horizon. Consequently, the MPC community has a large focus on developing fast optimization solvers, particularly QP solvers, for enabling the use of linear MPC in real applications.

This section is modified from the Project Thesis of Kvamme [2].

2.2.1. Requirements

Bemporad and Patrinos [8] describes the different requirements for a QP solver to be embedded in the control hardware, when implementing a MPC law. The QP solver must

- be so fast that it can provide a solution within short samling intervals, typically 1-50 ms
- run on simple hardware, e.g. a microcontroller
- be on a structure where the data defining the optimization problem and the code for implementing the algorithm itself, requires a small amount of memory space.
- have a good worst-case estimation of the computation time to meet hard real-time requirements
- have a simple code that is software-certifiable (especially if safety is critical).

2.2.2. Implicit and Explicit Solutions

When solving the QP problems appearing in MPC, there are two main approaches: *Implicit* and *explicit*, respectively also called *online* and *offline*.

In explicit MPC, the solution for all possible states are computed offline and stored in for example a table. This approach makes the computation for every time-step significantly smaller. Hence, the sampling time can be reduced. However, the demand for memory capacity will in general grow exponentially with the number of states, inputs, and horizon. As a consequence, if there are limitations in the memory capacity, an explicit MPC is restricted to small scale systems.

In explicit MPC the solution of the optimization is computed online, i.e. for every time-step a new QP problem is solved by the controller. However, online optimization is usually limited by the computation speed. In this Thesis the main focus is on online optimization.

2.2.3. Rate of Convergence and Upper Bound on Number of Iterations

The sampling time for fast embedded systems is often very short. As a consequence, embedded systems require an optimization algorithm that offers a tight bound on the total number of iterations that needs to be performed in order to compute the desired optimal solution. Nedelcu et al. [9] argue that second order methods, e.g. interior-point methods, offers a fast convergence rate in practice, i.e. they usually converge to a solution with a significantly lower number of iterations than those predicted by the theoretical worst case analysis. However, the worst cast complexity bounds for interior-point methods are high. Furthermore, these algorithms do involve complex iterations, e.g. inversion of matrices. Such operations are typically hard to implement on a microcontroller that requires simple computational instructions. Therefore, first order methods are often preferred, as they often use a number of iterations that is close to the worst case complexity analysis.

2.2.4. Literature Survey

A number of different approaches for MPC formulation exist today. The formulation and solving of the optimization problem have a large impact on computation and memory demands. However, many of the proposed formulations and algorithms utilize the same general ideas and tactics.

Among the papers investigated for this thesis are Bemporad and Patrinos [8], Nedelcu et al. [9], Kögel and Findeisen [10], Kögel et al. [11], Necoara and Nedelcu [12], Necoara and Patrascu [13], Richter et al. [14], and Zometa et al. [15]. These papers were all published during the last three years, and deal with solving QP problems for MPC schemes. Some commonalities are listed:

- They consider MPC on linear, time-invariant, discrete-time systems, subject to constraints on the states and input and a quadratic cost criterion.
- They formulate the QP on condensed form, and benefit from a well conditioned Hessian.
- Nesterov's fast gradient method and the augmented Lagrangian method are applied
- The solvers only provides approximate solutions, and strive towards providing tight estimate on the number of iterations in order to reach a given accuracy in terms of optimality and feasibility of the optimal solution.
- Warm-start are applied in the solvers to obtain fewer iterations in the converging to a feasible solution.

2.3. Optimization Theory

This section will give an introduction to some concepts within numerical optimization.

2.3.1. Optimization Algorithms

As a motivation, an overview of different optimization algorithms is given in the following. Optimization algorithms are iterative, i.e. they start with an initial guess of the optimization variable z and take steps (iterate) towards an optimal or close to optimal solution of z. Every new step is an improved estimates of the solution compared to the previous step. The scheme for generating the next iterate is what varies between the algorithms, but most strategies makes use of the values of the objective function f and the constraints, and possibly the first and second derivatives of these functions. Some algorithms utilize the information from previous iterates, while other only use information from the current point. Stopping criteria (also called termination criteria) vary between different algorithms. Regardless of how an algorithm is implemented, every good algorithm should possess the following properties [16]:

- Efficiency. It should be efficient in both computation time and memory usage.
- Robustness. Within a given class, the algorithm should be able to solve a variety of different problems, for all reasonable values of the starting point.
- Accuracy. It should be able to present a solution with precision.

However, these features may conflict with each other. For example, a faster computation time will often be gained by obtaining a solution with lower accuracy. Tradeoffs between the properties are central issues in numerical optimization, and will be a returning subject through this Thesis.

Line Search and Trust Region

Basically, the strategy for finding the next iterate can be divided into two main groups: *line search* and *trust region*.

The line search strategy is to find a descent direction p_k and search along this direction from the current iterate z_k for a new iterate with a lower function value. In other words, the algorithm first chooses the search direction and then computes how far to go in that direction. The search direction is usually the steepest descent direction $-\nabla f_k$.

Trust region methods, also known as restricted step methods, use the opposite strategy of line search. It first chose a maximum distance (the trust region radius) and then seek the direction and step that gives the best improvement subject to this distance constraint. If the new step is not satisfactory, the radius is reduced and the algorithm seek a new step.

All algorithms considered in this Thesis fall under the first category, i.e. line search.

First and Second Order Algorithms

Another way to distinguish different optimization algorithms is to categorize them as first- or second-order algorithms. Basically, second order method is used to compute the step length in first order method. Obviously there are advantages and disadvantages, as discussed in Section 2.2.3.

2.3.2. Unconstrained Optimization

In the simplest form, an optimization problem consists of minimizing or maximizing a real function. Such a problem can be expressed as

$$\min_{z \in \mathbb{R}^n} \quad f(z). \tag{2.21}$$

This is an unconstrained problem where f is a function dependent on the optimization variable z, where z is only constrained by the set off all real numbers $\mathbb{R}^n$. The goal is to find a z that makes the function value of f as small as possible. The optimal z and f are defined as

- z^* is the *optimal solution*.
- $f^* = f(z^*)$ is the *optimal value*.

The optimal solution z^* may also be referred to as the optimal point. The ideal scenario is to find a *global* optimal solution, which satisfy $f(z^*) \leq f(z)$ for all z. The solution will be a *strict* global optimal solution if it also satisfied $f(z^*) < f(z)$ for all z.

2.3.3. Convexity

The concept of convexity is fundamental in optimization. Many problems used in practical applications possess this property which generally makes them easier to solve. The term "convex" is described in [16] and can be applied to sets and to functions. A set $S \in \mathbb{R}$ is convex if every line segment between any two points in S lies inside the set. A function f is convex if its domain S is a convex set and if for any two points x and y in S, the following property holds:

$$f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y), \qquad \forall \alpha \in [0,1]. \tag{2.22}$$

Another important property is given by the fact that if $-f(z)$ is convex, then $f(z)$ is called concave. In this Thesis, only convex objective functions are considered. Theorem 2.5 in [16] states that when f is convex, any local minimizer z^* is also a global minimizer of f.

Lipschitz Continuity

A function f is said to be *Lipschitz continous* on som set $S \subset D$ if there exists a constant $L > 0$ such that

$$|f(z) - f(y)| \leq L \parallel z - y \parallel, \quad \forall z, y \in S. \tag{2.23}$$

where the constant L is called the *Lipschitz constant*, see [17]. In other words, a Lipschitz continuous function is limited in how fast it can change. The absolute value of the slope connecting any two points on the graph can not be grater than L.

The function f has a Lipschitz gradient on the set S if

$$\| f(z) - f(y) \| \leq L \| z - y \|, \quad \forall z, y \in S, \tag{2.24}$$

and is then bounded from above by the function

$$f(y) \leq f(z) + \langle \nabla f(z), y - z \rangle + \frac{L}{2} \|y - z\|^2, \quad \forall z, y \in S. \tag{2.25}$$

Strongly Convex Function

Nesterov [17] defines *strongly* convex functions in Definition 2.1.2. A continuously differentiable function $f(z)$ is called strongly convex on $\mathbb{R}^n$ if there exists a constant $\mu \geq 0$ such that for any z, $y \in \mathbb{R}^n$ the following relation holds

$$f(y) \geq f(z) + \langle \nabla f(z), y - z \rangle + \frac{\mu}{2} \|y - z\|^2. \tag{2.26}$$

The constant μ is called the convexity parameter of function f.

2.3.4. Recognizing a Local Minimum

One approach to find the local minimum is to compare all the different points, z, in the immediate neighbourhood and choose the point which gives the lowest function value of $f(z)$. Nevertheless, as this approach is quite computational expensive, there are other ways to tell if a point is the local minimum or not. If the function is *smooth*, Theorem 2.3.1 can be useed to identify the local minima. If f is twice differentiable it may also be possible to decide whether the local minimum is also a strict local minima, by using Theorem 2.3.2 and Theorem 2.3.3.

Theorem 2.3.1 (Theorem 1.2.1 in [18]: First order optimality condition)**.** *Let z^* be a local minimum of differentiable function $f(z^*)$. Then*

$$\nabla f(z^*) = 0. \tag{2.27}$$

Theorem 2.3.2 (Theorem 1.2.2 in [18]: Second-order Necessary optimality condition)**.** *Let z^* be a local minimum of twice differentiable function $f(z^*)$. Then*

$$\nabla f(z^*) = 0, \quad \nabla^2 f(z^*) \succeq 0. \tag{2.28}$$

Theorem 2.3.3 (Theorem 1.2.3 in [18]: Second-order sufficient optimality condition)**.** *Let function $f(z)$ be twice differentiable on $\mathbb{R}^n$ and let z^* satisfy the following conditions:*

$$\nabla f(z^*) = 0, \quad \nabla^2 f(z^*) \succ 0. \tag{2.29}$$

Then z^* is a strict local minimum of $f(z)$

2.3.5. Constrained Optimization

Constrained problems are problems where a feasible solution must lie within a given set $S \subset \mathbb{R}$. *Linear* constrained problems are problems where all functional constraints are linear. A general constrained optimization problem, which includes linear equalities and inequalities constraints as well as constraints on the optimization variable, can be expressed as follows:

$$\begin{aligned} \min_{z \in \mathcal{Z}} \quad & f(z) \\ \text{s.t.} \quad & a_i(z) = 0, \qquad i \in \mathcal{E} \\ & a_i(z) \leq 0, \qquad i \in \mathcal{I}, \end{aligned} \tag{2.30}$$

where $\mathcal{E}$ and $\mathcal{I}$ are finite sets of indices. The linear equality and inequality constraints $a_i(z) \in \mathcal{E} \cup \mathcal{L}$ are called *complicated* constraints. Projection on complicated constraints in the primal problem can be a complex and difficult operation. In the next sections it is shown how to use Lagrangian relaxation to move the complicated constraints into the objective function, which create a problem that is easier to solve.

Furthermore, the optimization variable z is constrained by by the set $\mathcal{Z}$. This is typically a *simple* set that includes a lower and upper bound for the variable. Given the vector $z \in \mathbb{R}^n$, the bounds can be described as the vectors $lb, ub \in \mathbb{R}^n$, and the set as $\mathcal{Z} = [lb, ub]$. Projection on simple set are done easily, and projection on $\mathcal{Z}$ has the complexity $\mathcal{O}(2n)$. The constraints made of lb and ub are also called *boxed constraints*.

2.3.6. Lagrange Multipliers

The concept of Lagrange multipliers are of great importance in optimality theory, and the *method of Lagrange multipliers* is a strategy for finding the local minima (or maxima) of a function subjected to constraints. The lagrangian function for the constrained problem in equation (2.30) is defined as

$$\mathcal{L}(z, \lambda) := f(z) + \sum_{i \in \mathcal{E} \cup \mathcal{L}} \lambda_i a_i(z), \tag{2.31}$$

where λ_i is called the Lagrangian multiplier. The *first order conditions* for the Lagrangian function, also known as the *Karush-Kuhn-Tucker conditions* (*KKT conditions*), are described in Theorem 2.3.4. The conditions in (2.32e) are complementarity conditions meaning that either $\lambda_i^* = 0$ or $a_i(z^*) = 0$, or both equals to zero. This property is useful for checking which inequality constraints are active at the solution. If the Lagrange multiplier $\lambda_i^* = 0$, the constraint a_i is not significant. However, it may still be active. If $\lambda_i^* > 0$, the constraint a_i is strongly active *or* binding, see Definition 12.8 [16].

Theorem 2.3.4 (Theorem 12.1 in [16]: First Order Necessary Conditions)**.** *Suppose that z^* is a local solution of (2.30), that the function f and a_i in (2.30) are*

continuously differentiable, and that the LICQ [2] *hold at* z^*. *Then there is a Lagrange multiplier vector* λ^*, *with components* $\lambda_i^*, i \in \mathcal{E} \cup \mathcal{I}$, *such that the following conditions are satisfied at* (z^*, λ^*)

$$\nabla_z \mathcal{L}(z^*, \lambda^*) = 0, \tag{2.32a}$$
$$a_i(z^*) = 0, \qquad \forall\, i \in \mathcal{E}, \tag{2.32b}$$
$$a_i(z^*) \leq 0, \qquad \forall\, i \in \mathcal{I}, \tag{2.32c}$$
$$\lambda_i^* \leq 0, \qquad \forall\, i \in \mathcal{I}, \tag{2.32d}$$
$$\lambda_i^* a_i(z^*) = 0, \qquad \forall\, i \in \mathcal{E} \cup \mathcal{I}. \tag{2.32e}$$

2.3.7. Duality

In optimization, *duality* means that optimization problems might be viewed from either of two perspectives, namely the *primal problem* or the *dual problem*. An optimization problem stated on the form as equation (2.30) is called the *primal problem*. The solution to the dual problem gives a lower bound to the solution of the primal problem. The difference, or gap, between the solution of the primal and dual is called the *duality gap*. For convex optimization problems under certain constraint qualifications conditions, the duality gap is zero. This means that value of the solution to the dual problem is equal to the primal problem. This is also called *strong duality*. The main advantage of constructing the dual problem is that it is usually easier to solve computationally.

Dependent on the optimization problem at hand, there are different approaches to construct the dual problem. Common approaches are e.g. the Wolfe dual problem and the Fenchel dual problem. However, when referring to the dual problem, this means the *Lagrangian dual problem* in most cases, and is also the dual approach used in this Thesis.

The Dual Problem

The Lagrangian dual problem is constructed using the Lagrangian function explained in Section 2.3.6. Given the primal problem in equation (2.30), the Lagrangian function is given be equation (2.31). The *dual function* d is then defined as follows:

$$d(\lambda) := \min_{z \in \mathcal{Z}} \mathcal{L}(z, \lambda). \tag{2.33}$$

Furthermore, the *dual problem* is to maximize the dual function, where the optimization variable is given by the Lagrange multiplier λ,

$$\max_{\lambda} d(\lambda) \qquad \text{s.t.} \quad \lambda_i \geq 0, \quad \forall i \in \mathcal{I}. \tag{2.34}$$

Note that calculation of the minimum in equation (2.33) requires finding the *global* minimizer of a function $\mathcal{L}(\,\cdot\,, \lambda)$ for the given λ, which may be a very difficult in practice. However, when the primal objective function f is convex and the constraints c

[2] Linear independence constraint qualification (LICQ) holds if the set of active constraint gradients is linearly independent, see Definition 12.4 [16].

are linear, the function $\mathcal{L}(\,\cdot\,,\lambda)$ is also convex, and the computation of d becomes is easier to solve in practice. Also, note that the Lagrangian multiplier must be equal or greater than zero for all inequality constraints.

Slater's Condition

Slater's condition is a specific example of a constraint qualification, and is a sufficient condition for strong duality to hold for a convex optimization problem. In other words, if Slater's condition holds for the primal problem, then the duality gap is 0, [19].

2.3.8. Augmented Lagrangian Method

The *augmented Lagrangian method*, also known as the *method of multipliers* is a certain class of algorithms for solving constrained optimization problems. They are similar to penalty methods in that they replace a constrained optimization problem by a series of unconstrained problems. The difference is that the augmented Lagrangian adds an additional term to the unconstrained objective. This term is used to mimic a Lagrange multiplier. In other words, the new unconstrained objective is in fact the Lagrangian function of the constrained problem with an additional quadratic penalty term. In this Thesis, the augmented Lagrangian method will only be considered for problems with equality constraints.

General Method: Equality Constraints

Consider the optimization problem:

$$\min_{z\in\mathcal{Z}} f(z) \qquad \text{s.t.} \;\; a_i(z) = 0, \quad \forall i \in \mathcal{E}. \tag{2.35}$$

By moving the constraints into the objective function, and adding the quadratic penalty term, the so-called augmented Lagrangian function is defined as:

$$\mathcal{L}_\rho(z,\lambda,\rho) := f(z) + \sum_{i\in\mathcal{E}} \lambda_i a_i(z) + \frac{\rho}{2}\sum_{i\in\mathcal{E}} a_i^2(z). \tag{2.36}$$

where $\rho > 0$ is referred to as the *penalty parameter*. The augmented *dual problem* is defined as:

$$\max_{\lambda} d_\rho(\lambda), \tag{2.37}$$

where $d_\rho(\lambda) := \min_z \mathcal{L}_\rho(z,\lambda,\rho)$ is the dual function. The augmented Lagrangian method is based on the following: For each iteration:

- for a given ρ_k and λ^k, solve $\min_{z\in\mathcal{Z}} \mathcal{L}_\rho(z,\lambda^k;\rho_k)$.
- Then, update ρ_k and λ^k.

Nocedal and Wright [16] suggest that λ^k should be updated the following way:

$$\lambda_i^{k+1} = \lambda_i^k - \rho_k a_i(z_k). \tag{2.38}$$

2.3.9. Stopping Criteria

In general, optimization problems are unsolvable. Hence, it is not guarantied that an iterative optimization algorithm will find the exact solution. In practice, it is "good enough" to find a solution that is close to the exact optimal solution, i.e an approximation within a given accuracy $\epsilon > 0$. This is also called a *suboptimal* solution. Nesterov and Nesterov [18] defines approximation:

> To approximate means to replace an initial complex object by a simplified one, which is close by its properties to the original.

However, it is not an easy task for the algorithm to know when it has found a good approximate solution. Therefore, it is crucial to have a good stopping criterion. Deciding stopping criteria can be considered it's own topic within optimization, and usually, more than one stopping criterion is used in combination to get the best behaviour of an algorithm. In the algorithms discussed in this Thesis, four different stopping criteria are utilized.

1. **Primal Suboptimality**
   This is one of the most basic stopping criteria and is based on Theorem 2.3.1 ($\nabla f(z^*) = 0$), by comparing the function value of f from the current and previous iterate. The difference must be within some predefined tolerance $\epsilon_{\text{ps}} > 0$.
   $$\left|f(z_k) - f(z_{k-1})\right| \leq \epsilon_{\text{ps}}. \tag{2.39}$$
   Smaller ϵ_{ps} gives a better approximation, but as a consequence the algorithm require more iterations.

2. **Maximum Iterations**
   To prevent the algorithm from running "forever", it is common to put an upper bound on the number of iterations. This criterion is also used when a predictable behaviour of the algorithm is desirable, i.e if it is known that a satisfying solution will be reached within the finite number of iterations.

3. **Primal Feasibility**
   This stopping criterion considers if the iterate is feasible or not. For example, in a problem with equality constraints $a(z) = 0$, all the constraints must be equal to zero for the solution to be feasible. However, since this may be difficult to achieve, the tolerance ϵ_{pf} is introduced. The stopping criterion is as follows:
   $$\|a(z_k)\| \leq \epsilon_{\text{pf}}. \tag{2.40}$$
   When the problem also contains inequalities, an extra operation in form of projection is needed
   $$\left\| \left[a(z_k)\right]_+ \right\| \leq \epsilon_{\text{pf}}. \tag{2.41}$$

4. **Dual Suboptimality**
   This stopping criterion is similar to the first criterion, i.e. primal suboptimality. However, it compares the function value of the dual function d, from equation (2.33).

$$|d(\lambda_k) - d(\lambda_{k-1})| \leq \epsilon_{\text{ds}}. \tag{2.42}$$

2.4. Quadratic Programming

Quadratic Programming (QP) is a special type of optimization problems where the function to be optimized f is quadratic and the constraints are linear. These kind of problems have a widespread use within practical applications. A general QP, subjected to equalities and inequalities, can be states as follows:

$$\begin{aligned} \min_{z\in\mathcal{Z}} \quad & f(z) \quad \left(= \tfrac{1}{2} z^T H z + c^T z\right) \\ \text{s.t.} \quad & a_i^T z = b_i, \qquad i \in \mathcal{E}, \\ & a_i^T z \leq b_i, \qquad i \in \mathcal{I}, \end{aligned} \tag{2.43}$$

where H is a symmetric $n \times n$ matrix, $\mathcal{E}$ and $\mathcal{I}$ are finite sets of indices, and c, z, $\{a_i\}$, $i \in \mathcal{E} \cup \mathcal{I}$, are vectors in $\mathbb{R}^n$. Moreover, the set $\mathcal{Z} = [lb, ub]$ is the simple box-constraint where $lb, ub \in \mathbb{R}^n$. If the Hessian matrix H is positive semidefinite, the QP is called a *convex* QP, and if H is positive definite it is a called *strictly convex* QP. *Non-convex* QPs are usually more challenging to solve, and will not be considered in this Thesis. QPs can always be solved or shown to be infeasible in a finite number of computations [16]. However, the number of computations needed is strongly dependent on the characteristics of the objective function and the number of constraints.

There are a rich variety of algorithms tailored for solving QPs. Deciding which one is the better choice depends of the problem formulation, size, properties, etc. Active set and interior point (barrier) methods are among the most popular methods for solving QPs. These are second order algorithms which solves the primal problem directly. In contrast, in this Thesis, first order algorithms which solves the dual problem that arises from relaxation of the complicated constraints, are considered. These algorithms are presented in the next chapter. However, primal subproblems on the form

$$\min_{z\in\mathcal{Z}} \quad f(z) \quad \left(= \frac{1}{2} z^T H z + c^T z\right), \tag{2.44}$$

without complicated constraints, needs to be solved for every outer iteration of the dual problem. To solve problem on this form, two methods are discussed: *Gradient Descent* and Nesterovs *Fast Gradient*. Gradient descent will only be explained as a step to understand fast gradient, which has a better convergence rate.

2.4.1. Gradient Descent Method

Gradient descent is one of the most basic and simple algorithms in convex optimization theory, and can solve a QP on the form of equation (2.44). The algorithm takes advantage of the well known fact that the antigradient, i.e. $-\nabla f(z)$, is the direction of the locally steepest descent of a differentiable function. Nesterov and Nesterov [18] Algorithm 1.2.9 states the unconstrained case of the gradient descent method. Algorithm **GDM** describes the case where the optimization variable z is constrained by the set $\mathcal{Z}$. The only difference is the projection onto the $\mathcal{Z}$ every iteration.

Algorithm GDM Gradient Descent Method

Require: $z_0 \in \mathcal{Z}$ and $k = 0$

while (stopping criteria not met) **do**

1. $z_{k+1} = [z_k - h_k \nabla f(z_k)]_{\mathcal{Z}}$

end while

In **DGM** the scalar factor h_k represents the step-size. h_k must be positive ($h_k > 0$), and the sequence $\{h_k\}_{k=0}^{\infty}$ is chosen in advance. This sequence is calculated using one out of a number of different schemes, with the simplest scheme being the *constant* step:

$$h_k = h > 0. \tag{2.45}$$

Nesterov [17] proves that if the function f is convex, then the gradient $\nabla f(z_k)$ is Lipschitz continous with a Lipschitz constant L. Furthermore, if L is given by the the largest eigenvalue of the Hessian H, the optimal constant step-size in given by

$$h = \frac{1}{L}. \tag{2.46}$$

As mentioned, Algorithm 1.2.9 [18] describes the unconstrained case, which is equivalent to setting $lb = -\infty$ and $ub = \infty$, in the set $\mathcal{Z}$. In the constrained case, where lb and/or ub exists, z_k is projected onto $\mathcal{Z}$ in every iteration. The projection in **GDM** is very simple to compute , i.e. it consists of two simple min/max statements. The resulting algorithm is as follows:

1. $z_{\text{temp}} = z_k - h_k \nabla f(z_k)$:
2. $z_{\text{temp}} = \min(ub, z_{\text{temp}})$.
3. $z_{k+1} = \max(lb, z_{\text{temp}})$.

This projection has a computational complexity of $\mathcal{O}(2n)$.

Convergence and Stopping Criteria

If the function f is convex and the gradient ∇f is Lipschitz, and it is not assumed that f is strongly convex, then the error in the objective value generated at each step k will be bounded by $\mathcal{O}(1/k)$. Furthermore, the stopping criterion in **GDM** will typically be primal feasibility (stopping criteria 1 from Section 2.3.9), in addition to a bound on maximum number of iterations.

2.4.2. Fast Gradient Method

The fast gradient method was first proposed by Yurii Nesterov in 1983 [17], and is an extension of the gradient descent method. Consequently, the algorithm is also called *Nesterov's gradient method*. The method is described in Algorithm **FGM** and can be found in Nesterov and Nesterov [18] (Algorithm 2.2.9).

Algorithm FGM Fast Gradient Method

Require: $y_0 = z_0 \in \mathcal{Z}$ and $k \geq 0$
while (stopping criteria not met) **do**
1. $z_{k+1} = [y_k - h\nabla f(y_k)]_{\mathcal{Z}}$
2. $y_{k+1} = z_{k+1} + \beta_k(z_{k+1} - z_k)$

end while

As in **GDM**, the step-size is chosen constant as $h = 1/L$, where L is the Lipschitz constant given by the largest eigenvalue of H. However, β_k may vary every iteration. The β described in [18] (Algorithm 2.2.9) is chosen as follows:

$$\beta_k = \frac{\gamma_k(1 - \gamma_k)}{\gamma_k^2 + \gamma_{k+1}} \tag{2.47}$$

where γ_0, $\gamma_k \in (0, 1)$ and

$$\gamma_{k+1}^2 = (1 - \gamma_{k+1})\gamma_k^2 + q\gamma_{k+1}, \tag{2.48}$$

where $q = \mu/L$. The convexity parameter μ is the smallest eigenvalue of the Hessian H. Solving the quadratic equation for γ_{k+1} gives:

$$\gamma_{k+1} = \frac{q - \gamma_k^2 \pm \sqrt{(q - \gamma_k^2)^2 - 4\gamma_k^2)}}{2}. \tag{2.49}$$

Because $\gamma_{k+1} \in (0, 1)$ we always use the solution:

$$\gamma_{k+1} = \frac{q - \gamma_k^2 + \sqrt{(q - \gamma_k^2)^2 - 4\gamma_k^2)}}{2}. \tag{2.50}$$

Hence, no projection is necessary. The variable β_k in equation (2.47) is regarded as a safe choice because f do not have to be strongly convex. However, this expression involves the calculation of a square in every iteration, which is a regarded as an complex operation. Algorithm 2.2.11 in [18] chooses $\gamma_k = \sqrt{\mu/L}$. The resulting β_k is

$$\beta_k = \frac{\sqrt{L} - \sqrt{\mu}}{\sqrt{L} + \sqrt{\mu}}, \qquad \forall k \geq 0. \tag{2.51}$$

This means that the β is kept constant and can be calculated offline. Note that, in theory, this does not work for $\mu = 0$, i.e. problems that are not strongly convex and have one or more eigenvalues equal to zero. When solving the dual problem, the subproblems will not always be strongly convex. It was observed, during real simulations during the project, that convergence was obtained with the constant β, even when the problem was not strongly convex. Nevertheless, β_k form equation (2.47) was chosen in the final implementation of **FGM**.

Convergence and Stopping Criteria

FGM uses the same stopping criteria as **GDM**. Furthermore, if the problem is Lipschitz, it can be proved that the error in the objective value generated at each step k is decreasing with $\mathcal{O}(1/k^2)$. This is a big improvement compared to **GDM** where the error generated is bounded by $\mathcal{O}(1/k)$.

3. Quadratic Dual First Order Optimization Algorithms

In this chapter, four different algorithms for solving convex quadratic programs (QPs) with complicated constraints will be presented. The algorithms are first order and solve the dual problem that arises from Lagrangian relaxation of the complicated constraints. After an introduction, including a problem formulation and some common properties, the different algorithms will be described in dedicated sections. The description should be detailed to the extent where implementation becomes trivial.

The four algorithms reviewed and implemented are called:

- Dual Gradient Method (**DGM**) - Section 3.4
- Dual Fast Gradient Method (**DFGM**) - Section 3.5
- Dual Augmented Lagrangian Method (**ALM**) - Section 3.6
- Dual Fast Augmented Lagrangian Method (**FALM**) - Section 3.7

These algorithms are all described in papers by professor Ion Necoara, see [9], [12], and [13], which also include full theoretical analysis of complexity and convergence.

3.1. Problem Formulation

In the previous chapter, in Section 2.4, the properties of QPs are discussed and a general form, containing equality and inequality constraints, is presented in equation (2.43). Furthermore, in Section 2.1, model predictive control is introduced and a condensed MPC problem is recast as a QP, see equation (2.19). This QP puts constraints on both the states x and inputs u (from equation (2.1)), but uses only the input u as optimization variable. Nevertheless, this problem formulation is very general and practical for MPC applications. By substituting u in (2.19) with z the QP is reformulated as

$$\begin{aligned} \min_{z} \quad & f(z) \quad \left(= \tfrac{1}{2} z^T H z + c^T z\right) \\ \text{s.t.} \quad & \hat{lb} \leq Gz - g \leq \hat{ub} \\ & lb \leq z \leq ub, \end{aligned} \tag{3.1}$$

where $H \in \mathbb{R}^{n \times n}$ is the Hessian, $G \in \mathbb{R}^{m \times n}$ is a matrix for the linear constraints, and $c, lb, ub \in \mathbb{R}^n$ and $g, \hat{lb}, \hat{ub} \in \mathbb{R}^m$ are column vectors. Equation (3.1) will be referred to as the *primal problem* and f as the *primal objective function*. Furthermore, this formulation gives different possibilities when deciding the bound value for the linear

constraints $\hat{lb}$ and $\hat{ub}$. Four different scenarios are created with respect to how $\hat{lb}$ and $\hat{ub}$ are chosen:

- **Case 1:** $\hat{lb} \neq \hat{ub}$.
  Resulting constraint: $\hat{lb} \leq Gz - g \leq \hat{ub}$.
- **Case 2:** $\hat{lb} = \hat{ub}$.
  Resulting constraint: $Gz = [g + \hat{ub}]$. (equality case)
- **Case 3:** $\hat{lb} = -\infty$ and $\hat{ub} \in \mathbb{R}^n$.
  Resulting constraint: $Gz \leq [g + \hat{ub}]$.
- **Case 4:** $\hat{lb} \in \mathbb{R}^n$ and $\hat{ub} = \infty$.
  Resulting constraint: $[g + \hat{lb}] \leq Gz$.

DGM and **DFGM** will be able to solve all four cases, while **ALM** and **FALM** are restricted to the case with equalities (case 2).

The optimization variable constraints lb and ub can be bounded or unbounded, regardless of the four cases. The totally *unconstrained* case is when $\hat{lb} = lb = -\infty$ and $\hat{ub} = ub = \infty$. For this case **FGM** can be applied directly.

3.2. Outer and Inner Problem

All the algorithms will utilize Lagrangian relaxation to move the linear constraints into the objective function and create a dual problem. To make the discussion clearer, it is distinguished between an *outer* and an *inner problem*. The Lagrangian multipliers in λ are updated in the outer problem, while the optimization variables in z are updated in the inner problem. The inner problem will have the same structure as the QP in equation (2.44). However, the matrix H and vector c may vary from every iteration in the outer loop, and is therefore denoted $\hat{H}$, and $\hat{c}$. Consequently, the inner problem which changes every outer iteration is defined as

$$\min_{z \in \mathcal{Z}} \quad \frac{1}{2} z^T \hat{H} z + \hat{c}^T z. \tag{3.2}$$

All the four algorithms use **FGM** from Section 2.4.2 to solve the inner problem, see Figure 3.1 . To summarize, for every iteration in the outer problem, the inner problem is called to minimize a problem on the from of equation (3.2). In Chapter 5 it is shown that the number of iterations for solving the inner problem is linked directly to the runtime of each algorithm.

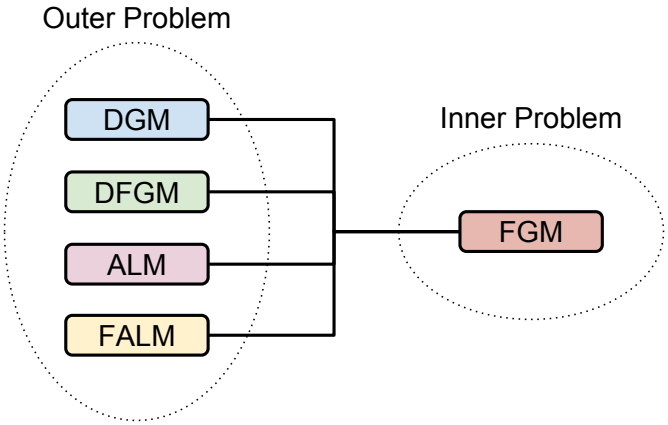


Figure 3.1: Overview of the algorithms and dependence of **FGM**.

3.3. Last Primal Iterate and Average Primal

Necoara and Patrascu [13] makes an iteration complexity analysis for **DGM** and **DFGM** which is based on two types of approximate primal solutions: the last primal iterate sequence $(v_k)_{k\geq 0}$ defined as

$$v_k = \arg\min_{v\in\mathcal{Z}} \mathcal{L}(v, \lambda_k), \tag{3.3}$$

and an average primal sequence $(\hat{z}_k)_{k\geq 0}$ of the form

$$\hat{z}_k = \frac{\sum_{j=0}^{k} \sigma_j z_j}{S_k}, \qquad \text{with} \quad S_k = \sum_{j=0}^{k} \sigma_j. \tag{3.4}$$

Note that the value of σ_j is chosen differently between the algorithms. The general assumption is that the average primal sequence has a better performance. However, simulations done in this Thesis shows that the average might not always be the better choice. The algorithms **ALM** and **FALM** are also analysed for both the last iterate and in the average, see [9]. Theoretical upper bounds of convergence, and accuracy estimates for all the algorithms are given with respect to the average and is summarized in Appendix A.

All the algorithms are implemented in the toolbox *DuQuad*, explained in Chapter 4, and the last and average are considered for all four algorithms. The result is 8 different scenarios. In addition, when including the four constraints cases (from how the linear constraints are chosen) the total amount of different scenarios grows to 20. This is illustrated in Figure 3.2.

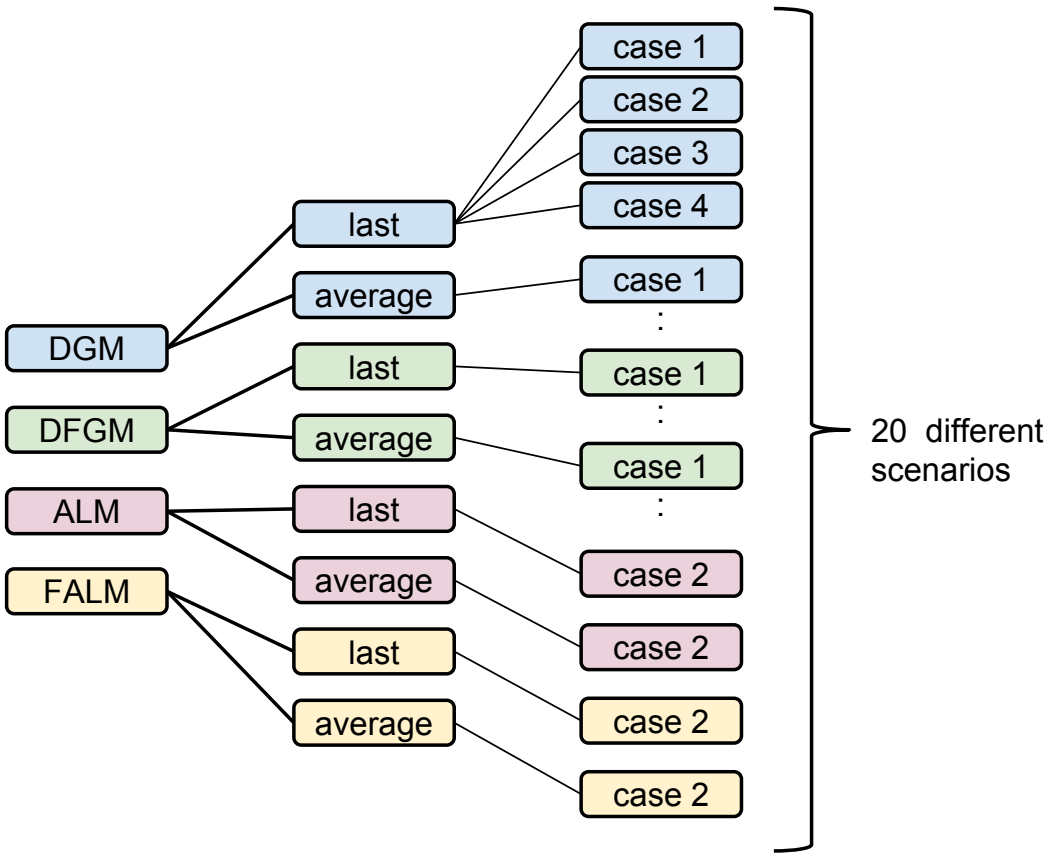


Figure 3.2: Overview of the algorithms cases.

3.4. Dual Gradient Method (DGM)

The first algorithm that is reviewed is called *Dual Gradient Method* (**DGM**) and is described in Necoara and Nedelcu [12] and Necoara and Patrascu [13]. The two main steps are shown in Algorithm **DGM**.

Algorithm DGM Dual Gradient Method

Require: $z_0 \in \mathcal{Z}$, $\lambda_0 \in [0, 1]$, and $k \geq 0$
while (stopping criteria not met) **do**

1. $z_k = \arg\min_{z \in \mathcal{Z}} \mathcal{L}(z, \lambda_k)$ ▷ Inner Problem
2. $\lambda_{k+1} = \left[\lambda_k + \alpha \nabla d(\lambda_k)\right]_+$

end while

Figure 3.3 gives an overview of how the algorithm iterates. In short, the primal problem in equation (3.1) is given as input to **DGM**, which construct the dual problem from Lagrangian relaxation of the complicated constraints. Then **DGM** start a while-loop which update the Lagrangian multiplier λ every iteration. Furthermore, for every outer iteration, the optimization variable z is updated with respect to λ. This update is the result of the minimization of the Lagrangian function, and is called the inner problem. The inner problem is solved by **FGM** from Section 2.4.2. After each z update, the stopping criteria are checked. If these are fulfilled, the algorithm terminates and z and λ from the last iteration are given as outputs.

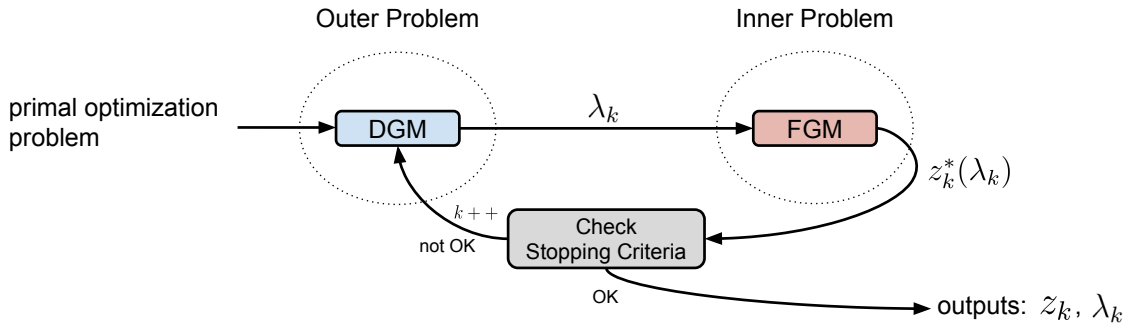


Figure 3.3: DGM.

A more detailed derivation of **DGM** is given in the following. It is assumed that $\hat{lb}$ and $\hat{ub}$ from the primal problem (3.1) meets the criteria of case 1, from Section 3.1. The resulting linear constraints is then given by

$$\hat{lb} \leq Gz - g \leq \hat{ub}, \tag{3.5}$$

which can be rearranged as

$$\begin{aligned} Gz - g - \hat{ub} &\leq 0, \quad &| \lambda^1 \geq 0 \\ -Gz + g + \hat{lb} &\leq 0, \quad &| \lambda^2 \geq 0, \end{aligned} \tag{3.6}$$

where $\lambda^1, \lambda^2 \in \mathbb{R}^m$ are the corresponding Lagrangian multiplier vectors. By moving these constraints into the objective function, the Lagrangian function is constructed as follows:

$$\mathcal{L}(z, \lambda^1, \lambda^2) := \frac{1}{2} z^T H z + c^T z + \left\langle \lambda^1, Gz - g - \hat{ub} \right\rangle + \left\langle \lambda^2, -Gz + g + \hat{lb} \right\rangle. \quad (3.7)$$

The corresponding dual function is defined as

$$d(\lambda^1, \lambda^2) := \min_{z \in \mathcal{Z}} \mathcal{L}(z, \lambda^1, \lambda^2). \quad (3.8)$$

Constant terms are redundant when minimizing the Lagrangian function. Consequently, the minimization problem can be rewritten as

$$\min_{z \in \mathcal{Z}} \quad \frac{1}{2} z^T H z + \left(c + G^T \lambda^1 - G^T \lambda^2\right)^T z + \text{constant}, \quad (3.9)$$

and by defining

$$\hat{c} := c + G^T \lambda^1 - G^T \lambda^2, \quad (3.10)$$

where $\hat{c} \in \mathbb{R}^n$, the resulting dual function is rewritten as

$$\hat{d}(\lambda^1, \lambda^2) = \min_{z \in \mathcal{Z}} \quad \frac{1}{2} z^T H z + \hat{c}^T z. \quad (3.11)$$

$\hat{d}$ has the same form as equation (3.2) and is the inner problem. The solution to the inner problem is defined as the argument $z^*(\lambda^1, \lambda^2)$.

The Dual Problem

By defining $\lambda = [\lambda^1, \lambda^2]^T$, the task of maximizing the dual function with respect to λ is called the *dual problem* and defined by

$$\boxed{\max_{\lambda \in \mathbb{R}^{2m}_+} d(\lambda).} \quad (3.12)$$

For every outer iteration in **DGM**, λ^1 and λ^2 are updated in the following way:

$$\begin{pmatrix} \lambda^1_{k+1} \\ \lambda^2_{k+1} \end{pmatrix} = \left[\begin{pmatrix} \lambda^1_k \\ \lambda^2_k \end{pmatrix} + \alpha \nabla d(\lambda^1_k, \lambda^2_k) \right]_+ \quad (3.13)$$

where the scalar α is the constant step size and the gradient is given by

$$\nabla d(\lambda^1_k, \lambda^2_k) = \begin{bmatrix} Gz^*(\lambda^1_k, \lambda^2_k) - g - \hat{ub} \\ -Gz^*(\lambda^1_k, \lambda^2_k) + g + \hat{lb} \end{bmatrix}. \quad (3.14)$$

Note that the gradient in equation (3.14) is calculated from the dual function in equation (3.8) which includes the constant term. Because $\lambda \geq 0$, the update of λ is projected on the non-negative orthant $\mathbb{R}^{2m}_+$.

Necoara and Nedelcu [12] proves that the gradient $\nabla d(\lambda_k^1, \lambda_k^2)$ is Lipschitz continuous with a Lipschitz constant given by

$$L_{\mathrm{d}} := \frac{\| [G, -G]^T \|^2}{\mu}, \tag{3.15}$$

where μ is the convexity parameter given by the smallest eigenvalue of H. The constant step-size α is then defined as

$$\alpha := \frac{1}{2L_{\mathrm{d}}}. \tag{3.16}$$

Stopping Criteria

In addition to stopping criterion 2 (maximum number of iterations) from Section 2.3.9, criteria 3 and 4 are also utilised in **DGM**. Criterion 3, dual suboptimality, is defined as

$$\left| d(\lambda_k^1, \lambda_k^2) - d(\lambda_{k-1}^1, \lambda_{k-1}^2) \right| \leq \epsilon_{\mathrm{ds}}, \tag{3.17}$$

and criterion 4, primal feasibility, becomes

$$\left\| \begin{bmatrix} Gz_k - g - \hat{u}b \\ -Gz_k + g + \hat{l}b \end{bmatrix}_+ \right\| \leq \epsilon_{\mathrm{pf}}. \tag{3.18}$$

Both of these stopping criteria must be satisfied for the algorithm to finish.

Last Primal Iterate and Average Primal

The only difference between the last primal iterate and the average primal in **DGM** is how to calculate the primal feasibility and the output. The last primal iterate v_k from equation (3.3) is always equal to z_k, i.e. $z_k = v_k, \forall\, k \geq 0$. However, in the average case, z_k is substituted with $\hat{z}_k$ in equation (3.18). The average primal $\hat{z}_k$ is calculated from equation (3.4) with $\sigma_j = 1$, $\forall j \geq 0$, resulting in

$$\hat{z}_k = \frac{1}{k+1} \sum_{j=0}^{k} z_j. \tag{3.19}$$

Output

When the algorithm terminates an approximate solution is found. The approximate optimal solution/point is given by

$$z^* = v_k, \tag{3.20}$$

for the last primal iterate, and

$$z^* = \hat{z}_k, \tag{3.21}$$

in the average primal sequence. The primal approximate optimal value then becomes

$$f^* = f(z^*). \tag{3.22}$$

The Different Cases

If $\hat{lb}$ or $\hat{ub}$ are unconstrained, it will result in the corresponding $\lambda = 0$. For example

$$\hat{ub} = \infty \quad \Rightarrow \quad \nabla d(\lambda_k^1) = -\infty \quad \Rightarrow \quad \lambda_{k+1}^1 = [-\infty]_+ = 0 \tag{3.23}$$

This entails that some of the computations in the algorithm can be omitted, which results in less complex iterations. The four different cases of the linear constraints are described in Section 3.1, and there is not much that needs to be changed from one case to the next. Nevertheless, the important changes are pointed out in the following.

First of all, the constant L_d from equation (3.15) can only be applied in case 1. In all the other cases

$$L_\mathrm{d} := \frac{\| G \|^2}{\mu}. \tag{3.24}$$

Case 1

This is the case where the linear constraints are bounded by both $\hat{lb}$ and $\hat{ub}$, and is described in detail above.

Case 2

In the equality case, the term with λ^2 is dropped, since $\lambda^1 = \lambda^2$. The Lagrangian function is then defined as

$$\mathcal{L}(z, \lambda^1) := \frac{1}{2} z^T H z + c^T z + \left\langle \lambda^1, Gz - g - \hat{ub} \right\rangle \tag{3.25}$$

Furthermore, projection is not necessary when there are no inequalities, and the resulting λ update is

$$\lambda_{k+1}^1 = \lambda_k^1 + \alpha \nabla d(\lambda_k^1). \tag{3.26}$$

Moreover, when calculating the stopping criterion for primal feasibility, the expression with $\hat{lb}$ is omitted and projection is not necessary. This gives

$$\left\| Gz - g - \hat{ub} \right\| \leq \epsilon_\mathrm{pf}. \tag{3.27}$$

Case 3

The Lagrangian function will be equal to the function in case 2. However, λ_{k+1}^1 in (3.26) and the vector in (3.27) must be projected on the positive set $\mathbb{R}_+^m$ in every outer iteration.

Case 4

In the fourth case λ^1 will always be equal to zero. Therefore, the Lagrangian function is defined for λ^2 as

$$\mathcal{L}(z, \lambda^2) := \frac{1}{2} z^T H z + c^T z + \left\langle \lambda^2, -Gz + g + \hat{lb} \right\rangle. \tag{3.28}$$

The λ update becomes

$$\lambda_{k+1}^2 = \left[\lambda_k^2 + \alpha \nabla d(\lambda_k^2) \right]_+, \tag{3.29}$$

and the primal feasibility is stated as follows:

$$\left\| \left[-Gz + g + \hat{lb} \right]_+ \right\| \leq \epsilon_\mathrm{pf}. \tag{3.30}$$

3.5. Dual Fast Gradient Method (DFGM)

The *Dual Fast Gradient Method* **DFGM**, presented by Necoara and Nedelcu [12] and Necoara and Patrascu [13], is the second algorithm that is discussed in this chapter. The algorithm is an an extension of Nesterov's optimal gradient method [18], and is also considered an extended variant of the algorithm **DGM** that was discussed in the previous section.

Algorithm DFGM Dual Fast Gradient Method

Require: $z_0 \in \mathcal{Z}$, $\lambda_0 = y_1 \in [0, 1]$, and $k \geq 1$

while (stopping criteria not met) **do**

1. $z_k = \arg\min_{z \in \mathcal{Z}} \mathcal{L}(z, y_k)$ ▷ Inner Problem
2. $\lambda_k = \left[y_k + \alpha \nabla d(y_k) \right]_+$
3. $y_{k+1} = \lambda_k + \frac{\theta_k - 1}{\theta_{k+1}} (\lambda_k - \lambda_{k-1})$

end while

DFGM updates two dual sequences (λ_k and y_k) and one primal sequence z_k. The algorithm also includes the parameters α and θ. Normally, the step-size α would change every iteration, but in this analysis α is kept constant and is chosen the same way as in **DGM**, see equation (3.16). The parameter θ is defined as:

$$\theta_{k+1} := \frac{1 + \sqrt{1 + 4\theta_k^2}}{2}, \qquad \text{with} \quad \theta_1 = 1. \tag{3.31}$$

Note that if $\theta_k = 1, \ \forall\, k \geq 1$, the resulting algorithm reduces to **DGM**.

The stopping criteria used in **DFGM** are the same as in **DGM**, see Section 3.4. However, when calculating dual suboptimality, v_k from equation (3.3) is used as input to the dual function instead of z_k. In practice this means that the inner problem is calculated twice for every outer iteration: one time with y_k as input and one time with λ_k as input.

DFGM is also implemented to handle all the four cases for the linear constraints. However, the changes in the algorithm are the same as described for **DGM** in Section 3.4.

Last Primal Iterate and Average Primal

Primal feasibility is calculated as in equation (3.18). However, for the last primal iterate, z_k is substituted with v_k from equation (3.3), and in the average case, z_k is substituted with $\hat{z}$, which is given by equation (3.4) with $\sigma_j = \theta_j$, giving

$$\hat{z}_k = \frac{\sum_{j=0}^{k} \theta_j z_j}{S_k^\theta}, \qquad \text{with} \quad S_k^\theta = \sum_{j=0}^{k} \theta_j. \tag{3.32}$$

3.6. Dual Augmented Lagrangian Method (ALM)

The theory of augmented Lagrangian is discussed previously in Section 2.3.8. In this section a dual augmented Lagrangian method (**ALM**), presented in Nedelcu et al. [9], is discussed. Although there are possible to utilize augmented Lagrangian on problems with inequalities, only the case with equalities, i.e case 2 from Section 3.1 is considered in this Thesis.

Algorithm ALM Dual Augmented Lagrangian Method

Require: $z_0 \in \mathcal{Z}$, $\lambda_0 \in [0, 1]$, $\rho > 0$, and $k \geq 0$

while (stopping criteria not met) **do**

1. $z_k = \arg\min_{z \in \mathcal{Z}} \mathcal{L}_\rho(z, \lambda_k, \rho)$ ▷ Inner Problem
2. $\lambda_{k+1} = \lambda_k + \alpha \nabla d_\rho(\lambda_k)$

end while

By setting $g = g + \hat{ub}$ in the general QP from equation (3.1), the resulting QP (case 2) is redefined as

$$\min_{z \in \mathcal{Z}} \quad f(z) \quad \left(= \frac{1}{2} z^T H z + c^T z \right) \qquad \text{s.t.} \quad Gz = g. \tag{3.33}$$

The augmented Lagrangian function is constructed by moving the complicated constraints into the objective function

$$\mathcal{L}_\rho(z, \lambda, \rho) := \frac{1}{2} z^T H z + c^T z + \lambda^T (Gz - g) + \frac{\rho}{2} \|Gz - g\|^2 . \tag{3.34}$$

Furthermore, the dual function is defined as

$$d_\rho(\lambda) := \min_{z \in \mathcal{Z}} \mathcal{L}_\rho(z, \lambda, \rho), \tag{3.35}$$

where the optimal $z^*(\lambda, \rho)$ is obtained. The quadratic term in equation (3.34) can be expressed as:

$$\frac{\rho}{2} \|Gz - g\|^2 = \frac{\rho}{2} z^T (G^T A) z - \rho G^T b z + \text{constant}. \tag{3.36}$$

It follows that the Lagrangian function can be rewritten as

$$\mathcal{L}_\rho(z, \lambda, \rho) = \frac{1}{2} z^T (H + \rho G^T A) z + \left(c + G^T \lambda - \rho G^T g \right)^T z, \tag{3.37}$$

where the constant term is omitted. By defining

$$\hat{H} := H + \rho G^T G, \tag{3.38}$$

and

$$\hat{c} := c + G^T \lambda - \rho G^T g, \tag{3.39}$$

it is clear that the simplified dual function can be expressed on the standard form

$$\hat{d}_p(\lambda) := \min_{z \in \mathcal{Z}} \frac{1}{2} z^T \hat{H} z + \hat{c}^T z, \tag{3.40}$$

which can be solved by **FGM** from Section 2.4.2.

The Dual Problem

As in **DGM** and **DFGM** the goal is to maximize the dual function with respect to λ. The dual augmented Lagrangian problem is stated as follows:

$$\boxed{\max_{\lambda \in \mathbb{R}^m_+} \quad d_\rho(\lambda).} \tag{3.41}$$

For every outer iteration the λ update is done the following way:

$$\lambda_{k+1} = \lambda_k + \alpha \nabla d_\rho(\lambda_k), \tag{3.42}$$

where the gradient is defined as

$$\nabla d_\rho(\lambda_k) := Gz^*(\lambda_k) - g. \tag{3.43}$$

Note that the gradient in equation (3.43) is calculated from the dual function in equation (3.35) that includes the constant term. Also, note that no projection is done when updating λ since the primal problem does not have inequalities constraints.

The gradient $\nabla d(\lambda_k)$ is Lipschitz continuous with a Lipschitz constant [9], given by

$$L_\mathrm{d} := \rho^{-1}, \qquad \rho > 0. \tag{3.44}$$

The scalar α in equation (3.42) is the constant step size and is defined as

$$\alpha := \frac{1}{L_\mathrm{d}}. \tag{3.45}$$

Even though the penalty parameter ρ should be able to change every outer iteration, this Thesis restrict its analysis to the case where ρ is kept constant.

FGM is utilized to solve the inner problem given by equation (3.40), and uses the smallest eigenvalue of the Hessian $\hat{H}$ to decide the step size h in equation (2.46). If ρ is changing every outer iteration, new eigenvalues must be calculated every time the inner problem is solved. This is a very expensive computation. It is therefore preferred to keep ρ constant, which allow for the Hessian $\hat{H}$ to be computed offline. Furthermore, the last term in $\hat{c}$ ($\rho G^T g$) is also constant and can be computed offline. In other words, $\hat{H}$ is kept constant every time the inner problem is solved, while $\hat{c}$ is changing every outer iteration when a new update of λ is available.

The last primal iterate and the average primal sequence are calculated equally as in algorithm **DGM**, see Section 3.4.

Stopping Criteria

The stopping criteria are the same as in **DGM** and **DFGM**, i.e. criteria 2, 3, and 4 from Section 2.3.9. Dual suboptimality is defined as

$$\left| d_\rho(\lambda_k) - d_\rho(\lambda_{k-1}) \right| \leq \epsilon_\mathrm{ds}. \tag{3.46}$$

and primal feasibility is defined as

$$\left\| Gz - g \right\| \leq \epsilon_\mathrm{pf} \tag{3.47}$$

3.7. Dual Fast Augmented Lagrangian Method (FALM)

The Dual Fast Augmented Lagrangian method (**FALM**) is presented and analysed in Nedelcu et al. [9]. In short, **FALM** is an extension of **ALM** identically to how **DFGM** is an extension of **DGM**. By following the derivations in Section 3.4, 3.5 and 3.6, it becomes redundant to derive this algorithm in detail.

Algorithm FALM Dual Fast Augmented Lagrangian Method

Require: $z_0 \in \mathcal{Z}$, $\lambda_0 = y_0 \in [0, 1]$, $\rho > 0$, and $k \geq 1$

while (stopping criteria not met) **do**

1. $z_k = \arg\min_{z \in \mathcal{Z}} \mathcal{L}_\rho(z, y_k, \rho)$ ▷ Inner Problem
2. $\lambda_k = y_k + \alpha \nabla d_\rho(y_k)$
3. $y_{k+1} = \lambda_k + \frac{\theta_k - 1}{\theta_{k+1}} (\lambda_k - \lambda_{k-1})$

end while

FALM solves the primal problem in equation (3.33). The other functions and variables are found from the following equations:

- $\nabla d_\rho(\,\cdot\,)$: equation (3.43)
- $\mathcal{L}_\rho(\,\cdot\,)$: equation (3.34)
- α: equation (3.45)
- θ: equation (3.31)

3.8. Theoretical Properties and Algorithm Commonalities

3.8.1. Warm-Start

All the algorithms benefits from warm-start when solving the inner problem. This means that when finding $z_k^*(\lambda_k)$, $z_{k-1}^*(\lambda_k)$ is given as input to **FGM**. This input serves as an initial point from where **FGM** start the search for a new optimal point. The assumption is that the previous optimal point will be close to the new optimal point, and consequently the number of iterations for finding this new point will be less than if the algorithm operates with a random initial point.

3.8.2. Iteration Complexity and Convergence

Since all the algorithms utilize **FGM** to solve the inner problem, the difference in the complexity is how the they solve the outer problem. From derivation of the algorithms, it is clear that implementation can be done using only simple arithmetic operations, i.e. addition, multiplication, etc. However, **DFGM** and **FALM** includes a square root operation which needs to be solved every outer iteration in order to compute the θ update, see equation (3.31). This is a comlex operation which can be problematic when running the program on low-cost HW. Furthermore, these two algorithms need to solve the inner problem twice every outer iteration, which makes the outer iteration at least twise as complex as for **DGM** and **ALM**. On the other hand, **DFGM** and **FALM** have a faster converges rate in theory.

In Chapter 5 it is shown that runtime of obtaining a solution has a direct link to the total number of iterations used to solve the inner problem. In Appendix A, theoretical upper bounds of convergence, and accuracy estimates are stated for all four algorithms.

4. Optimization Toolbox: DuQuad

In this chapter an optimization toolbox, that is referred to as *DuQuad*, is discussed. DuQuad embeds all the algorithms presented in Chapter 3 and was created to test and compare the performance of the algorithms. DuQuad has also proven to be a versatile tool for solving convex quadratic programs. A general overview of the toolbox, i.e. structure and workflow, is given in the first part of the chapter. In the second part, implementation challenges and optimization strategies are discussed.

A user manual for DuQuad is included in Appendix C, and a public project website [1] is created at `http://sverrkva.github.io/duquad/`.

4.1. Overview

Figure 4.1 shows how all the four algorithms from Chapter 3, i.e. **DGM**, **DFGM**, **ALM**, and **FALM**, are bundled into a toolbox to make the program called DuQuad.

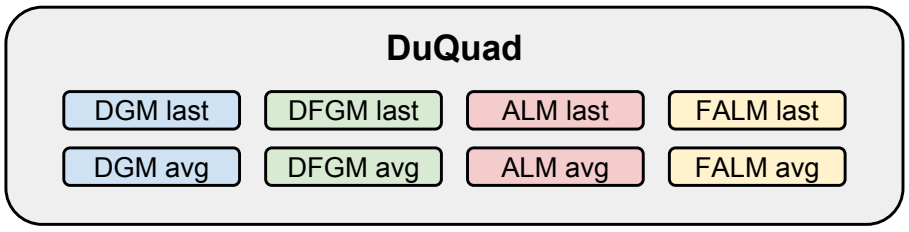


Figure 4.1: DuQuad Algorithms

Note that all four algorithms are able to utilize either of two approaches, with respect to the optimization variable. The two different approaches are the last primal iterate sequence and an average primal sequence. These are referred to as *last* and *avg*. The difference between these two methods was presented in Section 3.3. All the algorithms solve the primal problem in equation (3.1). However, the augmented methods **ALM** and **FALM** can only solve the case with linear equalities constraints (case 2, see Section 3.1). The user can specify which algorithm to utilise when running DuQuad.

All the algorithms are written in the programming language C and are optimized with respect to speed and memory properties. The C-code is implemented using only basic arithmetic and logical operations, e.g. $+, -, *, >$. The only complex operation is the computation of the square root, which is used in computing θ in equation (3.31) and β in equation (2.47). Furthermore, to be able to test and analyse the different algorithms properly and efficiently, DuQuad has a MATLAB interface to the C-code.

4.1.1. Implementation

The class diagram in Figure 4.2 gives an overview of the C-code structure of the toolbox. Basically, there are eight different modules that are placed in separate source files. Each module has a set of functions that can be either public or private. Public functions can be accessed by other modules, while private functions are only available in the module where they are declared.

The four different algorithms each have their own source file. From the class diagram it is shown that they all have one public function in addition to private functions. The public function takes one input in the form of a struct containing all necessary information, variables, and vectors that the algorithm needs to execute. The input-struct is tailored for each algorithm, i.e. it does not contain any unnecessary information.

The program starts in the main-module 'main.c', which take the optimization problem in equation (3.1) as input. The main function have private functions for initializing and allocating the data that is sent as input when it calls one of the four algorithm modules.

After the initialization, 'main.c' calls one of the four functions, i.e. DGM, DFGM, ALM, or FALM. One of the input parameters to these functions is whether the solution should be found from the last primal iterate of the average primal sequence.

All the algorithm modules have functions for allocating the memory needed to solve the inner problem (FGM). Both the main module and the four algorithm modules utilize the functions in the module 'general_functions.c' for allocating and freeing memory.

Furthermore, all the algorithm modules, including 'fgm.c', take advantage of the math library module 'math_functions.c'. This library contains optimized math functions, e.g. general matrix and vector operations.

Throughout the project, the C-code was commented on a format such that a program called Doxygen could be utilised to generate detailed code documentation from the comments. The documentation is on a html format, and can be used to get an overview of the code without reading the source files directly. This documentation is found at the website [20]: `http://sverrkva.github.io/duquad_doc_ccode/` .

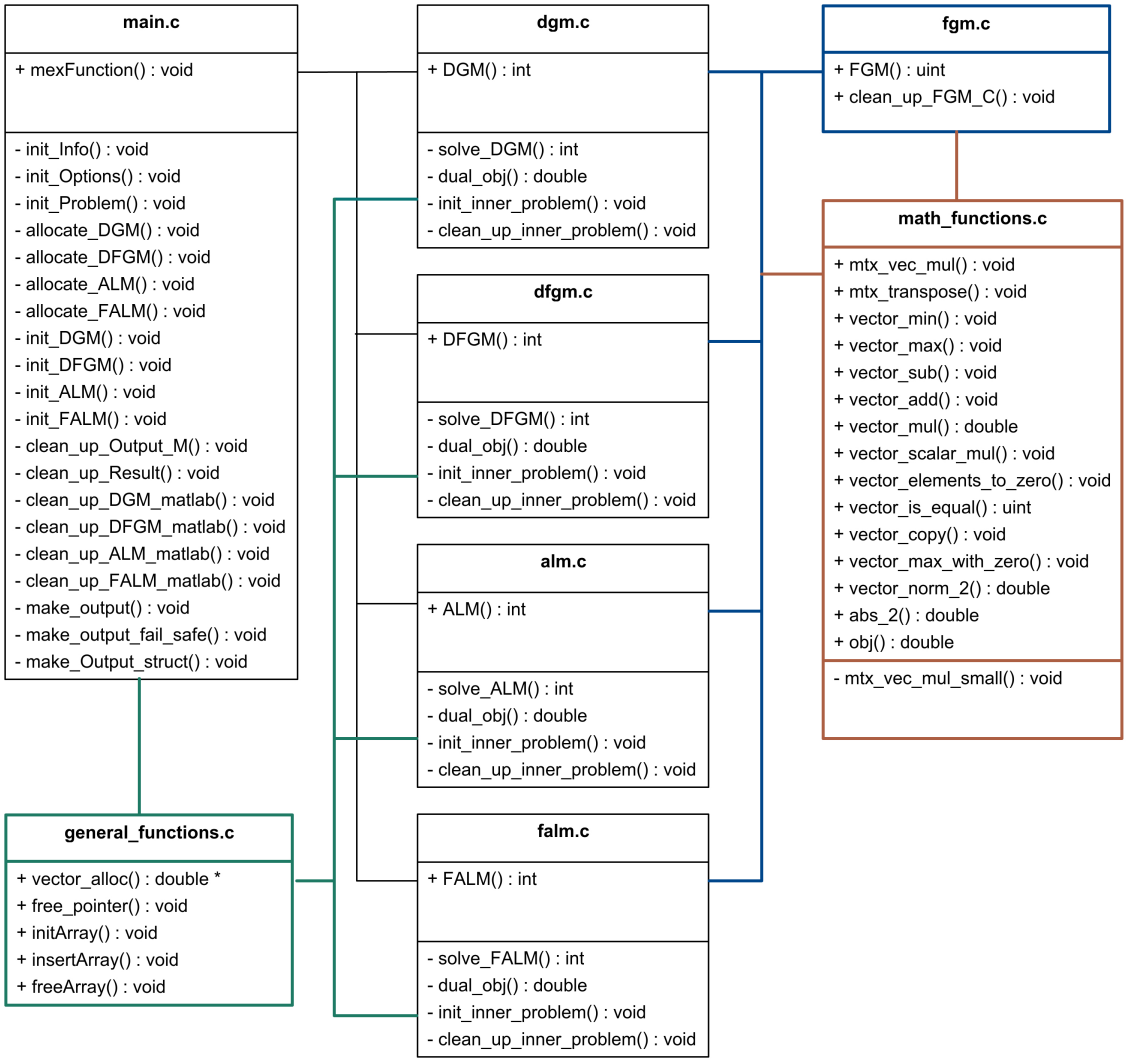


Figure 4.2: Class diagram of DuQuad. '+' means public function, '−' means private function, and the lines symbolize module dependencies.

4.1.2. MEX Framework

To create a MATLAB interface to the C-code, the MEX framework is utilized. MEXs stands for MATLAB Executable. The MEX framework includes a library with functions for converting data between MATLAB and C. A C-file, which includes the MEX-function **mexFunction** and the required link to MEX libraries, serves as an entry point to the code. Figure 4.2 shows that the MEX-function is included as the public function in main.c.

When using the MEX framework to compile the C-program, a binary executable file is created. This file will have the same name as the C-file that contains the MEX-function, including some file extension that is dependent on the operating system. For example, when compiling the code on a Linux 64-bit operating system, the resulting binary file will be called main.mexa64. This binary file is called a MEX-file and can be run as any other MATLAB function from within the MATLAB environment, i.e. by calling the name of the file without the file extension. A binary MEX-file compiled on one operating system will not work on another, i.e. if compiled in Linux it can not be used in Windows.

To summarize, after using the MEX framework to compile the C-code from Figure 4.2, the resulting binary file can be called as any other MATLAB function with associated inputs and outputs.

4.1.3. Program Workflow

An overview of the workflow in DuQuad is illustrated by an example in Figure 4.3.

A QP problem is constructed using a simple MATLAB script called test.m. Then the function *duquad* is called with the problem as input. The MATLAB file 'duquad.m' which implements the function *duquad* is regarded as a pre-prepossessing stage for the online optimization. In this function some complex computations are made. These are regarded as offline computations, i.e. computations that would have been done before the algorithm attempts to solve the optimization problem in a real application. The most important tasks of 'duquad.m' are:

- To check that the input is correct.
- To decide the problem case. (see Section 3.1).
- To set all default values that are not provided by the user.
- To compute the eigenvalues of the Hessian matrix.
- To compute the Lipschitz constant L_{d}.

When all necessary computations are made, the binary MEX-file is called, with the original problem and the extra info as input. The 'main.c' file of the C-code includes the MEX framework and is able to convert the MATLAB data into C format. Furthermore, the converted data gets bundled into a C struct and passed as input to the algorithm that will solve the problem. In Figure 4.3, the algorithm DGM is used as an example.

DGM attempts to solve the problem. It uses the module 'general_functions.c' to allocate all necessary variables for the inner problem, it uses the math library for calculations, and it uses the module 'fgm.c' to solve the inner problem. The result of the optimization is sent back to the MEX-function in 'main.c', where it is converted back to MATLAB formate.

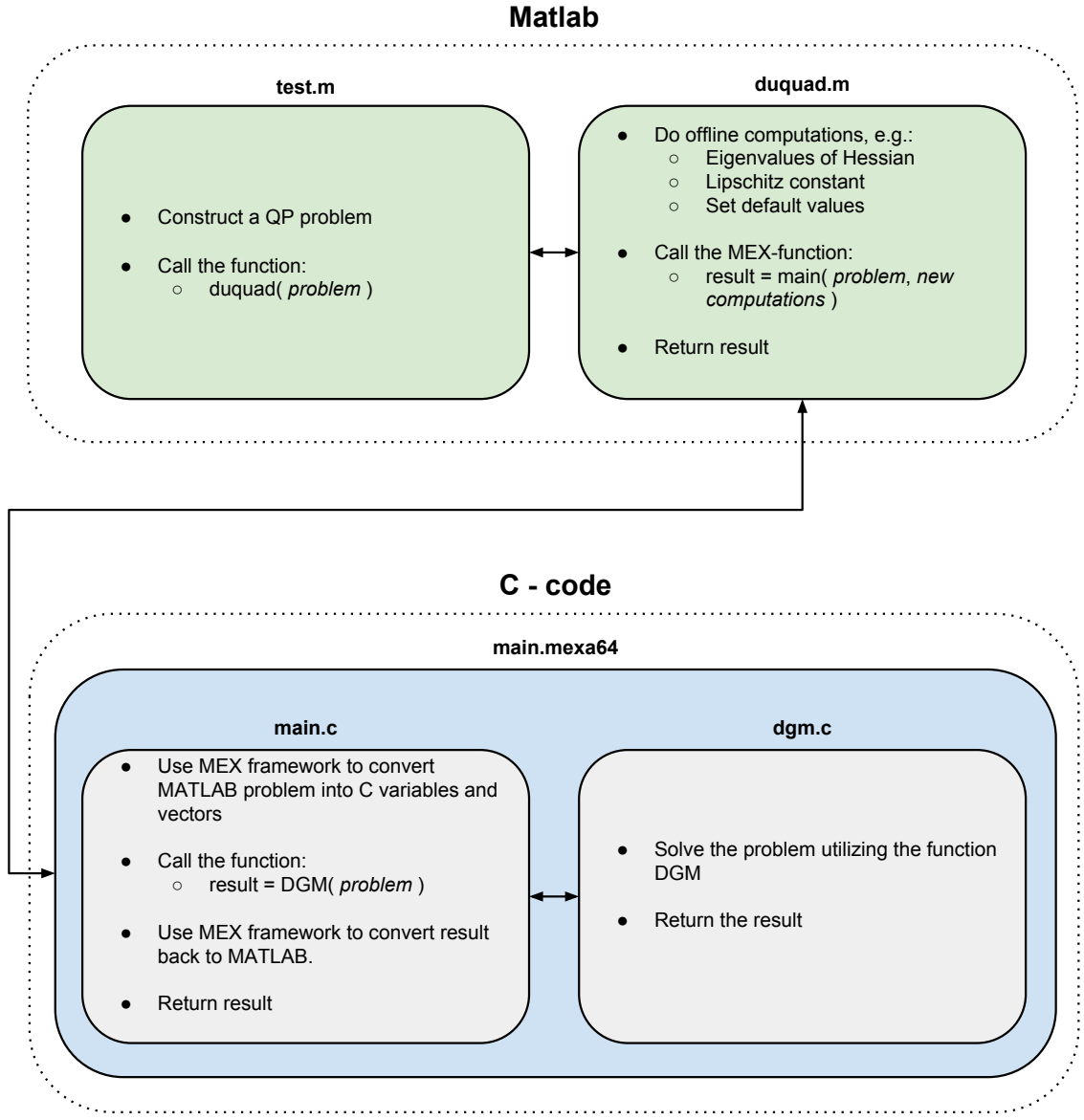


Figure 4.3: DuQuad workflow

4.2. Implementation Challenges and Optimization

During the implementation of the algorithms and the DuQuad toolbox, one of the main objectives was to make the code as optimized as possible with regards to fast execution runtime and low memory usage. However, the code should also be kept readable and maintainable. As a consequence, tradoffs between these different properties were constantly considered throughout the project.

4.2.1. Speed and Runtime

The main priority is to make the program run as fast as possible. This is mainly accomplished by making the complexity of each iteration in the algorithm as small as possible. Different measures were taken in the development process to accomplish this, and the most important are summarized.

- All the data or parameters that can be kept constant during the solving process are computed offline.
- All the matrices and vectors are stored as single pointers. The advantage of using single pointers instead of double pointers is that they yield faster reading and writing to memory when doing large computations. The reason for this is that all the elements in the matrix are stored one after another in one place in memory.
- All the functions in the program take a pointer to an vector instead of the vector itself, as input. Furthermore, if the vector is not altered during execution of the function, the input is declared as **const**.
- Wherever possible, multiplication is used instead of division. Division is a much more complex computation in terms of number of clock-cycles used to perform the operation. Division is typically twice as slow as addition and multiplication.
- No computations are done more than once every iteration. Throughout the implementation it is carefully made sure that no computation is done unnecessary. Especially the result of a matrix multiplication, which is a very time-consuming operations, is stored and reused if used more than once per iteration.
- The code is compiled with the highest level of optimization for fast execution time, i.e. with the **-O3** flag in the GCC compiler, and with no debugging activated, i.e. the **-DNDEBUG** flag.

Profiling

Profiling tools such as *gprof* and *valgrind* were used during the development to detect bottlenecks and parts of the program that were extra time-consuming. Figure 4.4 is a screen-shot from the result of profiling the code with gprof. In this simulation,

a standard QP with inequality constraints and dimensions $n = 150$ and $m = 225$ was solved by DFGM for the last primal iterate, i.e. 'DFGM last'. The profiling summary is listed in order of the time spent in each file. Note that the time for dfgm.c is 0.0%, but it is listed as the most time-consuming file. This is because functions in dfgm.c are not using much time to execute the code in there own function, but are calling time-consuming subfunctions. The *Time* column in the figure register the "actual" time spent in the file or function.

Figure 4.4 shows that almost all the time for executing the program is spent in the mat library module 'math_functions.c'. Furthermore, 'mtx_vec_mul' is by far the dominating function in this file. This function is multiplying a matrix with a vector, which is define as a special type of matrix multiplication.

Figure 4.5 shows a callgraph of the profiling for the function 'mtx_vec_mul'. A callgraph shows the parent and the children of the function. 'mtx_vec_mul' do not have any children because it does not call any other functions. The parents are the functions who called 'mtx_vec_mul', e.g. the figure shows that the function 'obj' called 'mtx_vec_mul' 12859 times during the execution of the program.

By increasing the dimension of the optimization problem, simulations show that the percentage of the time spent in the function 'mtx_vec_mul' grows even larger. Consequently, the total execution time of the program is strongly dependent on the optimization of this function. In Appendix B, the source code for the matrix-vector multiplication is shown and an alternative method is proposed.

Name (location)	Sampl	Calls	Time/Call	% Time
Summary	154			100.0%
dfgm.c	0			0.0%
fgm.c	1			0.65%
general_functions.c	0			0.0%
init_problem.c	0			0.0%
main.c	0			0.0%
math_functions.c	153			99.35%
abs_2	0	12983	0ns	0.0%
mtx_transpose	0	1	0ns	0.0%
mtx_vec_mul	142	26590	53.403us	92.21%
obj	0	13234	0ns	0.0%
vector_add	2	26464	755ns	1.3%
vector_copy	1	27464	364ns	0.65%
vector_max	0	12860	0ns	0.0%
vector_max_with_zero	0	372	0ns	0.0%
vector_min	1	12860	777ns	0.65%
vector_mul	2	26718	748ns	1.3%
vector_norm_2	0	124	0ns	0.0%
vector_scalar_mul	3	26215	1.144us	1.95%
vector_sub	2	26960	741ns	1.3%

Figure 4.4: gprof n150 dfgm case1

mtx_vec_mul	142	26590	53.403us	92.21%
parents	1	26590	376ns	0.65%
FGM (fgm.c:58)	1	12859	777ns	0.65%
obj (math_functions.c:187)	0	13234	0ns	0.0%
solve_DFGM (dfgm.c:118)	0	497	0ns	0.0%

Figure 4.5: gprof n150 callgraph dfgm case1.

4.2.2. Memory

The tradeoff between memory usage and speed is always an important aspect to consider when implementing programs for low cost HW. In DuQuad, the speed has been given higher priority than the memory usage in most cases. However, after optimizing for speed, memory usage is kept to an absolute minimum. Among the different measures taken, the most important are summarized.

- Only simple variables are allocated after the initialization of the algorithm, i.e. all the vectors used in the execution of the program are allocated at initialization.
- Only the pointers to the vectors are used as inputs to the different functions.
- The outer problem module and the inner problem module use pointers to the same problem vectors. For example when the Hessian matrix is the same for **DGM** and **FGM**, then they both point to the same location in memory where this matrix is stored.
- The Hessian matrix H and the linear constraint matrix G take up the most space in memory. H has the requirement that it has to be symmetric. This means that all information can be retrieved even if only the lower or upper triangular part of the matrix is stored. However, because of speed and readability, the whole matrix is stored in the DuQuad algorithms. G is an $m \times n$ matrix. The transposed of G is used in certain calculations in the program. Because matrix multiplication is a very expensive and time-consuming operation, it is beneficial to have the transposed of G also stored in memory. Then the elements multiplied will be after one another in the memory space, and the process of reading will go much faster.
- In addition to the space that is allocated to store the optimization problem, vectors used in the optimization such as *lambda* and y must be allocated. Furthermore, every algorithm also allocate a minimum number of vectors for storing temporary results, typically two vectors with dimension $n \times 1$ and two with dimension $m \times 1$.
- To make the actual total program size as small as possible, all functions that can be used by more than one module are shared between the modules. However, when the program is implemented on low cost HW, only the algorithm best suited for that particular job will be implemented. Consequently, sharing of code can not be taken advantage of.

4.2.3. Readability and Maintenance

In case of future development of DuQuad, it is very important that the code of the program is kept readable, and that changes and modifications easily can be made. Especially when the program is growing in size it is important that the code has a good modular structure. In DuQuad, certain measures are taken to keep control of the code. The most important are summarized.

- The program has a good modular structure, e.g all math functions are collected in a math library module.
- Most functions are restricted to doing only one dedicated task, which make them easy to test, modify, and optimize.
- All data is bundled into different structs. For example, all the vectors storing the original optimization problem, are contained in the struct *Problem*. This struct then has dedicated functions, e.g. allocating and freeing the memory for the struct members. The *Problem* struct is common for all the algorithms in the toolbox. Hence, they all use the same function for interaction with the struct.
- When passing a struct to a function, only the pointer to the struct is given as input to the function. Furthermore, in Duquad, all the structs that are needed by one algorithm are bundled into a new struct. In practise, a hierarchy of connected structs are constructed and only the pointer to the top of this hierarchy is passed as an argument to a function, who then can access the whole three. The greatest benefit using this approach is that it is easy to add or remove data without changing the function declaration.
- The C-code has header files (.h files) and source files (.c files). The header files are used as interfaces to the source files. For example, by studying the header file dgm.h, it is shown dependencies to other modules, the variable used by this module, and the public functions of the module. All the private functions, i.e. the function that is only utilized within the source file, are declared as static in the beginning of each source file.
- The code has comments wherever it is not self explaining.
- In most scenarios, an error output will be given if the if something goes wrong during the execution of the program.

4.2.4. Debugging

During the development and implementation of DuQuad, the output of the optimization was always tested against other solvers, e.g. Quadprog and Gurobi, to make sure that the algorithm worked properly. In every new stage of the project, the algorithms were tested by comparing the output of numerous randomly generated problems.

Furthermore, every algorithm was first implemented using the high level language MATLAB. The output of the C code should be identical to the output of the MATLAB code at all times. This approach prevented mistakes that otherwise would pass undetected, and it made the programming more effective because it was easier to plan in advance how the structure of the C-code would look like. As a result, a complete and finished version of DuQuad written exclusively in MATLAB is also available for download at the website [1].

Memory Leaks

It is easy to obtain segmentation fault in a C program, i.e. trying to access memory which are not properly allocated. However, sometimes an error will not occur even if the program is accessing memory that not belong to the program. This is called a runtime error and can cause strange behaviour when running the program. Nevertheless, it can be very difficult to discover runtime errors. Furthermore, when compiling the C subroutines into a binary MEX-file, and running this file from MATLAB, the program will shut down without warning or any error message if there is any case of illegal memory accessing. In other words, it is extremely important to correct all errors before the code is compiled with the MEX framework. Under the developing of DuQuad a certain procedure was followed to minimize time-consuming debugging.

1. Implement the algorithm in MATLAB and test the behaviour and output against public available solvers to be sure the algorithm is working properly.
2. Make a version of the algorithm in C-code
3. Profile the C-code with *Valgrind* to detect any illegal memory accessing.
4. Test the output of the C-code against the MATLAB code.
5. Connect the algorithm to the MEX framework.

By following this procedure, about 95% of the bugs in the code were discovered efficiently. Furthermore, when profiling the final version of DuQuad with Valgrind, no memory warnings or errors are detected.

5. Numerical Experiments

In this chapter, a presentation and discussion of the numerical simulations carried out during the project are given.

5.1. Introduction and Simulations Setup

Testing and comparison of the different algorithms that have been implemented, is an extensive task if all possible variants of the optimization problem, the different parameters, and the different settings, are considered. As a consequence, certain assumptions (described below) have been taken, and the presented results are the typical trends that were observed during the project. The commercial optimization solver *Gurobi* is used as a reference to the implemented algorithm outputs. In the following, the assumptions about the optimization problem formulation, the different algorithm settings, and the simulation approach, are given.

5.1.1. Problem Formulation

The two QPs in equation (2.19) and (2.20), introduced in Chapter 2, are mainly considered during the simulations. The QP from the condensed MPC formulation, i.e. (2.19)

$$\begin{aligned} \min_{z} \quad & \tfrac{1}{2} z^T H z + c^T z \\ \text{s.t.} \quad & \hat{lb} \leq Gz - g \leq \hat{ub} \\ & lb \leq z \leq ub, \end{aligned} \tag{5.1}$$

represents **case 1** from Section 3.1 where both $\hat{ub}$ and $\hat{lb}$ exists. The QP from the sparse MPC, i.e. (2.20)

$$\begin{aligned} \min_{z} \quad & \tfrac{1}{2} z^T H z + c^T z \\ \text{s.t.} \quad & Gz = g \\ & lb \leq z \leq ub, \end{aligned} \tag{5.2}$$

represents **case 2**. **DGM** and **DFGM** are tested against **case 1** and **case 2**, while **ALM** and **FALM** only solves problems on the form of **case 2**. Furthermore, the variant of the algorithms evaluated in the last primal iterate is referred to as **last**, and the variant evaluated in the average primal sequence is referred to as **avg**, e.g. **DGM avg**.

5.1.2. Assumptions

Certain assumptions are kept constant throughout the simulations if not specifically stated otherwise.

The Problem

- The Hessian H is symmetric and positive definite.
- A has the dimensions $m \times n$, where $m = n/2$.
- The problem has a feasible solution.
- The initial point is feasible.
- The problem has active constraints in the solution.

The Options

- The tolerance for dual suboptimality ϵ_{ds} and primal feasibility ϵ_{pf} are the same for all algorithms and all problem dimensions during one simulation.
- The tolerance for the primal suboptimality in the inner problem ϵ_{in} can vary between algorithms in the same experiment.
- Higher accuracy means lower tolerance and visa versa.
- The maximum number of iterations for solving the inner problem is set to 100.

The Simulations

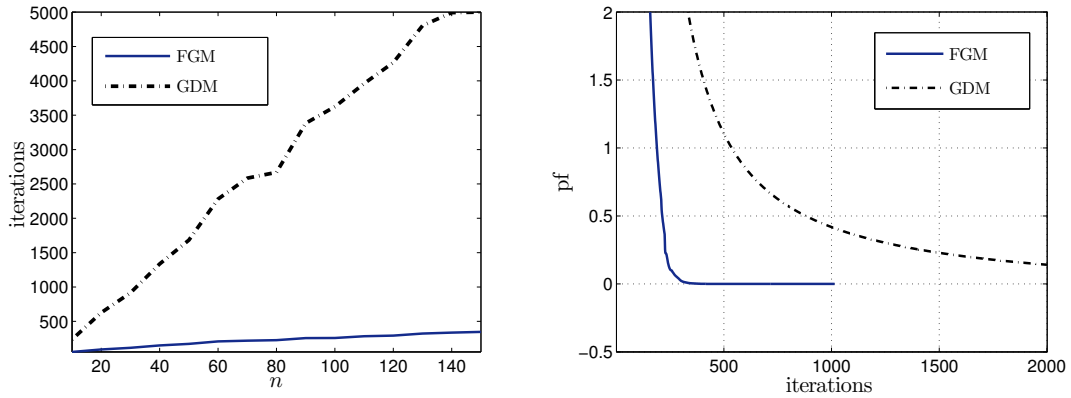
- The output of Gurobi is regarded as the optimal solution and denoted z^*, and the optimal value is denoted f^*. The approximate solution generated by out algorithms is denoted z_k.
- The input to the algorithms, i.e. the optimization problems, are generated randomly, given a certain dimension.
- All results are the average of at least 10 simulations.
- The timing of the algorithms includes all the offline computations done in 'duquad.m', e.g. finding the eigenvalues and calculating the Lipschitz constant.
- Simulations for all solvers are restricted to utilize only one thread, i.e. parallelism is prevented.
- When timing the algorithms, no other programs are running and wireless is turned off.
- Simulations are done on a Lenovo Yoga laptop with an Intel Core i7-3517U CPU, running at 1.90GHz with 8 GB of RAM.

5.1.3. Gurobi

Gurobi is a commercial solver for different types of optimization problems, i.e. also QPs. It has highly optimized algorithms and is regarded as one of the fastest solvers on the marked. When comparing the solution to the different problems generated in this project, Gurobi had the exact same output as the well known solver Quadprog, but Gurobi was significantly faster. Although Gurobi uses second order methods, it is used as a reference to the implemented algorithm in this Thesis during the simulations. To be able to make a fear comparison of the runtime, Gurobi was restricted to only using one thread.

5.2. GDM vs. FGM

In the end of Chapter 2, the gradient descent method (**GDM**) and fast gradient method (**FGM**) were discussed, see section 2.4.2. In theory, **FGM** should have a significant faster convergence rate. This corresponds well with the results presented in Figure 5.1. With growing problem dimensions, GDM do not stand a chance when compared to **FGM**. Needless to say, **FGM** was used to solve the inner problem for all the dual algorithms in this project.



(a) Number of iterations used to find solution with increasing problem dimension n

(b) Primal suboptimality $| f(z_k) - f(z_{k-1}) |$

Figure 5.1: Gradient Descent Method vs. Fast Gradient Method.

5.3. Tightness of Theoretical Complexity Bound for DGM

In this section, it is shown that the theoretical complexity bound of DGM also make an upper bound in practice. This simulation is done for **case 3** of equation 5.1, where the optimization variable z is unbounded. This optimization problem is formulated as:

$$\min_{z \in \mathbb{R}^n} \quad \frac{1}{2} z^T H z + c^T z \qquad \text{s.t.} \quad Gz \leq g. \tag{5.3}$$

From the theoretical convergence for **DGM** stated in Appendix A, the following relation should hold:

$$|f(\hat{z}_k) - f^*| \leq \frac{2L_{\mathrm{d}} R_{\mathrm{d}}^2}{k} \leq \epsilon_{\mathrm{out}}, \tag{5.4}$$

where k is the number of outer iterations. In theory, this means that by first calculating f^* with Gurobi, and then run **DGM** until the primal feasibility condition $|f(\hat{z}_k) - f^*| \leq \epsilon_{\mathrm{out}}$ is satisfied, the theoretical upper bound for number of iterations should be:

$$ub_k \leq \frac{2L_{\mathrm{d}} R_{\mathrm{d}}^2}{\epsilon_{\mathrm{out}}}. \tag{5.5}$$

By setting the inner accuracy equal to the outer accuracy, i.e. $\epsilon_{\mathrm{in}} = \epsilon_{\mathrm{out}} = 10^{-2}$, the result in Figure 5.2 is obtained. Even though the tightness of the bound can be discussed, it is observed that ub_k follows the trend of **DGM avg**.

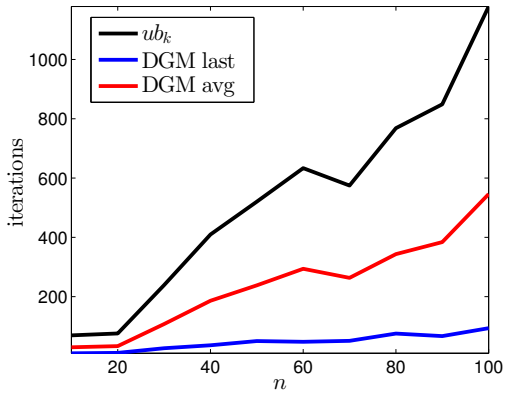


Figure 5.2: Number of iterations with increasing problem dimension n. ub_k is the theoretical upper bound for number of iterations for **DGM avg**.

5.4. Convergence Comparison for Case 1

In this section, simulations comparing **DGM** and **DFGM**, when solving problems on the form of **case 1** with increasing problem dimension n, are plotted. **DGM avg** is omitted because the result is not comparable to the other three algorithms, i.e. the convergence is much slower.

In the first experiment, shown in Figure 5.3, the inner tolerance is kept the same for both **DFGM last** and **DFGM avg** ($\epsilon_{\text{in}} = 10^{-7}$) , while **DGM last** is given a much higher tolerance ($\epsilon_{\text{in}} = 10^{-4}$). Other testing has proven that **DFGM last** is not converging if the inner tolerance is high compared to outer tolerance.

Figure 5.4(a) shows the number of outer iterations used to obtain a solution that satisfy the stopping criteria, with the given tolerances. The result correspond well to the theory, given that **DFGM last** uses least iterations and **DGM last** uses most.

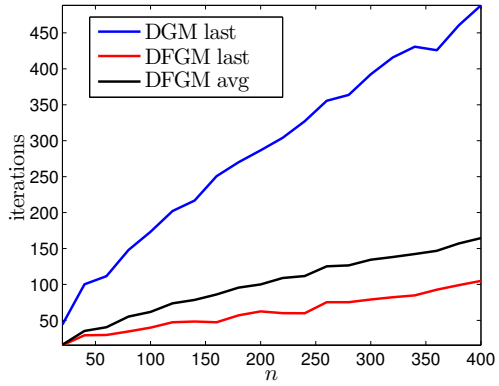
Figure 5.4(b) shows the total number of inner iterations, i.e. the sum of iterations used to solve the inner problem from all the outer iterations. Note that **DGM last** uses far less inner iterations. This is a consequence of the low accuracy for the inner problem. Furthermore, the **DFGM** must solve the inner problem twice for every outer iteration. One time to obtain the z_k update and one time to check the stopping criteria.

Figure 5.4(c) displays the runtime used by the algorithms to obtain a solution to the optimization problem. An interesting observation is that runtime of each algorithm reflects the total number of inner iterations in Figure 5.4(b). The runtime of Gurobi is also included in the plot, and it is observed that it is beaten by **DGM last**, which obtain the approximate solution faster.

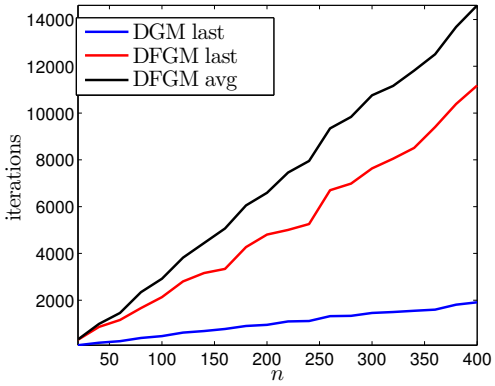
Figure 5.4(d) and 5.4(e) compare the approximate solution z_k to the optimal solution z^*, provided by Gurobi. These two plots do not say much about how good the solution is compared to Gurobi, because the order of the solutions are not stated. However, it is a good test to compare the implemented algorithms against each other. From the plots it is observed that **DFGM last** is more unstable that the other two methods. Furthermore, **DGM last** has better primal suboptimality, than **DFGM avg**, but with primal feasibility it is the other way around.

By performing simulations where the inner accuracy of **DFGM avg** is changed to be equal to the **DGM last**, but keeping all the other parameters constant, the results displayed in Figure 5.4 are obtained. It is observed that **DFGM avg** now has less inner iterations and faster runtime. In addition, the algorithm still gives good results with respect to suboptimality and feasibility.

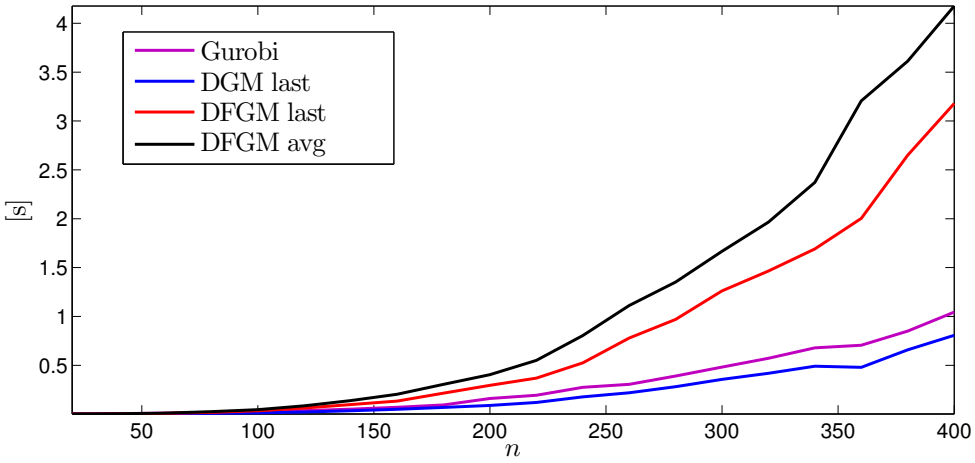
It should be mentioned that the inner accuracy requirement for the different algorithms, will be different with growing problem sizes, i.e. the experiments shown in Figure 5.3 and Figure 5.4 do not give the full picture of the reality. During other simulations it was observed that the algorithms did beat each other using different problem formulations and different problem dimensions.



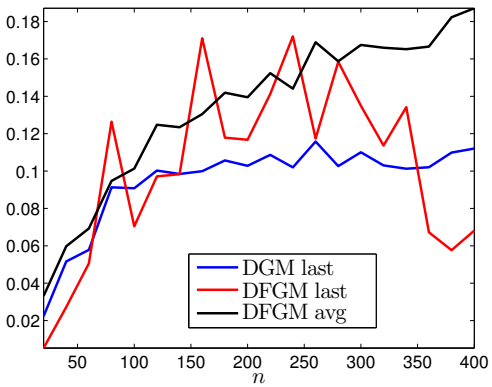
(a) Outer iterations



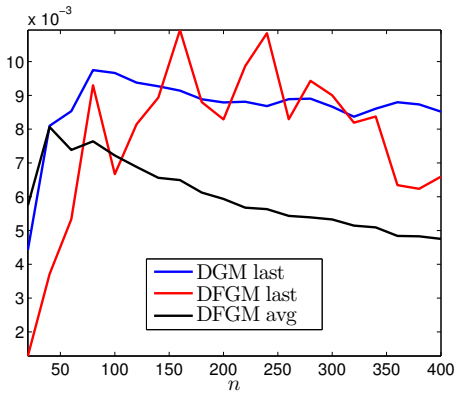
(b) Sum inner iterations



(c) Runtime

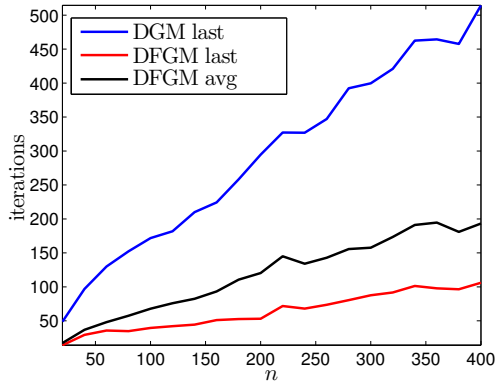


(d) Primal suboptimality $\mid f^* - f(z_k) \mid$

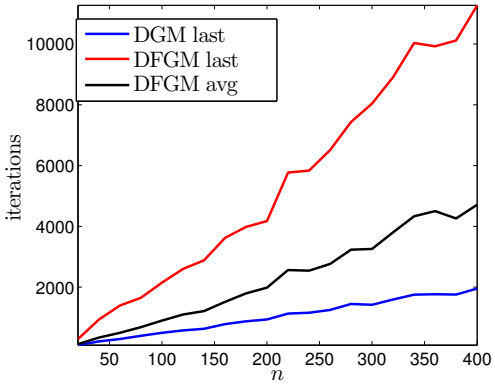


(e) Primal feasibility $\| z^* - z_k \|$

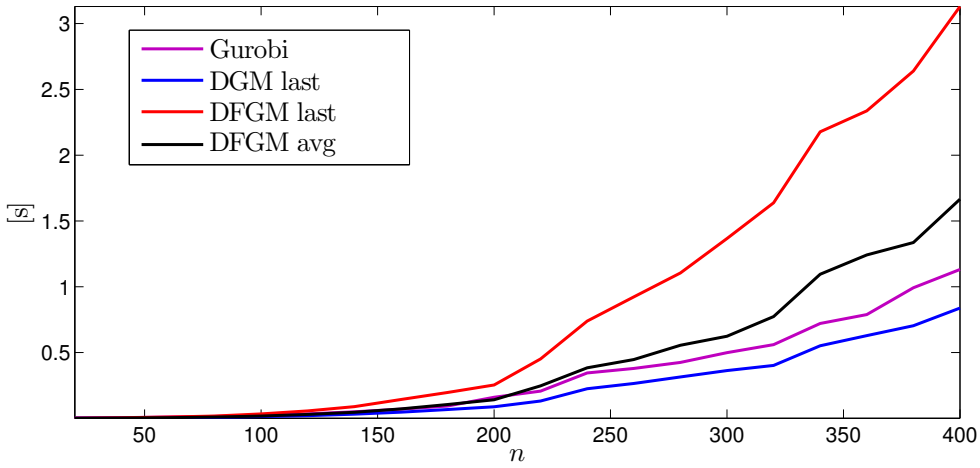
Figure 5.3: Case 1 simulated for **DGM** and **DFGM** with increasing n. **DFGM last** and **DFGM avg** have the same ϵ_{in}.



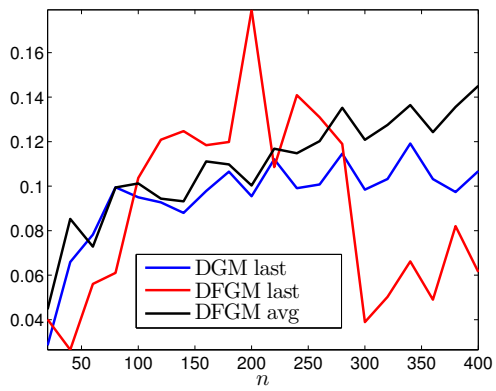
(a) Outer iterations



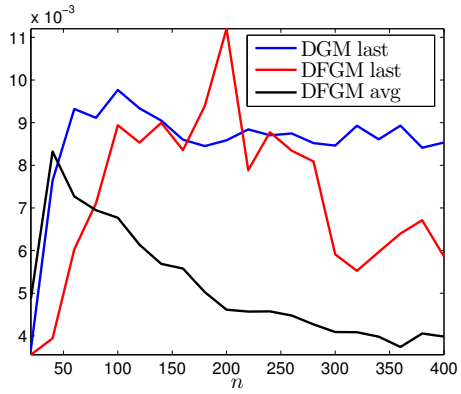
(b) Sum inner iterations



(c) Runtime



(d) Primal suboptimality $\mid f^* - f(z_k) \mid$



(e) Primal feasibility $\| z^* - z_k \|$

Figure 5.4: Case 1 simulated for **DGM** and **DFGM** with increasing n. **DGM last** and **DFGM avg** have the same ϵ_{in}.

5.5. Convergence Comparison for Case 2

In this section, the convergence of primal feasibility for all the algorithms are compared, using **case 2** from equation (5.2). Then the algorithms with best convergence properties are benchmarked against each other, and Gurobi, on big data problems, i.e. problems with large dimensions.

5.5.1. Convergence of Primal Feasibility

In this experiment, only one problem is considered. The figures show plots from one simulation where all the algorithms solve the same problem and all algorithms use the same tolerance ϵ_{in} when solving the inner problem. The outer tolerances for dual suboptimality and primal feasibility are set to zero. This means that the algorithms will continue until they reach the maximum number of outer iterations. The problem that is used in this simulation has has dimension $n = 200$.

In the experiment showed in Figure 5.5 the inner accuracy is set to $\epsilon_{\text{in}} = 10^{-3}$. Figures 5.5(a), 5.5(b) and 5.5(c) show three different zoomed perspectives of the same simulation.

Figures 5.5(a) has the perspective where all the 8 different algorithms of DuQuad are included. Clearly, the four Augmented algorithms have mush faster convergence then the other algorithms. This plot reflects the typical behaviour for other problems as well.

Figure 5.5(b) show a closer perspective of the four augmented algorithms. From this illustration it seems that **ALM avg** has a stable convergence, but it is still not comparable to the three best performing algorithms.

Figure 5.5(c) displays an even closer perspective on the value of the primal feasibility, while zooming out on the axis with number of iterations. From the illustration it is observed that **FALM avg** has a stable converging curve, but is much slower than the two algorithms that are evaluated in the last primal iterate. Furthermore, it is shown that even though the **FALM last** has the fastest rate of convergence in the beginning, it will stop the convergence at a certain value of primal feasibility and start oscillating. **ALM last**, on the other hand, does not converge as fast in the beginning as **FALM last**, but has a more stable behaviour and converges down to zero.

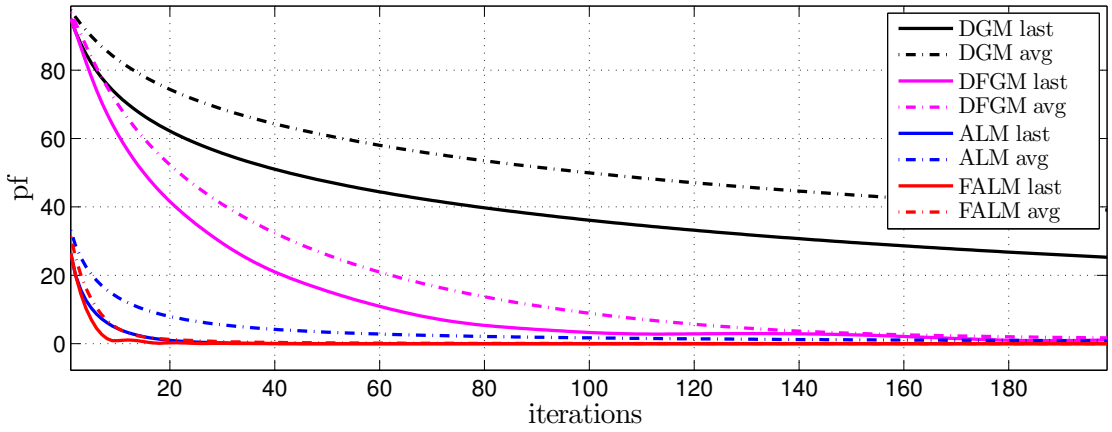
The plots illustrated in Figure 5.6 shows how some of the algorithms will converge with three different values of the inner tolerance.

Figure 5.6(d) illustrate the convergence of the best method evaluated in the average, i.e. **FALM avg**. The plot shows that the algorithm is more or less unaffected by the inner tolerance. This is also the case for all other algorithms that evaluate in the average. Furtermore, a similar robust behaviour is observed for **ALM last** in Figure 5.6(b).

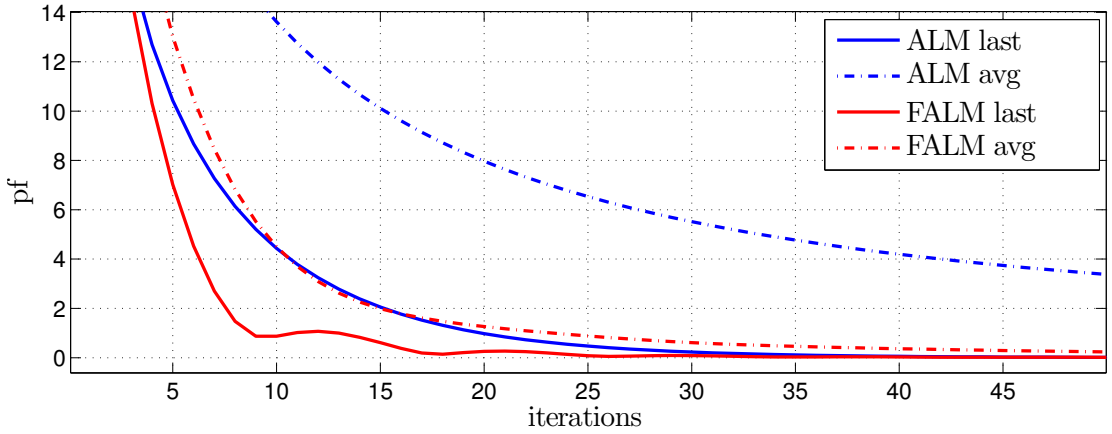
On the other hand, from Figure 5.6(a) and Figure 5.6(c), it is shown that **DFGM last** and **FALM last** are strongly dependent on the inner tolerance. When solving the inner problem with a higher accuracy, the algorithms converge closer to zero

before they start to oscillate.

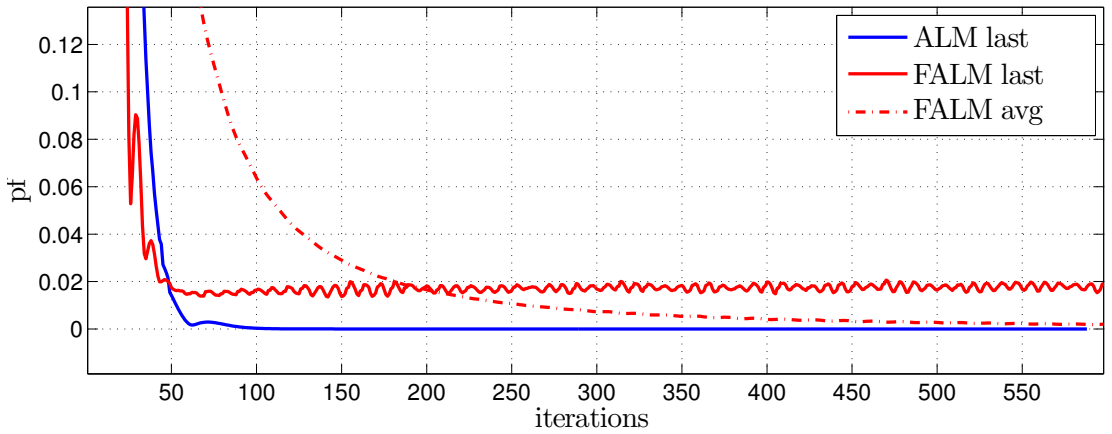
To summarize, if it is possible to set the stopping criterion tolerance for the primal feasibility at a level the algorithm is able to reach, then **FALM last** seem to have the best and fastest convergence properties. However, if the problem is unknown, **ALM last** is more robust and seems to be the better choice. By combining the fast convergence of one method and the robustness of the other, a hybrid algorithm of the two could be an interesting and promising option. **DGM last** and **DFGM last** seem to have the same type of relationship as the augmented algorithms, and a hybrid method of these two are presented in Necoara and Patrascu [13].



(a) Global perspective

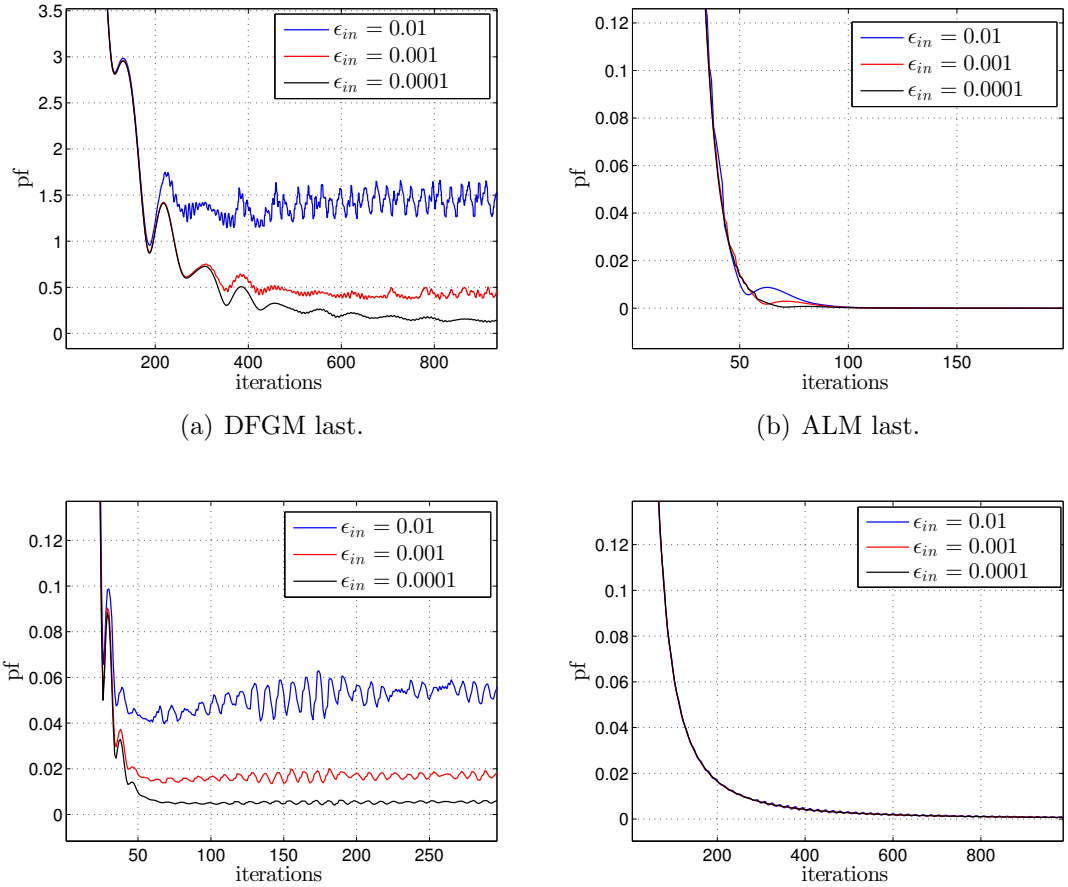


(b) Close perspective



(c) Close perspective over many iterations

Figure 5.5: Value of primal feasibility vs. number of iterations. All algorithms are solving the same problem.



(a) DFGM last.

(b) ALM last.

(c) FALM last.

(d) FALM avg.

Figure 5.6: Value of primal feasibility vs. number of iterations, simulated for three different values of ϵ_{in}.

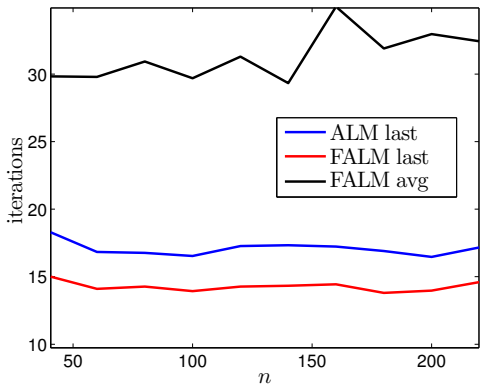
5.5.2. Benchmark on Big Data Problems

In this experiment, the algorithms that showed promising results in the previous subsection, when solving problems on **case 2**, are compared to each other and Gurobi. The algorithms solve problems with growing dimensions n. The inner tolerance is set differently: **ALM last**: $\epsilon_{\text{in}} = 10^{-4}$, **FALM last**: $\epsilon_{\text{in}} = 10^{-6}$, and **FALM avg**: $\epsilon_{\text{in}} = 10^{-3}$. These settings give the best results when testing the algorithms individually. Furthermore, the outer tolerances are kept constant and the same for all algorithms, with $\epsilon_{\text{ds}} = 10^{-4}$ and $\epsilon_{\text{pf}} = 0.03$. The optimal value f^* is of the order 10^3.

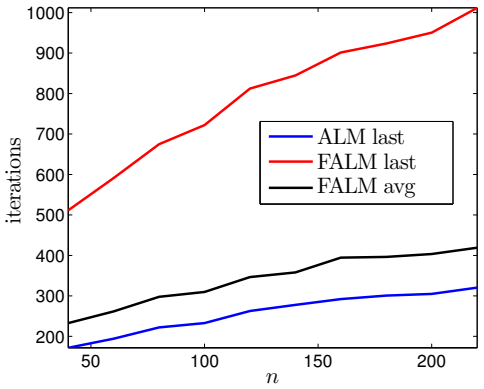
Figure 5.7 and Figure 5.8 have the same settings, but simulate over different problem dimensions n. In both cases it is observed that **FALM last** uses the least amount of outer iterations, but the highest amount of total inner iterations. On the other hand, **ALM last** uses the least amount of inner iterations and consequently has the fastest runtime.

Furthermore, Figure 5.8(d) and Figure 5.8(e) show that **FALM avg** has a substantially less accurate solution than the other algorithms, when comparing the approximate solution to the optimal solution provided by Gurobi.

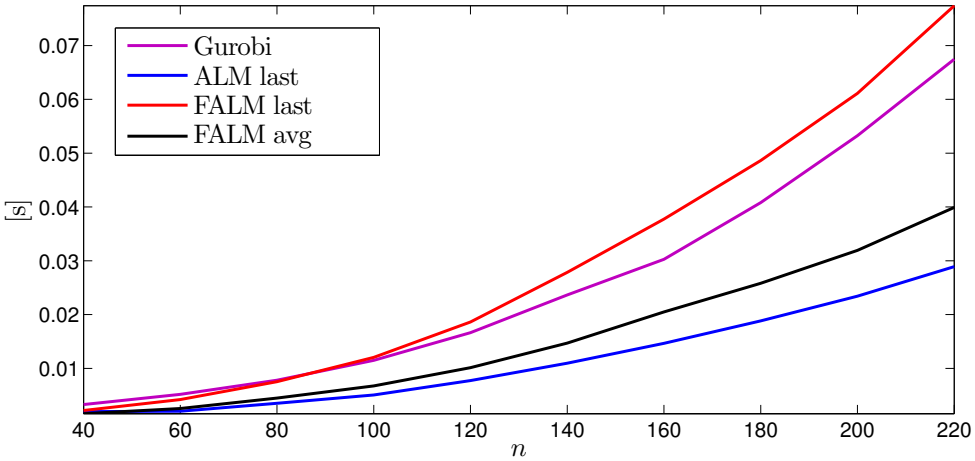
Figure 5.7(c) and Figure 5.8(c) illustrate the runtime for all the algorithms, including Gurobi. When the problem dimension increases, it is observed that all the tree implemented algorithms are able to beat the runtime of Gurobi. This is a very promising result and shows that the implemented first order algorithms are comparable to even one of the better solvers out there. It should be noted that this result only counts for certain types of optimization problems, and when an *approximate* solution is sufficient in the application.



(a) Outer iterations

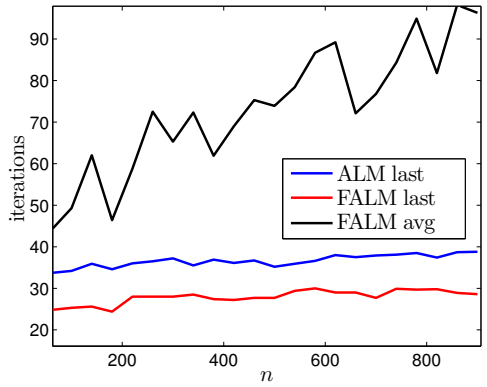


(b) Sum inner iterations

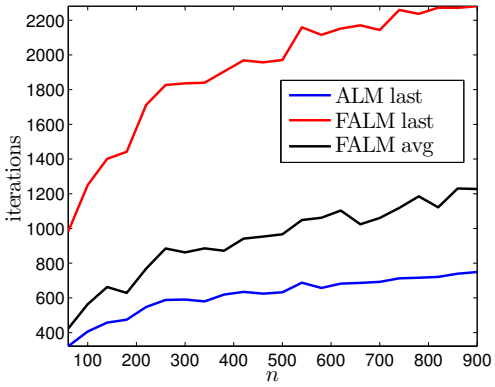


(c) Runtime

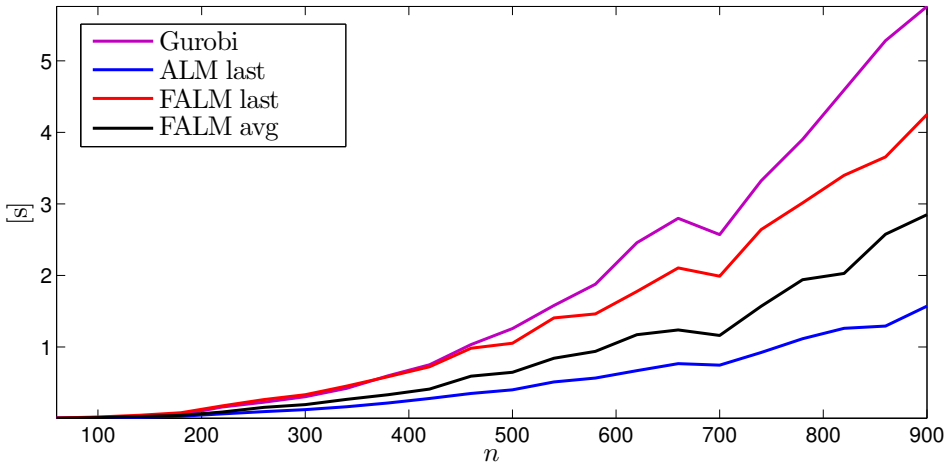
Figure 5.7: Solving **Case 2** problems with increasing n.



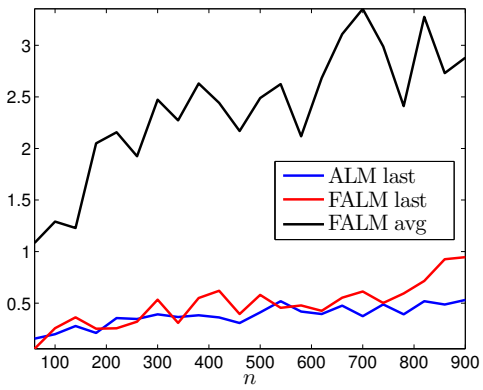
(a) Outer iterations



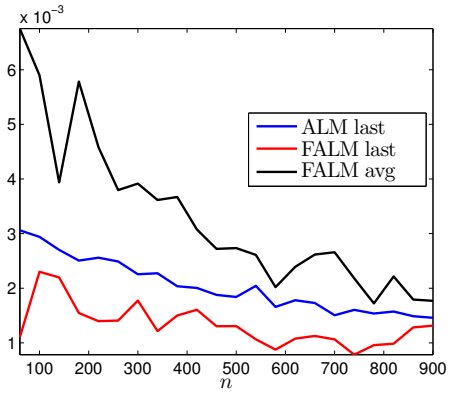
(b) Sum inner iterations



(c) Runtime



(d) Primal suboptimality $| f^* - f(z_k) |$



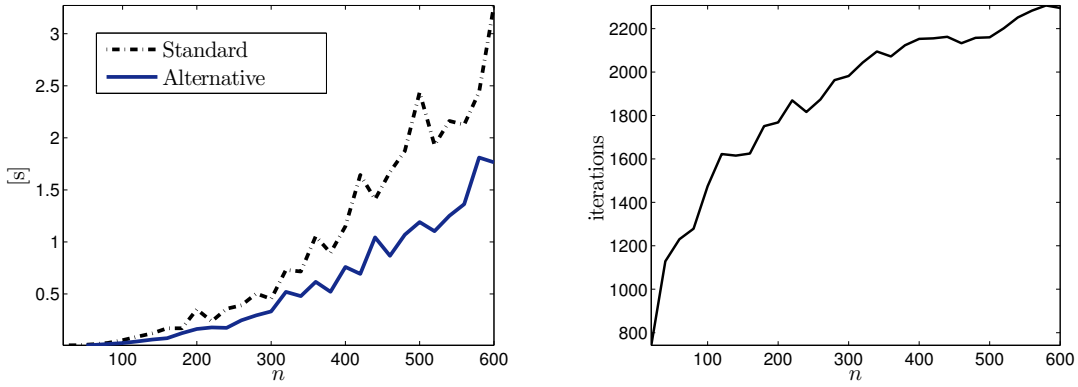
(e) Primal feasibility $\| z^* - z_k \|$

Figure 5.8: Solving **Case 2** problems with increasing n.

5.6. Matrix Multiplication

In section 4.2.1 the code was profiled when solving a larger problem using **DFGM last**. It was shown that the function that do the matrix-vector multiplication is using most of the total execution time of the program. In appendix B, two different implementations of this multiplication are presented. These two are referred to as the standard and the alternative approach. The alternative approach is used in all other simulations in this Thesis.

In Figure 5.9, these to approaches are compared by solving random generated problems using **FALM last**. Clearly the alternative approach achieves lower runtimes with growing problem dimension. Figure 5.9(b) shows the total number of inner iterations for the same experiment.



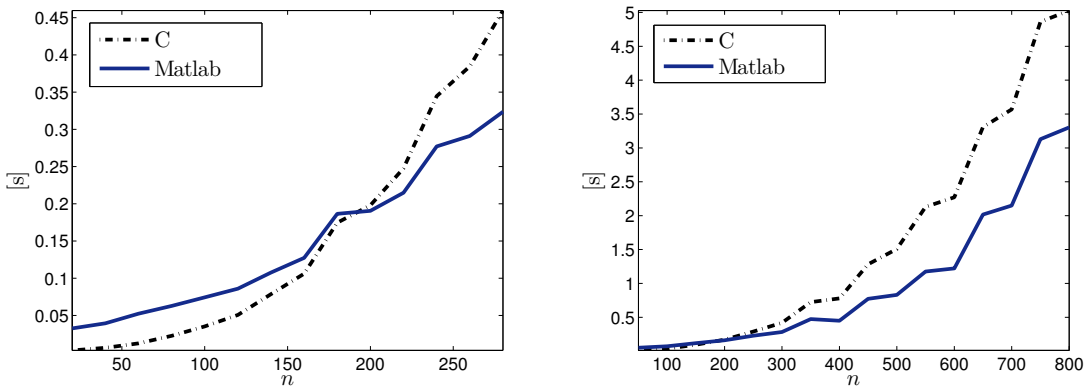
(a) Runtime

(b) Totale number of inner iterations

Figure 5.9: **FALM last** solves random problems with dimension n, using two different methods for the matrix-vector multiplication.

5.7. Runtime Comparison: C vs. MATLAB

To check the correctness of the C-code implementation and to plan the program structure, a version of DuQuad, written exclusively in MATLAB code, was created in parallel to the C-code implementation. The result is a complete MATLAB version of DuQuad, that has the same inputs and outputs as the C-version. The general assumption is that the C-version should be much more efficient in computational runtime. Figure 5.10 shows the runtime of the two implementations when solving the same problems with increasing dimension n. It is observed that the MATLAB version has better performance for larger problems. Note, this result is only obtained for certain problems and certain algorithms, and is mainly because MATLAB is known to use highly optimized libraries for matrix and vector operations. Because the implemented algorithm uses most of the time doing matrix-vector multiplication, this operation is done faster in MATLAB for larger dimensions.



(a) Runtime for small dimensions

(b) Runtime for larger dimensions

Figure 5.10: Runtime **FALM last** comparing C and MATLAB implementation on the same problems.

6. Conclusion

The main goal of this Thesis was to implement and analyse first order algorithms that solves convex quadratic problems that appear in MPC for embedded linear systems. Given the complete and optimized toolbox DuQuad, which contains eight different versions of such algorithms that can solve a wide range of different quadratic programs with promising results in efficiency, accuracy and robustness, the goal has been fulfilled.

Motivated by an introduction to MPC, topics of optimization theory and quadratic programming were presented, and some methods were reviewed for solving quadratic programs subjected to constraint that can be easily projected. Moreover, four dual methods for solving QPs with complicated constraints were presented in Chapter 3. These algorithms were evaluated in both the last primal iterate and the average primal sequence. The algorithms evaluated in the average showed a robust behaviour, but the algorithms evaluated in the last had a faster rate of convergence.

Furthermore, the augmented methods had better performance on the equality constrained problems. In addition, it was also shown that the algorithms that used the least amount of iterations when solving the inner problem, also had the fastest runtime.

By enforcing certain assumptions on the optimization problem and simulation process, some of the algorithms were able to obtain an approximate solution with a speed that was comparable to one of the most efficient solvers on the market.

7. Further work

The algorithms implemented and analysed in this Thesis were shown to be promising in efficiency, accuracy and robustness. However, further optimization and development can be made. Some suggestions for further work is given in the following.

A first approach may be to expand the augmented methods **ALM** and **FALM** to also solve the three other cases of the QP, i.e. to include inequalities in the linear constraints. Furthermore, by combining the fast convergence of some algorithms and the robustness of others, a hybrid version could be implemented. A hybrid solution for the **DGM** and **DFGM** is suggested by [13].

If DuQuad should be used as a toolbox on fast computing HW, the code should be connected to optimized libraries for matrix and vector operations, and parts of the code should be rewritten to take advantage of parallel computing.

If one of the implemented algorithms should be used on low-cost HW, it could be implemented with fixed point representation instead of floating point representation.

References

[1] Sverre Kvamme. Duquad, 2014. URL `http://sverrkva.github.io/duquad/`.

[2] Sverre Kvamme. *Real Time Embedded Model Predictive Control - An Introductory Study*. Project thesis at NTNU, 2014.

[3] Mark Cannon. C21 model predictive control. accessed: February, 2014, 2013. URL `http://www.eng.ox.ac.uk/~conmrc/mpc/`.

[4] Morten Hovd. *Lecture notes for the course Advanced Control of Industrial Processes*. Institutt for Teknisk Kybernetikk, NTNU, 2013.

[5] Lars Imsland. *Introduction to Model Predictive Control*. Internal report, Department of Engineering Cybernetics, NTNU.

[6] M. Rubagotti, P. Patrinos, and A. Bemporad. *Stabilizing embedded MPC with computational complexity guarantees*. Control Conference (ECC), 2013 European.

[7] Ion Necoara. *Worst-case computational complexity analysis for embedded MPC based on dual gradient method*. he 18th International Conference on System Theory, Control and Computing, Sinaia, 2014, 2014.

[8] Alberto Bemporad and Panagiotis Patrinos. *Simple and certifiable quadratic programming algorithms for embedded linear model predictive control*. IFAC, 2012.

[9] Valentin Nedelcu, Ion Necoara, and Quoc Tran Dinh. *Computational Complexity of Inexact Gradient Augmented Lagrangian Methods: Application to Constrained MPC*. SIAM Journal on Control and Optimization. 2014.

[10] Markus Kögel and Rolf Findeisen. *A fast gradient method for embedded linear predictive control*. Proceedings of the 18th IFAC world congress. 2011.

[11] Markus Kögel, Pablo Zometa, and Rolf Findeisen. *On tailored model predictive control for low cost embedded systems with memory and computational power constraints*. Technical report, 2012.

[12] Ion Necoara and Valentin Nedelcu. *Rate analysis of inexact dual first order methods: application to dual decomposition*. IEEE Transactions on Automatic Control, 2014.

[13] Ion Necoara and Andrei Patrascu. *Iteration complexity analysis of dual first order methods for convex programming*. Technical Report, University Politehnica Bucharest, 2014.

[14] Stefan Richter, Colin Neil Jones, and Manfred Morari. *Computational Complexity Certification for Real-Time MPC With Input Constraints Based on the Fast Gradient Method*, volume 57 of *Automatic Control, IEEE Transactions on*. 2012.

[15] P. Zometa, M. Kogel, and R. Findeisen. Ao-mpc: A free code generation tool for embedded real-time linear model predictive control. *American Control Conference (ACC), 2013*, pages 5320–5325, 2013.

[16] Jorge. Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2006.

[17] Yurii Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). *Soviet Mathematics Doklady*, 27:372–376, 1983.

[18] Yurii Nesterov and I E Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer, 2004. ISBN 1402075537.

[19] Morton Slater. *Lagrange multipliers revisited*. Springer, 2014.

[20] Sverre Kvamme. Duquad documentation, 2014. URL `http://sverrkva.github.io/duquad_doc_ccode/`.

Appendix

Appendix A

Theoretical Convergence and Accuracy Estimates

The theoretical bounds are stated for all four algorithms explained in Chapter 3. The bounds are proven when the algorithms solves for the average primal sequence. See [12] for proofs regarding **DGM** and **DFGM**, and [9] for proofs regarding **ALM** and **FALM**.

k_{out} is the number of outer iterations, ϵ_{out} is the outer tolerance in the stopping criteria, and ϵ_{in} is the inner tolerance (primal suboptimality in **FGM**). Furthermore, the constraints are denoted by $a(\,\cdot\,)$, the "real" optimal solution is denoted by f^*, and $R_{\text{d}} = \| \lambda^* \|$ (λ^* is the "real" optimal Lagrangian).

Algorithm DGM

- Outer iterations and inner accuracy:

$$k_{\text{out}} := \left\lfloor \frac{4L_{\text{d}}R_{\text{d}}^2}{\epsilon_{\text{out}}} \right\rfloor, \quad \text{and} \quad \epsilon_{\text{in}} := \epsilon_{\text{out}}. \tag{A.1}$$

- Dual Suboptimality:

$$f^* - d(\lambda_{k_{\text{out}}})) \leq \frac{5}{4}\epsilon_{\text{out}}. \tag{A.2}$$

- Primal Feasibility:

$$\left\| \left[a(\hat{z}_{k_{\text{out}}}) \right]_+ \right\| \leq \frac{2}{R_{\text{d}}}\epsilon_{\text{out}}. \tag{A.3}$$

- Primal Suboptimality:

$$-2\epsilon_{\text{out}} \leq f(\hat{z}_{k_{\text{out}}}) - f^* \leq \epsilon_{\text{out}}. \tag{A.4}$$

Algorithm DFGM

- Outer iterations and inner accuracy:

$$k_{\text{out}} := \left\lfloor 2R_{\text{d}}\sqrt{\frac{L_{\text{d}}}{\epsilon_{\text{out}}}} \right\rfloor, \quad \text{and} \quad \epsilon_{\text{in}} := \frac{\epsilon_{\text{out}}\sqrt{\epsilon_{\text{out}}}}{2R_{\text{d}}\sqrt{L_{\text{d}}}}. \tag{A.5}$$

- Dual Suboptimality:

$$f^* - d(\lambda_{k_{\text{out}}})) \leq 3\epsilon_{\text{out}}. \tag{A.6}$$

- Primal Feasibility:

$$\left\| \left[a(\hat{z}_{k_{\text{out}}}) \right]_+ \right\| \leq \frac{6}{R_{\text{d}}}\epsilon_{\text{out}}. \tag{A.7}$$

- Primal Suboptimality:

$$-6\epsilon_{\text{out}} \leq f(\hat{z}_{k_{\text{out}}}) - f^* \leq 2\epsilon_{\text{out}}. \tag{A.8}$$

Algorithm ALM

Given $\rho > 0$ and $0 \leq L_{\text{d}} \leq \bar{L}$:

- Outer iterations and inner accuracy:

$$k_{\text{out}} := \left\lfloor \frac{\bar{L}R_{\text{d}}^2}{\epsilon_{\text{out}}} \right\rfloor, \quad \text{and} \quad \epsilon_{\text{in}} := \frac{1}{2(1+\sqrt{L_{\text{d}}}R_{\text{d}})}\epsilon_{\text{out}}. \tag{A.9}$$

- Dual Suboptimality:

$$f^* - d_\rho(\lambda_{k_{\text{out}}})) \leq \epsilon_{\text{out}}. \tag{A.10}$$

- Primal Feasibility:

$$\left\| \left[A\hat{z}_{k_{\text{out}}} - b \right]_+ \right\| \leq \frac{3}{R_{\text{d}}}\epsilon_{\text{out}}. \tag{A.11}$$

- Primal Suboptimality:

$$-\left(\frac{3\left\|\lambda^*\right\|}{R_{\text{d}}} + \frac{9\rho}{2R_{\text{d}}^2}\epsilon_{\text{out}}\right)\epsilon_{\text{out}} \leq f(\hat{z}_{k_{\text{out}}}) - f^* \leq \left(\frac{1}{2} + \frac{\left\|\lambda_0\right\|^2}{2R_{\text{d}}^2}\right)\epsilon_{\text{out}}. \tag{A.12}$$

Algorithm FALM

- Outer iterations and inner accuracy:

$$k_{\text{out}} := \left\lfloor 2R_{\text{d}}\sqrt{\frac{L_{\text{d}}}{\epsilon_{\text{out}}}} \right\rfloor, \quad \text{and} \quad \epsilon_{\text{in}} := \frac{3}{8(1+\sqrt{L_{\text{d}}}R_{\text{d}})(k_{\text{out}}+3)}\epsilon_{\text{out}}. \tag{A.13}$$

- Dual Suboptimality:

$$f^* - d_\rho(\lambda_{k_{\text{out}}})) \leq \epsilon_{\text{out}}. \tag{A.14}$$

- Primal Infeasibility:

$$\left\| \left[A\hat{z}_{k_{\text{out}}} - b \right]_+ \right\| \leq \frac{3}{R_{\text{d}}}\epsilon_{\text{out}}. \tag{A.15}$$

- Primal Feasibility:

$$-\left(\frac{3\left\|\lambda^*\right\|}{R_{\text{d}}} + \frac{9\rho}{2R_{\text{d}}^2}\epsilon_{\text{out}}\right)\epsilon_{\text{out}} \leq f(\hat{z}_{k_{\text{out}}}) - f^* \leq \left(\frac{\left\|\lambda_0\right\|^2 + R_{\text{d}}^2}{2R_{\text{d}}^2}\right)\epsilon_{\text{out}}. \tag{A.16}$$

Appendix B

Matrix Multiplication

From the profiling done in section 4.2.1 it is shown that the matrix-vector multiplication is the most time-consuming function in the DuQual implementation. There exists many optimized math libraries for doing matrix multiplication. However, for DuQuad, one of the main objective is to restrict the use of external libraries as much as possible.

The free code generation tool for embedded real-time linear model predictive control presented in [15], utilizes a dual augmented gradient method to solving QPs, and has an implementation in C-code without external libraries for matrix multiplication. The matrix-vector multiplication for this tool is done using the most standard approach. Hence, the implementation of the function below is referred to as the standard approach.

```
1  /* matrix-vector multiplication: res = mtx * v
2   * mtx has size (rews x cols) */
3
4  void mtx_vec_mul(const real_t *mtx, const real_t *v,
5                   real_t *res, const uint32_t rows,
6                   const uint32_t cols)
7  {
8      uint32_t i;      // row number
9      uint32_t j;      // column number
10     uint32_t k = 0; // matrix index (row * col)
11     for (i=0;i<rows;i++) {
12         res[i] = 0.0;
13         for (j=0;j<cols;j++) {
14             res[i] += mtx[k++] * v[j];
15         }
16     }
17 }
```

Using the standard approach, only one element in the vectors mtx and v is accessed every iteration in the inner for-loop. On the other hand, the code below represent the alternative implementation of matrix-vector multiplication that is used in DuQuad. This function access four elements during every iteration in the inner for-loop. This is referred to as the alternative method and yields better results for large dimensions, see simulations in Section 5.6. The alternative method is not found in any official paper, but is discussed in various threads on programming websites.

```
1  /* matrix-vector multiplication: res = mtx * v
2   * mtx has size (rews x cols) */
3
4  void mtx_vec_mul(const real_t *mtx, const real_t *v,
5                   real_t *res, const uint32_t rows,
6                   const uint32_t cols)
7  {
8      uint32_t i; // row number
9      uint32_t j; // column number
10     uint32_t k; // matrix index (row * col)
11     real_t temp;
12     k = 0;
13     for (i=0;i<rows;i++) {
14         temp = 0.0;
15         for (j=0;j<=cols-4;j+=4) {
16             temp += (mtx[k] * v[j] +
17                      mtx[k+1] * v[j+1] +
18                      mtx[k+2] * v[j+2] +
19                      mtx[k+3] * v[j+3] );
20             k+=4;
21         }
22         for (; j<cols; j++){
23             temp += mtx[k++] * v[j];
24         }
25         res[i] = temp;
26     }
27 }
```

Appendix C

DuQuad: User Manual

C.1. Introduction

The *DuQuad* optimization toolbox solves convex quadratic programs using dual first order optimization algorithms. The algorithms have predictable and fast convergence, low memory footprint, and use only basic arithmetic and logical operations. DuQuad is therefore suited to be utilized by real-time applications running on low-cost HW such as simple microcontrollers. Furthermore, DuQuad has an user friendly Matlab interface for maximum productivity, and the algorithms are implemented in efficient C-code.

The algorithms attempts to solve the QP problem:

$$\begin{aligned} \min_{z} \quad & \tfrac{1}{2} z^T H z + c^T z \\ \text{s.t.} \quad & \hat{lb} \leq Gz - g \leq \hat{ub} \\ & lb \leq z \leq ub, \end{aligned} \tag{C.1}$$

where $H \in \mathbb{R}^{n \times n}$ is the Hessian, $G \in \mathbb{R}^{m \times n}$ is a matrix for the linear constraints, and $c, lb, ub \in \mathbb{R}^n$ and $g, \hat{lb}, \hat{ub} \in \mathbb{R}^m$ are column vectors.

Algorithms

DuQuad embeds the four different algorithms:

- Dual Gradient Method (DGM)
- Dual Fast Gradient Method (DFGM)
- Dual Augmented Lagrangian Method (ALM)
- Dual Fast Augmented Lagrangian Method (FALM)

Note that ALM and FALM can only solve problems with equality constraints, i.e. the case where $\hat{lb} = \hat{ub}$.

Download and Installation

Information, documentation, and code downloads can be found by following the link below.

- DuQuad website: `http://sverrkva.github.io/duquad/`
- Direct download: `https://github.com/sverrkva/duquad/`
- Documentation of C-code: `http://sverrkva.github.io/duquad_doc_ccode/`

The c-code needs to be compiled into a mex-file. A makefile (make.m) is included in the code download. If running a linux distribution it should be adequate to run the make.m to compile the program. Furthermore, an example-file is included to get a quick start.

The download also includes a Matlab version of DuQuad where all the algorithms are implemented in Matlab. The Matlab version has the same behaviour and almost identically inputs and outputs as the main version.

C.2. Short Tutorial

Formulate the Problem

Formulate an optimization problem on the form of equation (C.1). For example:

```
H = [11 4 ; 4 22];  % Hessian matrix
c = [3 ; 4];        % gradient vector
G = [1 1;2 1];      % linear constraints matrix
g = [2 ; 3];        % linear constraints vector
lb_hat = [-2 ; -2]; % lower bound for the linear constraints
ub_hat = [2 ; 2];   % upper bound for the linear constraints
lb = [-1 ; -2];     % lower bound for optimization variable z
ub = [0.5 ; 2];     % upper bound for optimization variable z
z0 = [0.5 ; -0.5];  % initial point
```

Run the Program

To solve the problem, call the duquad function with the problem as input:

```
[zopt,fopt] = duquad(H,c,G,g,lb_hat,ub_hat,lb,ub,z0);
```

If the linear constraints is have lower bound, and the optimization variable has no upper upper bound then the the function will be called as follows:

```
[zopt,fopt] = duquad(H,c,G,g,[],ub_hat,lb,[],z0);
```

If the optimization variable is unbounded, the linear constraints are $G \leq g$, and there is no initial point, the function can be called as:

```
[zopt,fopt] = duquad(H,c,G,g);
```

In conclusion, DuQuad is flexible towards the inputs. Furthermore, grab all possible outputs by calling the function as:

```
[zopt,fopt,exitflag,output,lambda1,lambda2] = duquad(H,c,G,g);
```

An overview of all inputs and outputs is viewed in Matlab console if the user is running the Matlab command:

```
help duquad
```

Include Options

DuQuad can also take different options as input, e.g. maximum number of iterations, tolerance for stopping criteria etc. All these options are collected in a struct (Table C.2) as follows:

```
% Maximum number of iterations in the outer loop
options.maxiter_outer = 1000;
% Maximum number of iterations in the inner loop
options.maxiter_inner = 100;
% Tolerance for dual suboptimality
options.eps_ds = 0.0001;
% Tolerance for primal feasibility
options.eps_pf = 0.001;
% Tolerance for primal feasibility in the inner problem
options.eps_inner = 0.0001;
% Penalty parameter used in ALM and FALM
options.rho = 1;
% Specifies the algorithm used to solve the problem.
options.algorithm = 1;
```

The option struct is included as input number 10 in the function:

```
[zopt,fopt] = duquad(H,c,G,g,[],[],[],[],[],options);
```

Note that the user must either specify all options or no options (when default values are utilized). This should be improved in a newer version of DuQuad.

C.3. Specifications

In this section, the different inputs and outputs are listed. DuQuad's full potential, regarding inputs and outputs, is utilized by running the following example Matlab command:

```
[zopt,fopt,exitflag,output,lambda1,lambda2]...
    = duquad(H,c,G,g,lb_hat,ub_hat,lb,ub,z0,options);
```

C.3.1. Inputs

Table C.1 gives an overview of the different inputs to the function. In addition the dimensions for each input is listed. The last input to the function is a struct called *options*, which set some criteria for the solving process. The different options are summarized in Table C.2. One of the options is to choose which algorithm that is solving the problem. This is specified by a number ranging from 1-8, and is listed in Table C.3.

Table C.1: The inputs for the duquad function

Input	Name	Desciption	Dimension
1	H	Hessian matrix	$n \times n$
2	c	Gradient vector	n
3	G	Linear constraints matrix	$m \times n$
4	g	Linear constraints vector	m
5	$\hat{lb}$	Lower bound for the linear constraints	m
6	$\hat{ub}$	Upper bound for the linear constraints	m
7	lb	Lower bound for optimization variable z	n
8	ub	Upper bound for optimization variable z	n
9	z_0	Initial point	n
10	options	Struct containing options for solver, see Table C.2	

Table C.2: Overview of the parameters in the *options* struct

Name	Description	Default
maxiter_outer	Maximum number of iterations in the outer loop	1000
maxiter_inner	Maximum number of iterations in the inner loop	100
eps_ds	Tolerance for dual suboptimality	0.0001
eps_pf	Tolerance for primal feasibility	0.001
eps_inner	Tolerance for primal feasibility in the inner problem	0.00001
rho	Penalty parameter used in ALM and FALM	1
algorithm	Specifies the algorithm used to solve the problem.	3

Table C.3: Values of the *algorithm* parameter from the option struct in Table C.2

Algorithm	**Value**
DGM last	1
DGM average	2
FDGM last	3
FDGM average	4
ALM last	5
ALM average	6
FALM last	7
FALM average	8

C.3.2. Outputs

The outputs of DuQuad is summarized in Table C.4. Among the outputs is a struct called *output*. This struct contains some results from the solving process.

Table C.4: The outputs of the duquad function

Output	**Name**	**Description**
1	z^*	Optimal point
2	f^*	Optimal value
3	exitflag	1 = solution found, 2 = max num of iteration reached, -1 = error
4	output	Struct containing various result, see Table C.5
5	λ_1	Set of Lagrangian multipliers
6	λ_2	Set of Lagrangian multipliers

Table C.5: Content of the output struct **output**

Name	**Description**
iterations	Number of outer iterations
iterations_inner_tot	Total number of iterations for the inner problem
time	Runtime of the algorithm after all initialization is done
time_tot_inner	Total time spent on solving the inner problem
flag_last_satisfied	Flag specifies which stopping criteria was resolved last. Value: 0 = dual suboptimality, 1 = primal feasibility
niter_feasible_ds	Number of iterations the criterion for dual suboptimality was satisfied
niter_feasible_pf	Number of iterations that the criterion for primal feasibility was satisfied
exitflag_inner	Exitflag for the inner problem. Values: 1 = feasible point found, 2 = Maximum number of iterations exceeded
num_exceeded_max_niter_inner	Total number of times the inner problem exceeded the number of iterations
ds_vector	Vector storing all the value of the dual suboptimality every iteration
pf_vector	Vector storing all the value of the primal feasibility every iteration
algorithm	Name of the algorithm used to solve the problem