DEGREE PROJECT IN ELECTRICAL ENGINEERING,
SECOND CYCLE, 30 CREDITS
*STOCKHOLM, SWEDEN 2018*

# Efficient Convex Quadratic Optimization Solver for Embedded MPC Applications

**ALBERTO DÍAZ DORADO**

**KTH ROYAL INSTITUTE OF TECHNOLOGY**
**SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE**

# Efficient Convex Quadratic Optimization Solver for Embedded MPC Applications

ALBERTO DÍAZ DORADO

Master in Electrical Engineering
Date: October 2, 2018
Supervisor: Arda Aytekin , Martin Biel
Examiner: Mikael Johansson
Swedish title: Effektiv Konvex Kvadratisk Optimeringslösare för Inbäddade MPC-applikationer
School of Electrical Engineering and Computer Science

# Abstract

Model predictive control (MPC) is an advanced control technique that requires solving an optimization problem at each sampling instant. Several emerging applications require the use of short sampling times to cope with the fast dynamics of the underlying process. In many cases, these applications also need to be implemented on embedded hardware with limited resources. As a result, the use of model predictive controllers in these application domains remains challenging.

This work deals with the implementation of an interior point algorithm for use in embedded MPC applications. We propose a modular software design that allows for high solver customization, while still producing compact and fast code. Our interior point method includes an efficient implementation of a novel approach to constraint softening, which has only been tested in high-level languages before. We show that a well conceived low-level implementation of integrated constraint softening adds no significant overhead to the solution time, and hence, constitutes an attractive alternative in embedded MPC solvers.

# Sammanfattning

Modell prediktiv reglering (MPC) är en avancerad regler-teknik som involverar att lösa ett optimeringsproblem vid varje sampeltillfälle. Flera nya tillämpningar kräver användning av korta samplingstider för att klara av den snabba dynamiken av den underliggande processen. I många fall implementeras dessa tekniker på inbyggd hårdvara med begränsade resurser, där det som följd är utmanande att utnyttja MPC.

Det här arbetet berör implementering av en inrepunktsmetod för att lösa optimeringsproblem i inbyggda MPC-tillämpningar. Vi föreslår en modulär design som gör det möjligt att skräddarsy lösaren i detalj, men ändå samtidigt producera kompakt och snabb kod. Vår inrepunktsmetod inkluderar en effektiv implementering av ett nytt tillvägagångssätt för att lätta på optimeringsvillkor. Denna method har tidigare endast implementerats i högnivåspråk. Vi visar att denna integrerade metod för att lätta på optimeringsvillkor inte medför någon signifikant ökning av lösningstiden och därmed är en attraktiv teknik för inbyggda MPC-lösare.

# Acknowledgements

First and foremost, I would like to express my gratitude to Professor Mikael Johansson for accepting me as a Master's Thesis student for this exciting and challenging project. He was not only a supportive and encouraging Examiner, but provided insightful feedback and inspirational leadership. I would also like to thank my supervisors, Martin Biel and Arda Aytekin, for helping and guiding me throughout the project. They teached me, but also pushed me beyond my limits, for which I will always be indebted.

I am also very grateful for the Department of Automatic Control as a whole, who welcomed me better than I could have wished. Special thanks go to my opponent Diego Gónzalez and my fellow Paul Verrax, with whom I shared hard work and entertaining discussions.

At last, I am immensely thankful for my parents, Alberto and Montse, and my siblings, Montsita and Rafa. They made this project possible, but, even more importantly, they make it very meaningful to me.

# Contents

# Chapter 1

# Introduction

This work addresses the implementation of an interior point algorithm for the solution of multi-stage quadratic programming (QP) problems. Of particular interest are the QPs that arise in the context of model predictive control (MPC) applications. We are interested in algorithms tailored to the special structure of this particular family of problems; hence, we do not intend to target general QPs. We focus on linear, time-invariant (LTI) system models with convex quadratic penalty terms in MPC applications. These applications result in convex QPs.

In the thesis, we first provide the necessary background on MPC and quadratic optimization. Later, we review the existing algorithms in the literature. Finally, we present the architecture for the implementation, which aims at tackling a variety of MPC problems in a flexible way without sacrificing much from efficiency.

In the subsequent sections, we try to motivate the need for a flexible and efficient solver by giving an MPC example. Then, we list the goals and the outcome of the work, and finally give a brief outline of the contents of the thesis.

## 1.1 Motivation

Model predictive control is an advanced control technique that naturally handles multi-variable, multi-objective control problems. It consists of four major components: *a value function*, *an internal system model*, *a horizon length* and *a constraint set*. The value function encodes the penalties on deviations of the inputs and the system's states from their desired values. At each sampling instant, MPC uses its internal system

model and the sampled state to predict the behaviour of the system's future states in the coming fixed-length horizon. Using these predictions, MPC minimizes its value function with respect to the possible input values, while satisfying the constraints defined by its constraint set, and then applies the optimal input to the system.

There are few characteristics that make MPC a superior alternative to traditional controllers, *e.g.*, PID-type controllers. First, MPC naturally handles MIMO (multiple-input, multiple-output) plants, whereas PID controllers are best suited for SISO (single-input, single-output) plants. Second, MPC readily incorporates system and design constraints to the problem formulation. This allows for safely exploiting all the slack provided by the constraints, rather than sticking to a conservative operation area, which translates into a more efficient control of the system. Last, MPC provides the optimal input for a given choice of control goals and associated weights. In contrast, PID controllers are always suboptimal for any given value function.

The main drawback of MPC, which has prevented it from seeing wide use accross industries, is that it involves solving an optimization problem at each sampling instant. Optimization techniques are computationally expensive and time consuming. Furthermore, optimization problems may be infeasible or unbounded, in which case the solvers may fail to provide a feasible input to the system. Because of these reasons, MPC has been traditionally used in the chemical and process industry, where the system dynamics are inherently slow and sampling times range from few minutes to hours. This enables the solver to provide an accurate solution to the optimization problem, or even reformulate the problem if it is found infeasible.

Thanks to the improvements in optimization algorithms as well as the superior capacity of computers and microprocessors, MPC has been employed in diverse domains spanning marine applications [1]–[3], aircraft control [4] and power electronic devices [5]. These applications have different requirements on the sampling intervals. For instance, while the marine applications require a sampling time in the order of seconds, aircraft control and power electronics devices demand sampling times in the order of milliseconds.

In spite of this progress, the solution of the optimization problem is still the bottleneck that prevents a more widespread utilization of MPC techniques. Most automatic controllers are not executed in computers, which have a superior computational power, but in embedded devices,

such as PLCs, micro-controllers, FPGAs, and ASICs [6]. These devices have limited memory and limited computation capability, which challenges the task of solving the optimization problem in time. For this reason, developing efficient and robust quadratic programming (QP) solvers is they key to the widespread utilization of MPC controllers.

While general purpose solvers may be used to solve optimization problems arising from MPC, these problems have a special structure that can be exploited for better efficiency. In other words, we can take advantage of the knowledge about the structure of the problem to reduce the computational effort of the algorithms. This is an active area of research. There have been proposals to adapt traditional techniques for solving QP problems to more specialized MPC-type problems. This includes, among others, explicit MPC, first-order methods, active-set methods and interior-point methods.

Even though they share a uniform structure, different MPC formulations may lead to different optimization problems. This work focuses on the so-called Linear MPC formulations, which involve convex quadratic value functions, linear constraints and LTI internal system models.

There are two main reasons for this choice. In the first place, this type of MPC formulation covers a great variety of real-world problems, since detailed nonlinear models are not always available and linearizing the dynamics around a working point is a widespread technique in the control community. In the second place, one popular approach to solve nonlinear MPC problems is to solve a sequence of convex quadratic optimization problems. This is the case of sequential quadratic programming (SQP) [7, Chapter 18], but also of some nonlinear interior point methods such as LANCELOT [8].

Within the family of Linear MPC, there are many variations and simplifications that make the arising optimization problem easier to solve. These simplifications are not fanciful, but rather obey common and sensible MPC formulations. For instance, the most common penalty is the Euclidean norm of states and inputs. While it is also possible to put linear penalties on cross-products of states and inputs, this is rather unusual. In addition, physical input constraints are usually just upper and lower bounded. A more general formulation allows for constraints involving linear combinations of inputs and states but this is also a less common scenario. Taking advantage of these particularities, rather than treating all QP problems in a generalized way, allows for better efficiency.

In this work, the proposed solver benefits from a flexible design that takes advantage of all possible nuances in the MPC formulation, while also solving the most general case. The design is highly modular a well suited for future expansions.

## 1.2 Goals of the Thesis

The goal is to write an efficient QP solver for Linear MPC applications, that exploits the structure of the problem as much as possible. In particular, the solver shall be:

1. Effective: it shall provide the correct solution to the target optimization problems.
2. Robust: it shall be able to handle infeasible initial points, provide a feasible solution in case of early termination and, ideally, provide some support in case of infeasibility or unboundeness. If the solver cannot handle any kind of degeneracies, they must be specified.
3. Flexible: it shall cover as many different MPC formulations as possible. The solver shall also allow for some parameters tuning such as, for instance, trade-off between constraint relaxation and optimality tolerances.
4. Efficient: it shall provide a solution with respect to the given tolerance as fast as possible. To do so, we shall take full advantage of the MPC formulation in order to
   (a) reduce the number of iterations, and,
   (b) reduce the cost of each individual iteration.

## 1.3 The Target Optimization Problem and Limitations

The details of the MPC formulation and the arising QP will be discussed in detail in the later chapters. By now, it suffices to know that the final solver shall be able to handle MPC problems in the form:

$$
\begin{aligned}
\underset{U,X}{\text{minimize}} \quad & \textstyle\sum_{k=0}^{T-1} \left\{ u_k^\top R u_k + x_k^\top Q x_k \right\} + x_T^\top P x_T \\
\text{subject to} \quad & x_{k+1} = A x_k + B u_k, && k = 0, \ldots, T-1 \\
& l_u \leq u_k \leq u_u, && k = 0, \ldots, T-1 \\
& F_u u_k \leq f_u, && k = 0, \ldots, T-1 \\
& l_x \leq x_k \leq u_x, && k = 0, \ldots, T-1 \\
& F_x x_k \leq f_x, && k = 0, \ldots, T-1 \\
& l_t \leq x_T \leq u_t, \\
& F_t x_k \leq f_t
\end{aligned}
\tag{1.1}
$$

In other words, it should support handling input, state, and terminal constraints (both hard and soft), and shall distinguish between box and linear constraints. In addition, we will show how output constraints, output penalties, tracking problems and input-rate penalties can be included in the given formulation.

We would also like to state the implicit limitations of formulation (1.1). It

1. only allows for convex quadratic optimization problem definition,
2. does not provide a wrapper for nonlinear MPCs,
3. does not allow for integer constraints,
4. does not allow for crossed input-state terms in the objective function; *i.e.*, it allows the formulation

   $$\begin{pmatrix} u[k] & x[k] \end{pmatrix} \begin{pmatrix} R & S\top \\ S & Q \end{pmatrix} \begin{pmatrix} u[k] \\ x[k] \end{pmatrix}$$

   uniquely for $S = 0$,
5. does not allow for quadratic constraints, and,
6. does not allow for semidefinite penalties.

## 1.4 Thesis outline

The thesis has the following structure:

1. *Background*: Chapter 2 gives the theoretical background on model predictive control and quadratic programming problems. It also provides further requirements of an efficient solver for embedded optimization and presents the major challenges and difficulties of the task.

2. *State of the Art*: Chapter 3 reviews existing approaches for convex QP in the MPC framework, noting the advantages and disadvantages of each method.

3. *Proposed Approach*: Chapter 4 presents the proposed optimization method in detail.

4. *System architecture*: Chapter 5 describes the software architecture. Its modular design is reported, with emphasis on how it helps to attain the goals of efficacy, robustness, flexibility and efficiency.

5. *Numerical results*: In Chapter 6, we report the performance of the software on numerical experiments.

6. *Conclusions:* Chapter 7 summarizes the results of the work, critically analyzes the outcome (the final software) and suggests future work.

# Chapter 2

# Background

This chapter introduces Model Predictive Control (MPC) and the structure of the arising Quadratic Programs (QP). Additionally, the requirements of efficient, embedded optimization are reviewed.

## 2.1 Model Predictive Control

### 2.1.1 Optimal Control

In the design of a *controller*, or control system, it is natural to speak of *control goals*. The task of the controller is to decide on the *control actions* that shall be executed in order to fulfill some control goal. The control goals describe the performance that is expected (or desired) from the system under control. Control goals are usually specified as desired output values, such as the power output of a turbine, the level of a tank, the trajectory of a car, etc.

Although regarded as an advanced control technique, *optimal control* is a rather intuitive idea. Over a set of feasible control actions, it is natural to ask the controller to choose the action that *minimizes* the deviation of the system from the given references. Optimal control naturally handles multiple control goals at a time, whose relative importance is weighted in the so-called *cost function*, allowing for complex control goal formulations such as "*the power output of the turbine shall stay as close as possible to the reference, without exceeding 1200°C in the combustion chamber and consuming as little fuel as possible*".

Optimal control typically exploits any knowledge about the dynamics of the system to predict its future behaviour and minimize the

value of the cost function over time. The time span considered by the controller is referred to as *prediction horizon*. For the classical formulation of optimal control problems [9], let the system be described by a state-space model consisting on a vector of states $x$, a vector of inputs $u$ and the nonlinear dynamics:

$$\dot{x}(t) = \mathrm{f}(x, u, t)$$

The cost function is defined as a weighted sum of control offsets over the prediction horizon length. If nonlinear penalties are allowed, the cost function is given by the general formulation:

$$\mathrm{V}(x, u, t) \coloneqq \int_{t=0} \top\, \ell(x(t), u(t), t))dt + \ell_{\mathrm{f}}(x(T))$$

Where $T$ stands for the prediction horizon length, $\ell$ is called the *stage cost* and $\ell_{\mathrm{f}}$ is referred to as the *final cost*. Additionally, inputs and states might be constrained to belong to some sets $\mathcal{U}(t)$ and $\mathcal{X}(t)$, respectively, which have been made time-dependant for the sake of generality.

The optimal control problem is formulated as the nonlinear program:

$$\begin{aligned}
\underset{u(t),\, t\in[0,T)}{\text{minimize}} \quad & \mathrm{V}(x, u, t) \\
\text{subject to} \quad & \dot{x}(t) = \mathrm{f}(x(t), u(t), t) \\
& u(t) \in \mathcal{U}(t)\,, && t \in [0, T) \\
& x(t) \in \mathcal{X}(t)\,, && t \in [0, T) \\
& x(T) \in \mathcal{X}_F
\end{aligned} \tag{2.1}$$

However, the problem formulation (2.1) is in many cases intractable. In other words: a general problem formulation can only be solved in some particular cases. We need to make assumptions about the functions f, $\ell$ and $\ell_f$, and about the sets $\mathcal{U}$, $\mathcal{X}$ and $\mathcal{X}_F$, in order to be able to handle the optimization problem.

The difficulty of general optimal control comes from the fact that (2.1) is a *function* optimization problem. Thus, a very common approach is to discretize the system, so that the optimization is carried out over a finite set of parameters (the so-called *decision variables*). In addition, discretizing the problem is a natural approach, since computers or microprocessors will execute the optimization in any real application. The resulting formulation is the optimization problem 2.2:

$$\begin{aligned} \underset{u_k,\, k=0,1\ldots T}{\text{minimize}} \quad & \sum_{k=0} \top \, \mathrm{l_k}(u_k, x_k) \\ \text{subject to} \quad & \mathrm{f_k}(u_k, x_k, x_{k+1}) = 0 \quad k = 0, 1 \ldots T \\ & u_k \in \mathcal{U}_k \,, \qquad\qquad k = 0, 1 \ldots T \\ & x_k \in \mathcal{X}_k \,, \qquad\qquad k = 0, 1 \ldots T \end{aligned} \tag{2.2}$$

Where $k$ represents the *step* in a sequence of discrete variables, rather than time. The function $\mathrm{l}_k$ represent the *stage cost* and $\mathrm{f}_k$ gives the discretized system dynamics. No assumptions are made regarding these two functions. Moreover, the sets $\mathcal{U}_k$ and $\mathcal{X}_k$ are completely general. Though formulation (2.2) is more practical than (2.1), it still remains a nonlinear optimization problem, which are hard to solve. In fact, if we make no further assumptions, problem 2.2 belongs to the class of NP-hard problems, for which no efficient algorithms exists. One of the major simplification of optimal control problems, which even allows for an analytical solution, is the linear quadratic regulator (LQR).

### 2.1.2 Linear Quadratic Regulator (LQR)

The Linear Quadratic Regulator (LQR) simplifies the formulation (2.2), in a number of ways. First, the cost function becomes a convex, quadratic polynomial. Second, the system dynamics are discretized. Finally, the only constraints in the problem are the system dynamics. The LQR is usually formulated as follows:

$$\underset{U,X}{\text{minimize}} \quad \sum_{k=0}^{T-1} \{u_k \top R u_k + x_k \top Q x_k\} + x_T \top P x_T \tag{2.3}$$

$$\text{subject to} \quad x_{k+1} = A x_k + B u_k, \qquad k = 0, 1 \ldots T-1 \tag{2.4}$$

$$x_0 = \tilde{x}(t) \tag{2.5}$$

Where $U \coloneqq \begin{bmatrix} u_0^\top & u_1^\top & \ldots & u_{T-1}^\top \end{bmatrix}^\top$, $X \coloneqq \begin{bmatrix} x_0^\top & x_1^\top & \ldots & x_T^\top \end{bmatrix}^\top$ and $\tilde{x}(t)$ is the state estimate at time $t$. We assume that the *penalty matrices* are symmetric and positive definite: $R = R^\top \succ 0$, $Q = Q^\top \succ 0$ and $P = P^\top \succ 0$. If they are not symmetric, we can reformulate the problem such that they are symmetric, replacing $R$ by $(R+R^\top)/2$, which

produces the same minimizer of the problem. Positive definitness is a mathematical condition that guarantees that the problem is strictly convex and, hence has a unique minimizer.

This formulation is sometimes referred to as *finite-horizon* LQP. Another variant is the so-called *infinite-horizon* LQP, where $T = \infty$ and we drop the final penalty cost $P$. In both cases, there is an analytical solution for the problem, which is expressed as the feedback law $u_k^* := -L_k x_k$, where the superscript $^*$ indicates that the referred variable is optimal. The feedback matrices $L_k$ are obtained using the *Discrete Algebraic Riccati Equation* (see [10] for a detailed explanation).

The optimization problem (2.3) - (2.5) has an analytical solution as a state linear feedback. This circumvents the computational complexity that prevents optimal control from been applied. Nevertheless, LQR has little application in practice, because few real world problems fit into such a restricting simplification. The abscence of constraints on inputs and states is limiting. For example, a valve can only be opened between $0\%$ and $100\%$, which cannot be captured by LQR.

Model Predictive Control can be understood as an expansion of LQR, that increments the computational burden for the sake of a more realistic approach.

### 2.1.3 Model Predictive Control (MPC)

There are two main additions to LQR that make it applicable in practice: the inclusion of constraints and the receding horizon idea. The resulting controller may be referred to as Linear-Time-Invariant Model Predictive Controller (LTI-MPC), which is the most elementary version of MPC.

Constraints naturally come into play when controlling real systems. May it be by definition of the system (*e.g.*, the opening of a valve), because of physical limitations (*e.g.*, the maximum power supplied by the electrical network) or because of control goals themselves (*e.g.*, the maximum temperature allowed in the combustion chamber of a turbine), there are always limitations on the system. Such limitations can be neglected, and the system may be controlled as if they did not exist. That is the case of traditional PID controllers, which provide a control action without considering whether it will end in the violation of some constraints or not. However, this usually results in a bad performance of the controller. For example, if the input saturates, the closed-loop control is broken, meaning that the control action is

no longer related to the control error as it should be in the original controller design.

In Linear MPC, we consider just linear constraints; *i.e.*the feasible sets are described as $\mathcal{X} = \{x \mid Fx \leq f\}$. The resulting problem formulation is as follows:

$$\begin{aligned} \underset{U,X}{\text{minimize}} \quad & \textstyle\sum_{k=0}^{T-1} \left\{u_k^\top R u_k + x_k^\top Q x_k\right\} + x_T^\top P x_T \\ \text{subject to} \quad & x_{k+1} = Ax_k + Bu_k, \quad k = 0, 1 \ldots T-1 \\ & F_u u_k \leq f_u, \qquad\qquad k = 0, 1 \ldots T-1 \\ & F_x x_k \leq f_x, \qquad\qquad k = 0, 1 \ldots T-1 \\ & F_t x_T \leq f_t \\ & x_0 = \tilde{x}(t) \end{aligned} \tag{2.6}$$

In the rest of the text, we will use the following nomenclature:

- $T > 0$ is referred to as the *prediction horizon*.
- $u_k \in \mathbb{R}^m$ is the *input vector* at step $k$. Similarly, $x_k \in \mathbb{R}^n$ is the *state vector* at step $k$. There are $m$ inputs and $n$ states in the system.
- Capital letters $U \in \mathbb{R}^{T \cdot m}$ and $X \in \mathbb{R}^{T \cdot n}$ are used to represent the *sequences* of inputs and states, respectively. That is:

$$\begin{aligned} U &\coloneqq \begin{bmatrix} u_0^\top & u_1^\top & \ldots & u_{T-1}^\top \end{bmatrix} \top \\ X &\coloneqq \begin{bmatrix} x_1^\top & x_2^\top & \ldots & x_T^\top \end{bmatrix} \top \end{aligned}$$

- We use a superscript asterisk to denote the *optimal solution* to the optimization problem. For instance, $u_k^*$ stands for the optimal input at time step $k$, and $U^*$ stands for the optimal sequences of inputs up to the prediciont horizon length.
- The matrices $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times m}$ represent the *system dynamics*. $A$ describes the influence of the current state on the next state, and $B$ describes the influence of the input on the next state.
- $R \in \mathbb{R}^{m \times m}$, $Q \in \mathbb{R}^{n \times n}$ and $P \in \mathbb{R}^{n \times n}$ are the input, state and terminal state *penalty matrices*. They weight the relative importance of different deviations from the control goals.
- The matrices $F_u \in \mathbb{R}^{\kappa_u \times m}$, $F_x \in \mathbb{R}^{\kappa_x \times n}$ and $F_t \in \mathbb{R}^{\kappa_t \times n}$ are the *coefficient matrices* for the *input constraints*, *state constraints* and

*terminal constraints*, respectively. The vectors $f_u \in \mathbb{R}^{\kappa_u}$, $f_x \in \mathbb{R}^{\kappa_x}$ and $f_t \in \mathbb{R}^{\kappa_t}$ are the right hand sides to the referred constraints.

- $x_0 \in \mathbb{R}^n$ is referred to as the *initial state*. $\tilde{x}(t) \in \mathbb{R}^n$ is the *state estimate* at time $t$, which is feeded to the optimization problem by an observer.

- We will use caligraphic letters $\mathcal{U}$, $\mathcal{X}$ and $\mathcal{X}_t$ to represent the input, state and terminal state *feasible sets*, given by:

$$\begin{aligned}\mathcal{U} &= \{u \in \mathbb{R}^m \mid F_u u \leq f_u\} \subset \mathbb{R}^m \\ \mathcal{X} &= \{x \in \mathbb{R}^n \mid F_x x \leq f_x\} \subset \mathbb{R}^n \\ \mathcal{X}_t &= \{x \in \mathbb{R}^n \mid F_t x \leq f_t\} \subset \mathbb{R}^n\end{aligned}$$

### 2.1.4 The Receding Horizon Idea

The program (2.6) states an open-loop optimal control problem. The solution sequence $U^*$, though optimal, has important problems that make it unsuited as a sequence of control actions.

First, it is short sighted. The controller can only see $T$ time steps ahead of time; whatever happens after it is not considered in the optimization. It may happen that a certain sequence is optimal for the next five time steps, but drives the system out of the controllable set in its aim to minimize the value of the cost function. In contrast, ten steps prediction horizon may be enough to recognize that scenario and drive the system to a safer, though more expensive, trajectory. Although this problem may be alleviated by increasing the prediction horizon length, the same argument still holds. For practical reasons, long prediction horizons are preferred over short ones, but any finite prediction horizon will be short-sighted in the sense explained above. Infinite-horizon optimization problems are, in turn, infeasible to solve in practice.

It shall be remarked here that LQR succesfully solve infinite-horizon problems by appropriately chosen the final penalty matrix $P$. As explained in [10, Chapter 9], choosing $P$ as the solution to the Discrete Algebraic Riccati Equation (DARE) is equivalent to solving an infinite horizon optimization problem. This approach does not suffice for constrained MPC problems, since the DARE does not take into account the system constraints. To make the finite-horizon MPC equivalent to an infinite-horizon MPC problem, we must introduce an appropriate terminal set. However, the choice of the terminal set is non-trivial.

Moreover, the introduction of complex terminal sets greatly increments the complexity of the numerical solution to the problem. For these reasons, the choice of the terminal set is most often based on heuristics, which does not guarantee that the solution is the same as in the inifite-horizon scenario.

Secondly, the series $U^*$ is an open-loop sequence, and, as such, it is weak againts model inaccuracies and disturbances. In presence of these, a successive application of the optimal inputs will not result in the predicted series of states $X^*$. Bear in mind that an input $u_k^*$ is optimal for the corresponding state $x_k^*$. When the system deviates from its predicted, optimal trajectory, the inputs are no longer optimal, which can lead to a degradated performance or even instability.

Both problems are addressed at a time by the *receding horizon* idea. As stated in [10, Chapter 12]: "*An infinite horizon suboptimal controller can be designed by repeatedly solving finite time optimal control problems in a receding horizon fashion (...)*", which refers to the the problem of the short-sighted controller. Additionally, the receding horizon strongly improves the robustness of the controller by introducing feedback (and, thus, closed-loop control) in an indirect way. The closed-loop control structure is depicted in Figure 2.1.

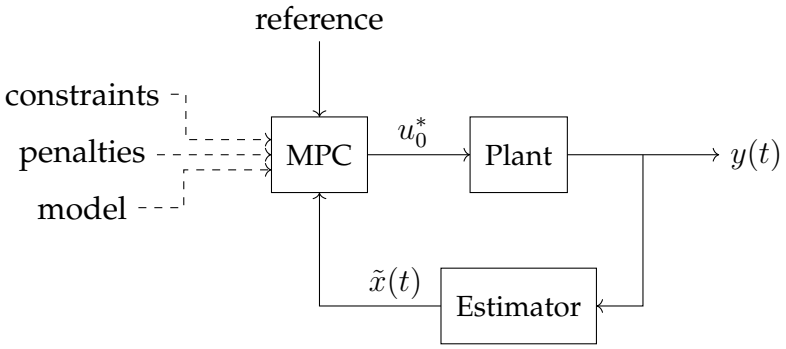


Figure 2.1: Closed-loop operation of an MPC controller

Under the receding horizon approach, a finite-horizon optimization problem is solved at each sampling instant. The first element in the optimal input sequence, $u_0^*$, is applied to the system during the next sampling interval. However, the rest of the sequence (*i.e.*, $u_1, u_2, \ldots u_{T-1}$) is discarded and a new optimization problem is solved at time $t + T_s$, where $T_s$ is the sampling time. The new optimization problem uses the new state estimate, $\tilde{x}(t + T_s)$, as the initial state, thus

indirectly introducing feedback as any mismatch between the actual $x(t + T_s)$ and the predicted $x[k + 1]$ will not propagate into the future.

### 2.1.5 Enhancements of LTI-MPC

The MPC problem that we address in this work has the form (2.6). However, as it has been described, only *regulation problems* fit into that formulation, with penalties and constraints limited to states and inputs. In practical formulations of MPC there are many other desirable features, such as reference tracking, input rate penalties and constraint softening. It will be shown here that all those problem formulations can be translated into formulation (2.6).

#### The tracking problem

In the regulation problem, the control goal is to drive the system to the origin, which is achieved by penalizing any deviation of states and inputs from the zero vector. Note that this approach is not a particular control problem, but rather a very common one. Usually we linearize the plant around a working point and want the controller to keep the plant at that state. Hence, penalizing deviations from the linearization point is a widely used control technique.

In the tracking problem, however, the control goal is that the system outputs track a given reference (or trajectory). Additionally, note that the optimal input that keeps the system at its reference in steady state is different from zero in this case. Though one may compute what is the optimal reference for the input and penalize any deviation from such, it is often the case that the system oscillates around the reference due to model inaccuracies and distrubances. To circumvent this problem, the *input change rate* is penalized, rather than the input itself, which introduces a kind of *integral action* in the controller.

The system output $y_k \in \mathbb{R}^p$ is the set of mesurable signals that are used to estimate the system state. In general, the system state signals are not directly available for direct measurement (for example, the magnetic flux in an induction machine), but can be estimated using measurements of system outputs (for instance, the stator currents in such machine). In a linear state-space model, the output is a linear combination of inputs and states, namely:

$$y_k = Cx_k + Du_k$$

However, the system is usually reformulated such that $D$ is the zero matrix, and thus the system output is described as a linear combination of system states:

$$y_k \coloneqq Cx_k$$

The matrix $C \in \mathbb{R}^{n \times p}$ is sometimes referred to as the *output matrix*. In tracking problems it is a common practice to set references for the system outputs rather than for the system states, though that can be done as well.

Furthermore, the input rate $\Delta u \in \mathbb{R}^n$ is defined as the difference between the current and the last input to the system; *i.e.*:

$$\Delta u_k \coloneqq u_k - u_{k-1}$$

The input rate at $k = 0$ is only well defined if information about the last input is available.

We will show that output reference tracking and input rate penalties can be introduced into the formulation (2.6). First, note that any penalties and constraints on $y$ might be mapped to penalties and constraints on $x$ using the output matrix $C$. For a given reference $r_y$, let the output penalty for step $k$ be denoted by $\mathrm{V}_k$:

$$\begin{aligned}
\mathrm{V_k}(y_k) &= (y_k - r_y)^\top Q_y (y_k - r_y) \\
&= y_k^\top Q_y y_k - 2 r_y^\top Q_y y_k + r_y^\top Q_y r_y \\
&= (Cx_k)^\top Q_y (Cx_k) - 2 r_y^\top Q_y (Cx_k) + r_y^\top Q_y r_y \\
&= x_k^\top \left(C^\top Q_y C\right) x_k - (2 R_y C r_y)^\top x_k + r_y^\top Q_y r_y
\end{aligned}$$

Where $Q_y$ is the output penalty matrix. By identifying

$$\begin{aligned}
Q_x &\coloneqq C^\top Q_y C \\
r_x &\coloneqq -2 Q_y C y_0 \\
r_0 &\coloneqq y_0^\top Q_y y_0
\end{aligned}$$

The penalty on the output control error is rewritten as a penalty on the state with an additional linear term (note that the last term is constant and thus can be dropped from the cost function):

$$\begin{aligned}
\mathrm{V_k}(y_k) &= \mathrm{V_k}(x_k) && (2.7) \\
&= x_k^\top Q_x x_k + r_x^\top x_k + r_0 && (2.8)
\end{aligned}$$

We do a similar transformation on the output constraints:

$$F_y y_k \leq f_y \rightarrow F_y \left(C x_k\right) \leq f_y \tag{2.9}$$
$$\rightarrow \left(F_y C\right) x_k \leq f_y \tag{2.10}$$

Where $F_y C$ is the matrix of coefficients for the inequality expressed in terms of the system states. (2.7) and (2.9) show how to include output penalties and output constraints, respectively, in formulation (2.6).

Second, note that the input rate can be effectively penalized and constrained, by means of a system augmentation. By writing $u_k \coloneqq \Delta u_k + u_{k-1}$, one may rewrite the system dynamics as:

$$\begin{aligned} x_{k+1} &\coloneqq A x_k + B u_k \\ &= A x_k + B \Delta u_k + B u_{k-1} \end{aligned}$$

Then, considering the input rate $\Delta u_k$ as the input to the system, and $u_{k-1}$ as another state, the system may be augmented to:

$$\begin{aligned} \begin{bmatrix} x_{k+1} \\ u_k \end{bmatrix} &= \begin{bmatrix} A & B \\ 0 & I \end{bmatrix} \begin{bmatrix} x_k \\ u_{k-1} \end{bmatrix} + \begin{bmatrix} B \\ I \end{bmatrix} \Delta u_k \\ &= A_a x_a + B_a \Delta u_k \end{aligned}$$

Where $x_a \coloneqq \begin{bmatrix} x_k & u_{k-1} \end{bmatrix}^\top$ is the augmented state vector and $A_a$ and $B_a$ are the augmented dynamics matrices. The penalty matrices and the system constraints shall be written in terms of the augmented system, which meets the formulation (2.6). As a disadvantage of this approach, note that is increases the state size to from $n$ to $m + n$, which results in a larger, more difficult to solve optimization problem.

### Constraint softening

Constraint softening is a very popular technique in industrial MPC applications. It can guarantee that the optimization problem is always feasible, and hence, that the controller always provides a valid input to the system. This kind of robustness against infeasibility is one of the major concerns in MPC applications.

There are different ways to implement soft constraints. The most common approach is to introduce additional slack variables, which represent the violation of the soft constraints, and penalize them. This approach increases either the state of the input space, hence resulting in a larger optimization problem (see [11]).

In this work, we use a novel approach to constraint softening, introduced in [11], that uses no additional variables. A detailed explanation of this method is given in Chapter 4, Section 3.

## 2.2 Quadratic Programming in the MPC Framework

### 2.2.1 Optimization and Quadratic Programming

Quadratic programing is a branch of optimization where the variables are real (as opposed to integers), the cost function is quadratic (that is, a polynomial of second order) and the constraints are limited to linear equalities and inequalities. A QP with $n$ variables and $m$[1] constraints has the general form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \mathrm{V}(x) = x\top Hx + h^\top x \\ \text{subject to} \quad & Ax \leq b \\ & x \in \mathbb{R}^n \end{aligned} \tag{2.11}$$

Where $x$ is the optimization variable, and $H \in \mathbb{R}^{n\times n}$, $h \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$. Occasionally, $H$ and $A$ are referred as the problem *Hessian* and *constraint matrix*, respectively. Sometimes equality constraints are given together with the inequality constraints, as in

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \mathrm{V}(x) = x\top Hx + h^\top x \\ \text{subject to} \quad & Ax \leq b \\ & Cx = d \\ & x \in \mathbb{R}^n \end{aligned} \tag{2.12}$$

With $C \in \mathbb{R}^{p\times n}$ and $d \in \mathbb{R}^p$. However, instances of the form (2.12) can always be transformed into the form (2.11) by solving the problem in a lower dimensional subspace. In particular, the equality constraints can be disregarded by solving the problem in the null space of $C$ (see [12] for further details). For this reason, we will only consider inequality constraints in the following discussion, without any loss of generality.

[1] Note that, in this section, $n$ and $m$ do not stand for the number of states and inputs in a state-space model. The same letters are used here for a different purpose because such is the typical nomenclature in optimization problems.

Let $\mathcal{I} \coloneqq \{1, \ldots, m\}$ be the set of inequality constraints. Then we define:

**Definition 1** (Feasible point)**.** We say that a point $x \in \mathbb{R}^n$ is feasible if it fulfills all constraints in the problem; *i.e.*, $Ax_0 \leq b$.

**Definition 2** (Feasible set)**.** The feasible set $\mathcal{F}$ is the set of all feasible points:

$$\mathcal{F} \coloneqq \{x \in \mathbb{R}^n \mid Ax \leq b\}$$

**Definition 3** (Feasible problem)**.** We say that an optimization problem is feasible if its feasible set is not empty.

**Definition 4** (Local minimizer)**.** We say that a point $x^* \in \mathcal{F}$ is a local minimizer of (2.11) if there is an $\varepsilon \in \mathbb{R}$, $\varepsilon > 0$, such that $\mathrm{V}(x^*) \leq \mathrm{V}(x)$ for all $x \in \mathcal{F}$ that satisfy $\|x - x^*\| < \varepsilon$.

**Definition 5** (Global minimizer)**.** We say that a point $x^* \in \mathcal{F}$ is a global minimizer of (2.11) if $\mathrm{V}(x^*) \leq \mathrm{V}(x)$ for all $x \in \mathcal{F}$.

**Definition 6** (Active constraints)**.** Let $x_0 \in \mathcal{F}$ be a feasible point. Then the sets of active and inactive constraints at $x_0$ are:

$$\mathcal{A}(x_0) = \{i \in \mathcal{I} \mid A_i x_0 = b_i\}$$
$$\mathcal{NA}(x_0) = \{i \in \mathcal{I} \mid A_i x_0 < b_i\}$$

Where $A_i$ is the $i$-th row in $A$, and $b_i$ is the $i$-th element in $b$. In addition, let $A_{\mathcal{A}(x_0)}$ and $b_{\mathcal{A}(x_0)}$ be the submatrices of $A$ and $b$ corresponding to the rows indexed by $\mathcal{A}(x_0)$.

**Definition 7** (Regularity)**.** We say that a point $x_0 \in \mathcal{F}$ is regular to (2.11) if $A_{\mathcal{A}(x_0)}$ has full row rank; that is, if all the active constraints at $x_0$ are linearly independent.

In addition, we are interested in *convex* optimization problems.

**Definition 8** (Convex QP)**.** We say that a QP in the form (2.11) is convex if the cost function $\mathrm{V}(x)$ and the feasible set $\mathcal{F}$ are convex.

For out purposes, it is sufficient to know that a twicely differentiable function $\mathrm{f}(x)$ is convex on a domain $\mathcal{D} \subseteq \mathbb{R}^n$ if the Hessian matrix $\nabla^2 \mathrm{f}(x)$ is positive semidefinite for all $x \in \mathcal{D}$. Similarly, it is sufficient to know that any set defined by affine inequality constraints, as in the feasible set of (2.11), is a convex set. We are interested in convexity because of the following:

**Theorem 2.2.1** (Convexity gives global optimality)**.** *Assume that* $\mathrm{V}(x)$ *is a convex function on* $\mathcal{F}$*. If* $x^*$ *is a local minimizer to* (2.11)*, then it is also a global minimizer to* (2.11)*.*

The theorem is presented here without proof. For more general definitions of convexity and a proof of the theorem, the reader is refered to [12].

Finally, we can say more if we require $H \succ 0$:

**Theorem 2.2.2** (Uniqueness of primal solution)**.** *Let* $H$ *be a strictly positive definite matrix. Then* $\mathrm{V}(x)$ *is a strictly convex function and* (2.11) *cannot have multiple optima nor be unbounded. If the feasible set* $\mathcal{F}$ *is not empty, the solution is unique.*

A similar result holds for the dual solution $\lambda^*$ if the primal optimizer $x^*$ is a regular point:

**Theorem 2.2.3** (Uniqueness of dual solution)**.** *If the primal optimizer* $x^*$ *of* (2.11) *is a regular point, the dual solution* $\lambda^*$ *is unique.*

The interested reader can find the proof for Theorem 2.2.2 and Theorem 2.2.3 in [10, Chapter 3].

Finally, we present the First Order Necessary Optimality Conditions for convex QP problems, also called Karush-Kuhn-Tucker (KKT) conditions. Because of convexity, the KKT conditions are also sufficient conditions of optimality for problems in the form (2.11) (see [12]):

**Definition 9** (KKT conditions)**.** The KKT conditions for (2.11) are

$$Hx^* + A^\top \lambda^* + h = 0, \tag{2.13}$$

$$Ax^* \leq b, \tag{2.14}$$

$$\lambda_i \geq 0, \forall i = 1, \ldots, m, \tag{2.15}$$

$$\lambda_i(A_i x^* - b_i) = 0, \forall i = 1, \ldots, m. \tag{2.16}$$

For some $\lambda \in \mathbb{R}^m$. We say that $\lambda$ is the vector of *dual variables* or *Lagrage multipliers*.

The four conditions listed above receive particular names. (2.13) is the *stationarity* condition. (2.14) and (2.15) are, respectively, the primal and dual *feasibility* conditions. Finally, (2.16) is referred to as the *complementary slackness* condition. Note that the complementary slackness is

the only nonlinear equation in (2.13) - (2.16), which will be important in following chapters.

For completeness, the KKT conditions for a problem in the form (2.12) are presented below. They will be needed later, because in the chosen interior point solver, the equality constraints are not removed from the problem formulation to preserve sparsity:

**Definition 10** (KKT conditions)**.** The KKT conditions for (2.12) are:

$$\begin{aligned} Hx^* + A^\top \lambda^* + C^\top \nu^* + h &= 0, \\ Ax^* &\leq b, \\ Cx^* &= d, \\ \lambda_i &\geq 0, \, \forall i = 1, \ldots, m, \\ \lambda_i (A_i x^* - b_i) &= 0, \forall i = 1, \ldots, m. \end{aligned}$$

For some $\lambda \in \mathbb{R}^m$ and some $\nu \in \mathbb{R}^p$. Here, $\nu$ are the Lagrange multipliers associated with the equality constraints.

### 2.2.2 QP Formulation of a Linear MPC

Consider a general linear time-invariant MPC in form (2.6). It is an optimization problem with a quadratic cost function and affine constraints; as such, we can formulate it as a convex QP.

Note that both equality and inequality constraints are present in (2.6). The first correspond to the system dynamics, and the second to the problem constraints on inputs and states. Because of both types of constraints are present, we can choose between two QP formulations: explicitely handling the equality constraints as in (2.12), or eliminating the equality constarints and solving a problem in form (2.11). In the MPC context, both formulations have advantages and disadvantages. If the equality constraints are removed from the QP, we speak of a *dense* QP formulation. Otherwise, we speak of a *sparse* formulation.

Let $K$ denote the total number of constraints in (2.6); that is, $K \coloneqq T\kappa_u + (T-1)\kappa_x + \kappa_t$. Then, the sparse formulation of (2.6) is a QP over $T(m+n)$ variables with $K$ constraints. In constrast, the dense formulation results in a much smaller QP with just $Tm$ variables (but with the same number of constraints). In turn, the sparse formulation produces sparse QP matrices, that are easier to manipulate with the sparse matrix algebra.

In the sparse formulation we group all inputs and states in a sole vector $z \coloneqq \begin{bmatrix} u_0^\top & x_1^\top & u_1^\top & \dots & u_{T-1}^\top & x_T^\top \end{bmatrix}^\top$ of dimension $T(n+m)$. The sparse formulation of (2.6) is

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & z^\top H z \\ \text{subject to} \quad & Fz \leq f, \\ & Cz = d, \end{aligned} \tag{2.17}$$

where

$$H = \begin{bmatrix} R & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & Q & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & R & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & Q & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & R & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & P \end{bmatrix} \in \mathbb{R}^{T(n+m) \times T(n+m)}$$

$$F = \begin{bmatrix} F_u & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & F_x & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & F_u & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & F_x & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & F_u & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & F_t \end{bmatrix} \in \mathbb{R}^{K \times T(n+m)}$$

$$C = \begin{bmatrix} B & -I & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & A & B & -I & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & A & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \\ 0 & 0 & 0 & 0 & \dots & -I & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & A & B & -I \end{bmatrix} \in \mathbb{R}^{(T \cdot n) \times T(n+m)}$$

$$f = \begin{bmatrix} f_u^\top & f_x^\top & \dots & f_u^\top & f_t^\top \end{bmatrix}^\top \in \mathbb{R}^K$$

$$d = \begin{bmatrix} -x_0^\top C^\top & 0 & \dots & 0 & 0 \end{bmatrix}^\top \in \mathbb{R}^{T \cdot n}$$

The so-called controllability matrix [10, Chapter 11] is a basis for the null space of $C$ in the formulation above. Because the controllability matrix is a linear map from the inputs to the future states, the null space of $C$ is formed by the input sequence $U$. In the dense formulation of

(2.6),the state variables $x$ are eliminated, resulting in an optimization problem over $u$:

$$\begin{aligned} \underset{U}{\text{minimize}} \quad & U^\top H' U + h'(x_0)^\top U \\ \text{subject to} \quad & F' U \leq f'. \end{aligned} \tag{2.18}$$

Where $U \in \mathbb{R}^{T \cdot m}$, $H' \in \mathbb{R}^{Tm \times Tm}$, $F' \in \mathbb{R}^{K \times Tm}$, $h' \in \mathbb{R}^{Tm}$ and $f' \in \mathbb{R}^K$. The reader is referred to [13] for a detailed description of them.

### 2.2.3 Properties of the arising QP

Convex QPs have important properties that numerical solvers can exploit to find the minimizer. We are interested in the conditions under which a Linear MPC produces a convex QP. More in particular, we would like to know what should we require from (2.6) so that Theorems 2.2.1, 2.2.2 and 2.2.3 hold.

First, we would like that the arising QP is strictly convex, so that the KKT conditions are sufficient optimality conditions and the minimizer is unique. For $H \succ 0$, it suffices that $R \succ 0$ and $Q, P \succeq 0$. Some algorithms require that $Q$ and $P$ are strictly positive definite too, but this is not necessary for Theorems 2.2.1 and 2.2.2 to apply. More in particular, the chosen algorithm does require $Q, P \succ 0$.

Second, we want the active constraints to be linearly independent at the minimizer, such that $\lambda^*$ is unique. In general, if two (affine) active inequality constraints are linearly dependant at some point, it means that one of them is redundant. In that case, the redundant constraint can be removed without modifying the problem [10, Chapter 5], and every point becomes a regular point.

From now on, we will assume that the QP that we are solving is convex and that $H$ is strictly positive definite. Moreover, we will assume that any redundant inequality constraint has been removed and that the inequality constraints constitute a minimal representation of the feasible set.

## 2.3 Embedded Optimization

Optimization problems arise in many different contexts, each of them posing different requirements on the software. In [14], the author lists the main requirements on embedded optimization code for MPC:

1. **Throughput:** The code shall be fast enough to give an answer to the optimization problem within the next sampling interval.
2. **Simple hardware:** The code shall be able to run on simple hardware, such as microprocessors, microcontrollers, FPGAs, etc.
3. **Little memory:** There is limited memory for data and code.
4. **Worst-case execution time:** The execution time in the worst-case scenario shall be tightly estimated.
5. **Simplicity:** The code shall be simple enough to be *verifiable* or *certifiable*.

In plain English, the code shall deliver a good and fast solution (throughput) with certain reliability (worst-case execution time, simplicity), subject to hardware limitations (simple hardware, little memory).

To deliver a software with those charcateristics, there are important decisions to make. Of particular importance is the choice of the optimization algorithm, which is the topic of Chapters 3 and 4. Also in [14], the author translates the previous demands on the optimization code, to more specific requirements on the optimization algorithm:

1. **Degrees of freedom:** The algorithm shall allow to trade between throughput, memory and robustness.
2. **Numerical robustness:** The algorithm shall perform correctly in the target precision arithmetic (*e.g.*, single precision, but even fixed-point).
3. **Simple linear algebra:** The algorithm shall be limited to basic linear algebra operations (*e.g.*, at most matrix-vector products). This translates into a simpler code.
4. **Guaranteed feasibility:** Feasibility is more important than optimality, and hence the algorithm shall ensure that a feasible solution is always provided (*e.g.*, in case of early termination).
5. **Worst-case iteration count:** The maximum number required by the algorithm shall be upper bounded, either theoretically (*e.g.*, gradient projection methods) or experimentally (*e.g.*, active set methods).

6. **Robustness against ill-conditioned data:** MPC problems are usually badly scaled (*e.g.*, inbalanced penalties between control goals, very high penalties for soft constraints, etc). Hence, the algorithm shall be robust against ill-conditioned problems (or have good automatic scaling methods).

The trade-off between throughput, memory requirements and robustness is desirable, such that the same algorithm can be used in different scenarios.

The numerical robustness is needed, since otherwise round-off errors may accumulate and the performance of the solver may degradate, in which case the controller may fail at its task. Also, numerical overflow can occur, with potentially disastrous consequences for the controlled system. However, numerical robustness in floating- or fixed-point arithmetics is difficult to assess theoretically. A good example is [15], where the author derives upper and lower bounds in the number of bits that are needed to attain a certain accuracy in a gradient projection method (see Chapter 3).

Limiting the linear algebra to basic operations is a practical limitation of given embedded hardware. Whereas operations with linear ($\mathcal{O}(n)$) and quadratic complexity ($\mathcal{O}(n^2)$) in the problem dimension $n$ are efficiently implemented, more complex routines are limited to regular computers. In addition, simpler linear algebra results in a simpler code, which is easier to mantain, develop and analyze (*e.g.*, as the authors do in the aforementioned work [15]).

Feasibility is a major concern for the algorithms under consideration. In a real-time applications, the controller must always provide a control input, even if it is suboptimal. This concept has sometimes been called anytime optimization: "*Given a feasible solution, keep improving its optimality as long as CPU is available, otherwise stop and apply best solution so far*" [14]. In other words: a feasible solution shall always be immediately available, even if the time needed to solve the QP exceeds the sampling interval of the optimization task is preempted. A more strict condition is that any feasible solution shall ensure stability, but this requirement is harder to guarantee theoretically.

The maximum iteration count is a desirable theoretical property, if it is available. If we do not have a tight bound on the iteration count, which is often the case, we must rely on experimental measurements. A maximum (or expected) iteration count is very useful, since it allows for a tight estimate on the worst-case solution time, which is crucial for

real-time applications.

Finally, the Hessian in (2.11) is typically ill-conditioned in MPC problems. Moreover, the constraints matrix in (2.11) may also be problematic in active set methods. Algorithms that rapidly deteriorate with high condition numbers are not desirable, since they will not be able to handle typical MPC problems.

# Chapter 3

# State of the Art

In this chapter, we review the state-of-the-art algorithms for solving quadratic programs in the MPC framework. *Offline* MPC constitutes a category on its own and we review it first. *Online* methods, contrarily, is a very generic term that covers many different algorithms, which may be classified according to diverse criteria. For example, QPs can be solved in the primal or in the dual space (or in both at the same time). Also, algorithms may be best suited for dense or sparse MPC formulations. Another important characteristic is whether the MPC formulation has full linear constraints or is limited to box constraints in the form $lb \leq u_k \leq ub$.

In the following, we will classify the algorithms according to the *order* of the information they use. In that sense, *first order* methods use the information from first order derivatives, whereas *second order* mathods make use of second derivatives.

## 3.1 Explicit MPC

It is well known that the optimizer $U^*$ for problem (2.6) is a picewise affine on polyhedra function of the initial state [10]. The explicit MPC approach has two phases: an offline and an online phase.

In the offline phase, the feasible set is divided into regions and the picewise affine control law is calculated via multi parametric programming as in [10, Chapter 12]. This step involves the main computational burden, but it can be moved offline.

Then, since the optimal control law is explicitely available, the online phase of the controller simplifies to allocating the initial state to the

correct region and reading the control law from a look-up table.

The main advantage of explicit MPC is that the optimization problem is parametrized with respect to $x_0$ and solved offline. However, this method has limited application in practice, because the number of regions where $x_0$ can be allocated grows exponentially with the number of constraints [13]. This results in rapidly increasing memory requirements (to store the explicit control laws for each region) and also required time to correctly allocate the initial state in the look-up table, which make this approach infeasible for medium to large size problems.

## 3.2 First Order Methods

First order methods obtain the search direction in every iteration using gradient information, but need not to compute second derivatives. This has a few consequences that are common to all first order methods. On the one hand, every iteration is inexpensive to compute, since the second order information is disregarded. On the other hand, the search directions at every iterate are worse than in second order methods, which results in a converge rate which is at most superlinear.

Despite slower convergence rates, first order methods have proven to be competitive in practice and there are examples of succesful applications of them to MPC problems. Finally, it is also worth noting that these methods are very sensitive to the condition number of the Hessian matrix $H$, and because of this, pre-conditioning is usually a common feature of all algorithms.

### 3.2.1 Gradient Projection Method

The Gradient Projection Method (GPM) exists both in primal and dual versions and is the simplest first order method. It is a generalization of the steepest descent method for unconstrained optimization: the gradient at every iterate is used as the search direction, which is later projected onto the feasible set if a constraint is hit [13]. The $k$-th iteration of the GPM is

$$z^{k+1} = \mathrm{P}_{\mathcal{F}}\big(z^k - \alpha^k \nabla(\mathrm{V}(z))\big) \tag{3.1}$$

Where $\mathrm{P}_{\mathcal{F}}(\cdot)$ is the projection operator onto $\mathcal{F}$, $\nabla(\mathrm{V}(z)) \coloneqq Hz$ is the gradient of the cont function at $z$, and $\alpha^k$ is the step size at iterate $k$.

The most expensive operation in (3.1) is the projection operator $P_{\mathcal{F}}$. It can be calculated efficiently for simple constraints, such as sphere or box constraints, but otherwise the projection operator involves solving a QP on its own. For this reason, primal GPM is limited to MPC problems where the only constraints are box constraints. Due to this limitation, GPM is not common in practical applications. Nonetheless, it is always possible to solve the problem in dual space, since the feasible region of the dual variables is the positive orthant and the projection is thus inexpensive to compute.

Another disadvantage of GPM is that it shows a linear convergence rate, since it becomes a version of steepest descent after the optimal basis has been identified [13]. There have been attempts to overcome this difficulty by adding second-order improvement steps to the pure first-order methods, see [16].

### 3.2.2 Fast Gradient Projection Method

Fast Gradient Projection Methods (FGM) are essentially the same as GPM, and, as such, they share most characteristics (inexpensive iterations, limitation to simple constraints or dual space and high sensitivity to Hessian condition number). However, FGM achieve a superlinear convergence rate by accumulating momentum: the search direction is not the gradient at the current iterate, but a combination of it and the accumulated gradient at the last iterate. Intuitively, accumulating gradient information is a means of using second order information without explicitely computing second order derivatives.

An additional advantage of FGM is that it has been object of extensive research in the MPC community. For example, tight bounds on the maximum number of iterations are available [17], which is advantageous for embedded MPC applications. Even a fixed-point implementation has been developed that prevents overflow and limits round-off error propagation by adequately choosing the number of integer and fractional bits [15]. Both examples are dual FGM algorithms.

## 3.3 Second Order Methods

As opposed to first order methods, each iteration of second order method is expensive to compute, but in turn fewer iterations are needed. There are two families of second order methods: active set methods,

whose iterates move along the boundary of the feasible set, and interior point methods, whose iterates lie in the strict interior of the feasible region. A common advantage to all second order methods is that they are rather insensitive to the Hessian condition number.

### 3.3.1 Active Set Methods

Active set methods (ASM) are an enhancement of the Simplex method for Linear Programming and, as such, they attempt to identify the optimal active set in a finite number of iterations. Starting from an arbitrary *working set* of inequality constraints, a sequence of equality constrained QP subproblems is solved. In every QP, the constraints in the working set are treated as equality constraints and the rest are disregarded. The result of every subproblem determines whether a constraints should be added to or removed from the working set. This update procedure continues until the optimal basis for the problem is identified.

Active set methods exist both in primal, dual and primal-dual versions. Furthermore, they normally favour the dense QP formulation, since they cannot really take advantage of sparsity in the matrices due to the required factorizations and associated fill-in phenomena.

In all versions, ASM show the following common characteristics. In the first place, there is no tight bound on the maximum number of iterations that ASM require to converge to the minimizer. The maximum iteration count grows exponentially with the problem size, and examples can be built where the ASM has to try all possible constraint combinations before finding the optimal one. However, ASM have proven to require a much lower number of iterations for convergence in practice and excel at a feature that other methods cannot effectively benefit from: *warm start*.

We speak of warm start when a good initial guess of the optimal active set is available. In such case, few iterations will be needed to converge, whereas a bad initial guess can lead to poor performance of the algorithm. In the MPC framework, ASM is best-suited to steady-state operation, where few changes in the control actions are needed. Attempts have been made to accelerate the identification of the optimal active set by adding or removing several constraints in every iteration. One of the most notable examples is qpOases, a multi-parammetric active set method [18]. Other approach is the combination of first-order

methods for fast active set identification and active-set methods for fast, quadratic convergence rates [13].

In the second place, every iteration of the ASM is expensive to compute when compared to first order methods. This characteristic is common to all second order methods. As opposed to first order methods, in which the calculation of the search direction involves at most matrix-vector operations, second order methods requires the solution of an indefinite linear system of equations. This system of linear equations is sometimes referred to as *KKT system* and can be very large depending on the problem dimensions, which makes efficient linear algebra of great importance. Different methods have been developed for solving the KKT system in the MPC framework, which we review in Chapter 4.

Finally, there are examples of dual ASM as well. Primal ASM require a feasible initial point which shall be calculated in a so-called *initial phase* of the algorithm. In contrast, Non Feasible ASM exploits the fact that a feasible dual point is always inmediately available, thus avoiding this inconvinience [19].

### 3.3.2 Interior Point Methods

Interior point methods (IPM) receive their name because they do not explore the bundaries of the feasible set, nor do they attempt to identify the optimal active set. Instead, IPM attempt to directly solve the KKT conditions (2.13) - (2.16), which have an unique solution despite being nonlinear (Theorems 2.2.1, 2.2.2 and 2.2.3).

Newton's method is used to solve the system of nonlinear equations. That is, a sequence of linearized KKT systems is solved, each of them closer to the final solution. However, a typical problem of IPM is that the KKT conditions become highly nonlinear near the boundary of the feasible set, resulting in a poor, ill-conditioned linearization if the initial iterate is not close to the minimizer.

To overcome this difficulty, the *central path* is introduced. The central path is the set of suboptimal solutions for perturbed KKT systems, where (2.16) is substituted by:

$$\lambda_i(Ax_i - b_i) = t_k, \quad i = 1, \ldots, m \tag{3.2}$$

Where $t$ is a strictly positive scalar. This perturbation can be explained in terms of a logarithmic barrier method, and the interested reader

is referred to [20, Chapter 16]. Here it is sufficient to know that the minimizer to the optimization problem lies in the interior of the feasible region for any $t > 0$. The approach of all IPM is to solve a sequence of linearized KKT systems where the parameter $t$ is progressively reduced. This way, the optimal solution is approached from the interior of the feasible region, avoiding the ill-conditioning that occurs near the boundaries.

IPM do not find the exact solution for the optimization problem, as it lies in the limit $t \to 0$. Nonetheless, we can get a solution as accurate as desired by continuously decreasing $t$. In general, different methods differ in their strategy to perturb the KKT condition and decrease the parameter $t$. The most succesful IPM are primal-barrier and primal-dual methods. Each iteration of a primal-dual methods is generally more expensive to compute, since the dual variables are incorporated into the problem. In turn, such methods are better conditioned than primal-barrier ones.

As in ASMs, not only the algorithm, but also the solution of every linearized KKT system is a concern for IPMs. This step is more crucial here, because IPMs take all constraints into account at the same time, thus resulting in KKT systems typically larger than those in ASM. Though every IPM iterate is computationally expensive when compared to ASM, the iteration count is normally low and rather constant, in a range of at most 75 to 100 iterations for large problems.

One of the main advantages of IPM is that they are well suited for exploiting the sparsity and the structure of the matrices in the problem formulation. We have already seen that the sparse MPC formulation produces very sparse matrices. In fact, the matrices are either block diagonal ($H, F$) or block bi-diagonal ($C$). The MPC community has put great emphasis into preserving and exploiting the sparsity patterns in MPC problems, with notorious reduction in the computational complexity as in [21], [22].

# Chapter 4

# Proposed Approach

There are many different algorithms suitable for solving the optimization problems associated with MPC. In this chapter, we will present the algorithm that we have chosen for our implementation, along with the reasons that motivate this choice. The chosen algorithm is the *Mehrotra Predictor-Corrector Primal-Dual Interior Point Method* (from now on, the Mehrotra algorithm).

First of all, it is an interior point solver. Hence, we know that the effectivity of this method relies in a small and rather constant number of iterations. We also know that these iterations are computationally expensive when compared to other methods.

The reason for the choice of an IPM is that they excel at exploiting the structure and the sparsity of QPs arising from MPC problems, although IPMs are not the only choice for sparse optimization problems. For example, conjugate gradient (CG) methods may use an *incomplete Cholesky factorization* as a preconditioner [7, Chapters 5, 6] when solving linear systems of equations. This prevents the fill-in of pure Cholesky factorizations, and can be applied to the sparse linear systems that arise in active set methods. However, this approach is "blind" regarding the structure of the MPC problem. The CG method for sparse linear system neglects any structure in those, since the matrices are treated as generic sparse matrices. On the other hand, IPMs can actively exploit not only the sparsity of MPC problems, but also their structure, as it has been shown in [21] and [22]. For this reason, we believe that IPMs adapt best to MPC-type optimization problems.

Second, among the family of IPM, the Mehrotra algorithm is a Primal-Dual version. We shall note that pure primal versions have been

reported in the literature, as in [21]. Primal methods (also logarithmic-barrier methods) only use information related to primal variables. Hence, they avoid the computational cost of evaluating the dual variables of the problem at each step, which results in cheaper iterations. In Primal-Dual methods, we are interested in computing these variables, and make use of the information they provide. This results, in general, in better convergence properties, at the cost of a more complicated algorithm. A profuse analysis on the similarities and differences between Primal and Primal-Dual methods can be found in [12, Chapter 11, Section 7].

Finally, we have chosen the Mehrotra algorithm for its Predictor-Corrector characteristic. Most Primal-Dual IPM, as they are presented in the literature ([7], [12], [20]), rely on heuristics for the choice of the initial barrier parameter $t_0$ in 3.2. Also the update strategy for this parameter follows some heuristic rule, typically in the form $t_k = \rho \cdot t_{k-1}$ for some $\rho < 1$. We shall note that the choice of the pair $\{t_0, \rho\}$ can have a great influence in the performance of the algorithm. Whereas a good choice of them will result in a successful, efficient algorithm, a poor choice of them can result in slow or even non-convergent algorithms. (See a discussion about this topic in [12, Chapter 11].) The Predictor-Corrector method, on the other hand, introduces the concept of *adaptative choice* of $t_k$ depending on the available primal and dual information.

Additionally, the Mehrotra algorithm introduces a *corrector* term that takes into account a second order approximation of the KKT conditions, as opposed to the pure linearization that is common for all second order methods. Both enhancements of the basic Primal-Dual algorithms come at the cost of additional computational effort, but produce better results in practice.

## 4.1 The Mehrotra Predictor-Corrector Primal-Dual Interior Point Algorithm

Since the IPM can exploit sparsity, the sparse MPC formulation (2.17) is a natural choice. The QP under consideration is reproduced here for

convinience:

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & z^\top H z + z^\top h \\ \text{subject to} \quad & Fz \leq f, \\ & Cz = d, \end{aligned}$$

Where $z \in \mathbb{R}^{T(n+m)}$ and matrices $H$, $F$ and $C$ are sparse.

In the following, we will introduce slack variables to replace the inequality constraints by equality constraints. This results in a neater formulation of the algorithm. The introduction of slack variables $s$ produces the following QP:

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & z^\top H z + z^\top h \\ \text{subject to} \quad & Fz + s = f, \\ & s_i \geq 0, \quad \forall i \\ & Cz = d, \end{aligned} \tag{4.1}$$

If $K$ denotes the total number of inequality constraints (along the whole prediction horizon), we have $s \in \mathbb{R}^K$. The KKT conditions for (4.1) are:

$$Hz^* + F^\top \lambda^* + C^\top \nu^* + h = 0, \tag{4.2}$$
$$Fz^* + s^* = f, \tag{4.3}$$
$$Cz^* = d, \tag{4.4}$$
$$\lambda_i s_i = 0, \quad i = 1, \ldots, K, \tag{4.5}$$
$$\lambda_i \geq 0, \quad i = 1, \ldots, K, \tag{4.6}$$
$$s_i \geq 0, \quad i = 1, \ldots, K, \tag{4.7}$$

where $\lambda \in \mathbb{R}^K$ and $\nu \in \mathbb{R}^{T \cdot n}$ are the vectors of Lagrange multipliers associated with the inequality and equality constraints, respectively. As we explained in Chapter 3, IPMs attempt to directly solve the KKT conditions (4.2) - (4.7). Since they are nonlinear due to the complementary slackness condition (4.5), we take a Newton approach for solving the system.

The Newton Method is a numerical, iterative method for solving nonlinear equations. At every step, we obtain a search direction by solving a linearized approximation of the original system. Under certain conditions, the iterates sequence gets closer and closer to the non-linear solution and eventually converges:

**Definition 11** (Newton's Method)**.** Let the function $\mathrm{f}\colon n \mapsto m$ define the nonlinear equation $\mathrm{f}(x) = 0$. Starting at an initial point $x_0 \in \mathbb{R}^n$, every iterate of the Newton's Method is calculated as:

$$\nabla \mathrm{f}(x_k)^\top \Delta x_k = \mathrm{f}(x_k),$$
$$x_{k+1} = x_k + \Delta x_k,$$

where $\nabla \mathrm{f}(x_k)$ is the transpose of the function Jacobian.

In the Mehrotra algorithm, equations (4.2) - (4.5) are linearized and equations (4.6) - (4.7) are satisfied implicitely. We represent the linearized system by the so-called KKT matrix:

$$\begin{pmatrix} H & C\top & F\top & 0 \\ C & 0 & 0 & 0 \\ F & 0 & 0 & I \\ 0 & 0 & S & \Lambda \end{pmatrix} \begin{pmatrix} \Delta z \\ \Delta \nu \\ \Delta \lambda \\ \Delta s \end{pmatrix} = - \begin{pmatrix} r_c \\ r_e \\ r_i \\ r_s \end{pmatrix}, \tag{4.8}$$

where $S = \mathrm{diag}(s)$ and $\Lambda = \mathrm{diag}(\lambda)$ are diagonal matrices, with the diagonal entries containing the vectors $s$ and $\lambda$ at the current iterate. The residuals on the right hand side are given by:

$$r_c = Hz + F^\top \lambda + C^\top \nu + h,$$
$$r_e = Cz - d,$$
$$r_i = Fz + s - f,$$
$$r_s = S\Lambda e,$$

where $e$ is the vector of ones of appropriate dimension.

Note that the system (4.8) is very sparse, but it is also of large dimension. In particular, the KKT matrix is a square matrix of dimension $T^3(2n + m) + 2K$. In addition, note that $K$ depends on the prediction horizon in the form $K = T\kappa_u + (T - 1)\kappa_x + \kappa_f$, where $\kappa_u$ and $\kappa_x$ are the number of input and state constraints per prediction step, and $\kappa_t$ is the number of terminal constraints. Even if the dynamic system under consideration is small, the resulting KKT matrix can grow very fast in size. Because of this, special methods shall be employed to solve systems in the form (4.8). By now, we will assume that (4.8) is solved by some means and the vector of incremental variables is available.

Every step of the Mehrotra algorithm is composed of three search directions. The first one is the so-called *affine search direction*, which is obtained as the solution to the system (4.8), and is denoted by the

superindex $^{\text{aff}}$: $\Delta z^{\text{aff}}$, $\Delta \nu^{\text{aff}}$, $\Delta \lambda^{\text{aff}}$ and $\Delta s^{\text{aff}}$. The affine search direction corresponds to a pure Newton step towards the minimizer. However, recall that the KKT conditions become highly nonlinear close to the boundaries of the feasible set. As a result of this, the affine search direction becomes distorted if the iterates approximate the bounds, and the algorithm rapidly degradets its performance.

In practice, no interior point method implements a pure Newton search direction. Rather than this, they introduce a *centering search direction*, with the aim of biasing the iterates towards the central path. This produces a search direction of lesser-quality (since it is a distortion of the linearization), but the iterates stay central, so that future search directions will not be ill-formed due to approaching the boundary.

There are different techniques for including the centering component in the search direction. In the Mehrotra algorithm, the centering search direction is calculated as the solution to:

$$\begin{pmatrix} H & C\top & F\top & 0 \\ C & 0 & 0 & 0 \\ F & 0 & 0 & I \\ 0 & 0 & S & \Lambda \end{pmatrix} \begin{pmatrix} \Delta z^{\text{cent}} \\ \Delta \nu^{\text{cent}} \\ \Delta \lambda^{\text{cent}} \\ \Delta s^{\text{cent}} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ -t_k e \end{pmatrix},$$

where $t_k \in \mathbb{R}$ is the usual barrier parameter in all IPMs. Most algorithms use a pair $\{\rho, t_0\}$ as a heuristics for updating the barrier parameter:

$$t_k = \rho^k t_0,$$

for some $\rho < 1$ and $t_0 > 0$. More sofisticated methods use the *duality gap* as a measure of the progress made towards the solution, and choose the barrier parameter accordingly.

The duality gap is defined as:

**Definition 12** (Duality gap)**.** The duality gap in QP problems is the scalar given by:

$$\mu = \left(\lambda^\top s\right)/K. \tag{4.9}$$

In optimization, the duality gap is generally defined as the difference between the optimal values of a primal problem and its associated dual problem. However, it simplifies to (4.9) in QP problems. Furthermore, we know that the duality gap is exactly zero at the solution if the feasible set has a non-empty interior (due to Slater's condition, see [12]). In math: $\mu^* = (\lambda^*)^\top s^*/K = 0$.

Using the duality gap in the calculation of the centering search direction is superior to the update rule $\{t_0, \rho\}$. The barrier parameter is chosen as

$$t_k = \sigma\mu_k,$$

where $\mu_k$ is the duality gap at iterate $k$ and $\sigma \in [0, 1]$ is the so-called centering parameter. The choice $\sigma = 0$ corresponds to the pure Newton method and is disencouraged for the reasons discussed above. On the other hand, $\sigma = 1$ forces the iterates to stick to the central path, but no progress is made towards the optimizer, as it discussed in [23] and [7].

There is no obvious choice of $\sigma$, which is a tuning parameter of the algorithm. The scalar $\sigma$ weights the trade-off between fast convergence and centrality. Low values of $\sigma$ are used to rapidly converge to the minimizer, but may result in poor results if the iterates do not stay central. On the other hand, high values of $\sigma$ ensure that the iterates stay central and that the algorithm will eventually converge, though slower. It is difficult to say *a priori* which choice of $\sigma$ is appropriate. If the problem is well-conditioned and the iterates naturally stay central to the feasible region, low values of $\sigma$ guarantee the quadratic convergence rate of the Newton's method. On the other hand, large values of $\sigma$ are necessary if the problem is ill-conditioned and the central path approaches the boundaries [23].

To overcome this blind decision making, Mehrotra proposes an adaptative choice of the centering parameter. After evaluating affine step, we perform a *linesearch* that allows us to test the quality of the pure Newton step. The linesearch states what is the maximum step size (in the affine search direction) such that the next iterate is primal and dual feasible with respect to the inequality constraints. This corresponds to equations (4.6) and (4.7), and justifies why they can be excluded from the KKT system (4.8). It is worth noting that, as opposed to other QP algorithms (*e.g.*, primal-logarithmic interior point methods, [21]), *exact* linesearch can be performed (that is, back tracking is not needed). The affine step size is calculated as:

$$\alpha^{\text{aff}} = \operatorname{argmin}\big(\alpha \in [0, 1] \mid \lambda + \alpha\Delta\lambda^{\text{aff}} \geq 0,\ s + \alpha\Delta s^{\text{aff}} \geq 0\big).$$

After calculating the maximum step size, the *affine duality gap* measures the quality of the serch direction as:

$$\mu^{\text{aff}} = (\lambda + \alpha\Delta\lambda^{\text{aff}})^\top (s + \alpha\Delta s^{\text{aff}})/K.$$

We say that the affine step makes great progress towards the solution if $\mu^{\text{aff}}$ is small compare to $\mu$. In this case, we will benefit from a high-quality search direction by enforcing little centering. On the other hand, we say that the search direction is poor if $\mu^{\text{aff}}$ is similar to $\mu$. If this happens, we will impose a large centering component, in an attempt to get closer to the central path, and hence produce a good Newton direction in the next iteration. Taking all of this into account, the centering paramenter for step $k$ is chosen as follows:

$$\sigma = \left( \frac{\mu_k^{\text{aff}}}{\mu_k} \right)^3 \tag{4.10}$$

Here, we have presented the heuristics proposed by Mehrotra in his orginial paper [24], which has proven successful in practice. It is worth noting that different heuristics have been tried, specially in the same form as (4.10) but with different exponents. However, none of them proved any improvement with respect to Mehrotra's original contribution (see [23, Chapter 10]).

The inclusion of (4.10) results in the following formulation of the centering step:

$$\begin{pmatrix} H & C\top & F\top & 0 \\ C & 0 & 0 & 0 \\ F & 0 & 0 & I \\ 0 & 0 & S & \Lambda \end{pmatrix} \begin{pmatrix} \Delta z^{\text{cent}} \\ \Delta \nu^{\text{cent}} \\ \Delta \lambda^{\text{cent}} \\ \Delta s^{\text{cent}} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ -\sigma \mu^{\text{aff}} e \end{pmatrix} \tag{4.11}$$

Note that, in the Mehrotra algorithm, the centering step can only be calculated after evaluating the affine step. This is not the case for other centering strategies (such as the aforementioned $t_k = \rho^k t_0$ or $t_k = \sigma\mu$). In those cases, the right hand side in (4.11) can be added with the right hand side in (4.8) and both the affine and centering directions are calculated simultaneously.

Finally, Mehrotra introduces a *correction term* that takes into account the linearization error produced in (4.8). This is arguably the major enhancement of this algorithm with respect to other interior point methods, as the author himself recognizes in [24]. Note that the linearization of (4.5) comes from the expansion:

$$(\lambda_i + \Delta\lambda_i)(s_i + \Delta s_i) = \lambda_i s_i + s_i \Delta\lambda_i + \lambda_i \Delta s_i + \Delta\lambda_i \Delta s_i, \quad \forall i,$$

where the last, nonlinear term is dropped.

However, because we know the exact value of the affine search direction, we can now approximate the nonlinear term $\Delta\lambda_i\Delta s_i$ in our formulation by $\Delta\lambda_i^{\text{aff}}\Delta s_i^{\text{aff}}$, thus reducing the linearization error. The quality of this approximation is discussed in [23]. The corrector step satisfies the following system of linear equations:

$$\begin{pmatrix} H & C\top & F\top & 0 \\ C & 0 & 0 & 0 \\ F & 0 & 0 & I \\ 0 & 0 & S & \Lambda \end{pmatrix} \begin{pmatrix} \Delta z^{\text{cor}} \\ \Delta\nu^{\text{cor}} \\ \Delta\lambda^{\text{cor}} \\ \Delta s^{\text{cor}} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ -\Delta\Lambda^{\text{aff}}\Delta S^{\text{aff}}e \end{pmatrix}. \tag{4.12}$$

The search direction at every iterate of the Mehrotra method corresponds to the sum of the affine, centering and correction components. This attempts to improve the quality of the search direction when compared to general IPMs, which commonly include no correction term and use an open-loop centering based on heuristics [21]. However, this also comes at the cost of solving three large linear systems, instead of one.

The key of Mehrotra algorithm is that all three linear systems (4.8), (4.11) and (4.12) have the same coefficients matrix. For this reason, the KKT matrix has to be factored just once, which is the most expensive operation when solving systems of linear equations. In fact, the cost of factoring a general $(n \times n)$ matrix is $\mathcal{O}(n^3)$, whereas the backward and forward substitution against a triangular (factored) matrix is just $\mathcal{O}(n^2)$. Furthermore, the centering and the correction terms are independent of each other and may be computed at the same time by adding the right hand sides in (4.12) and (4.11) and solving against it. In the following, this will be referred to as the *correction plus centering* step and denoted by the superscript $^{\text{cc}}$: $z^{\text{cc}}$, $\nu^{\text{cc}}$, $\lambda^{\text{cc}}$, and $s^{\text{cc}}$.

Due to these reasons, the enhancements of the Mehrotra algorithm come at a relatively low cost, whereas they produce a great improvement in the quality of the search directions. Whereas the affine search direction is the solution to the linearized KKT conditions, the addition of the corrector term produces a search direction that is the approximate solution to the second order Taylor approximation of the KKT system. In short, the corrector term takes into account the curvature of the central path when computing the search direction, rather than a pure first order approximation. The reader is referred to [23, Chapter 11] for a detailed and beautiful explanation of the corrector term and its consequences.

The complete Mehortra method is outlined in Algorithm 1, where all the variables are compacted in the vector $\xi = (z, \nu, \lambda, s)^\top$ for the sake of clarity.

**Data:** $H$, $h$, $F$, $f$, $C$, $D$ (as in (4.1))
**Result:** $\xi^* = (z^*, \nu^*, \lambda^*, s^*)^\top$
**Initialization:** $x_0$, $\xi_0 = (z_0, \nu_0, \lambda_0, s_0)^\top$, $k = 0$

**while** *not done* **do** `/* check optimality condition */`

Compute $\Delta\xi_k^{\text{aff}}$. `/* as in (4.8) */`
$\alpha^{\text{aff}} \leftarrow \text{linesearch}\left(\Delta\xi_k^{\text{aff}}\right)$ `/* affine linesearch */`
$\sigma \leftarrow \left(\mu_k^{\text{aff}}/\mu_k\right)^3$
Compute $\Delta\xi_k^{\text{cc}}$. `/* as in (4.12) and (4.11) */`
$\Delta\xi_k \leftarrow \Delta\xi_k^{\text{aff}} + \Delta\xi_k^{\text{cc}}$ `/* FULL search direction */`
$\alpha \leftarrow \eta \cdot \text{linesearch}(\Delta\xi_k)$ `/* linesearch */`
$\xi_{k+1} \leftarrow \xi_k + \alpha\Delta\xi_k$
$k \leftarrow k + 1$

**end**

**Algorithm 1:** Primal-Dual Predictor-Corrector Mehrotra Algorithm

Every step in Algorithm 1 has already been presented but for two items. First, there is a parameter $\eta \in (0, 1]$ scaling the step length once that the full search direction $\Delta\xi_k$ has been computed. $\eta$ prevents the pairs $(\lambda_i, s_i)$ from becoming too unbalanced, which can produce numerical instabilities in the KKT matrix. Different implementations of the Mehrotra algorithm use different values, ranging from the conservative $\eta = 0.9$ to a more aggresive $\eta = 0.9999$. Experience has shown that the choice $\eta < 1$ speeds up the asymptotic convergence and the robustness of the algorithm [23].

Second, we shall define a termination condition for the algorithm. The –unique– minimizer of the problem exactly fulfills the first order optimality conditions, and, as such, produces a zero residual in 4.8. Taking into account that the exact optimizer cannot be attained because of numerical inaccuracy, the most common stopping criterium is a maximum norm on the residual of the linearized KKT conditions.

However, it is not necessary to take all the residuals into account. It is usual to restrict the termination criterium to two of the residuals. First,

the equality residual $r_e$ should lay below some treshold to guarantee that the problem is feasible with respect to the equality constraints. Second, the duality gap is used as an optimality criterium: the current iterate is assumed to be optimal if it is feasible and the duality gap lies below some treshold. (Note that the duality gap is equal to the 1-norm of the complementary slacknes residual $r_s$).

## 4.2 Solving the KKT System in the MPC Frame

The most computationally expensive step in Algorithm 1 is the factorization of the KKT matrix. The complexity comes from the large size of it. If a general, dense factorization scheme is employed, the cost of factoring the matrix is $\mathcal{O}(T^3(2n+m+2K)^3)$. This is prohibitive even for medium size problems, given that we have to factorize the system at every iterate of the algorithm.

The success of interior point methods depends to a great extent on the ability to exploit the sparsity in the KKT matrix. This can be achieved using sparse algebra. Nonetheless, QPs arising from MPC problems have a very clear structure that can be used to further boost the factorization (and solution) of the linear system. The ideas in [21], further developed by [22], will be presented below. The KKT system (4.8) is reproduced here for convenience.

$$\begin{pmatrix} H & C\top & F\top & 0 \\ C & 0 & 0 & 0 \\ F & 0 & 0 & I \\ 0 & 0 & S & \Lambda \end{pmatrix} \begin{pmatrix} \Delta z \\ \Delta\nu \\ \Delta\lambda \\ \Delta s \end{pmatrix} = - \begin{pmatrix} r_c \\ r_e \\ r_i \\ r_s \end{pmatrix},$$

In first place, note that the matrices $S$ and $\Lambda$ are positive definite by construction. For this reason, the variable $\Delta s$ can be eliminated from the system using the fourth block equation:

$$\Delta s = -\Lambda^{-1}(r_s + S\Delta\lambda), \tag{4.13}$$

which results in a reduced system of three block equations:

$$\begin{bmatrix} H & C\top & F\top \\ C & 0 & 0 \\ F & 0 & -\Lambda^{-1}S \end{bmatrix} \cdot \begin{bmatrix} \Delta z \\ \Delta\nu \\ \Delta\lambda \end{bmatrix} = - \begin{bmatrix} r_c \\ r_e \\ r_i - \Lambda^{-1}r_s \end{bmatrix}$$

The product $\Lambda^{-1}S$ is again positive definite by construction. We can use it to eliminate $\Delta\lambda$ using the third block equation:

$$\Delta\lambda = S^{-1}\Lambda(F\Delta z + r_i - \Lambda^{-1}r_s) \tag{4.14}$$

The elimination results in a KKT system with Shur Complement structure:

$$\begin{bmatrix} H + F\top S^{-1}\Lambda F & C\top \\ C & 0 \end{bmatrix} \cdot \begin{bmatrix} \Delta z \\ \Delta\nu \end{bmatrix} = - \begin{bmatrix} r_c + F\top S^{-1}(\Lambda r_i - r_s) \\ r_e \end{bmatrix}$$

This system is often referred to as the *augmented KKT system* and is characteristic to all second order methods (including active set methods). In the primal-dual framework we rename the upper, left block to simplify notation. The residuals vector is also renamed, resulting in the canonical formulation:

$$\begin{bmatrix} \Phi & C\top \\ C & 0 \end{bmatrix} \cdot \begin{bmatrix} \Delta z \\ \Delta\nu \end{bmatrix} = - \begin{bmatrix} r_d \\ r_e \end{bmatrix} \tag{4.15}$$

Where we have introduced the contractions

$$\Phi \coloneqq H + F\top S^{-1}\Lambda F$$
$$r_d \coloneqq r_c + F\top S^{-1}(\Lambda r_i - r_s)$$

The *augmented KKT matrix* in (4.15) is a square matrix of dimension $T(2n + m)$. In addition, it has a very well known structure (the *Shur Complement* structure), which has been profusely studied in the literature. Three solving strategies are used in the MPC framework: direct solution, null-space methods and range-space methods.

## Direct Solution

The direct solution involves a triangular factorization of (4.15), followed by forward- and backwards substitution against the right hand side. For square matrices, there are two main factorization schemes: $LU$ factorization and $LDL^\top$ factorization. The latter is superior in performance and computational cost, but can only be applied to symmetric matrices.

If the matrix is not only symmetric, but it is also positive definite, we can employ a variation of the $LDL^\top$ factorization where $D$ is the

identity matrix. This particular case is known as the *Cholesky* factorization and is superior not only in terms of computational cost, but also in terms of numerical stability.

We know that the augmented KKT matrix is symmetric and indefinite [7, Chapter 16]. The best factorization scheme for it is the $LDL^\top$ factorization, which is the preferred choice for symmetric, indefinite matrices. Given a matrix $K$, the $LDL^\top$ factorization has the form:

$$K = P(LDL^\top)P^\top,$$

where $L$ is (lower) triangular, $D$ is diagonal and $P$ is a permutation matrix introduced for numerical stability and, ocasionally, to preserve sparsity. The $LDL^\top$ factorization complexity is cubic in the matrix dimension. In our case, the augmented KKT matrix has dimension $T(n+m)+Tn$ and the direct solution has complexity $\mathcal{O}(T^3(2n+m)^3)$, where $T$ is the prediction horizon, $n$ is the size of the system state and $m$ is the size of the system input.

## Null Space Methods

The equality constraints subdivide the variables into two subspaces. First, the *range space* of vectors that may be expressed as a linear combination of the rows in $C$ (*i.e.*, $\mathcal{R}(C^\top)$). And second, the *null space* of the vectors orthogonal to the rows of $C$ (*i.e.*, $\mathcal{N}(C)$). Since $C \in \mathbb{R}^{(Tn)\times(T(m+n))}$ and all its rows are linearly independent (*i.e.*, $\mathrm{rank}(C) = Tn$), we have $\mathcal{R}(C^\top) \subseteq \mathbb{R}^{Tn}$ and $\mathcal{N}(C) \subseteq \mathbb{R}^{Tm}$.

In null- and range-space methods, one of the subspaces is eliminated and the problem is solved in a smaller subspace. The null-space method eliminates the equation $C\Delta x = -r_e$ and solves the system in the subspace of the $T \cdot m$ free variables. The range-space methods does the contrary: $\Phi\Delta x + C\top\Delta\mu = -r_d$ is eliminated and the problem is solved in the subspace of the Lagrange multipliers ($T \cdot n$ variables).

In the null-space method, the direction $\Delta z$ is divided into two components, namely:

$$\Delta z = Y\Delta z_y + Z\Delta z_z$$

Where the columns of $Z \in \mathbb{R}^{T(n+m)\times T\cdot m}$ are a basis for $\mathcal{N}(C)$, and the columns of $Y \in \mathbb{R}^{T(n+m)\times T\cdot n}$ are a basis for $\mathcal{R}(C\top)$. The first term represents a particular solution for the equality constraints (must not be

calculated if a feasible point is available), and the second term represents a correction in the subspace of the free variables.

In MPC problems, it is natural to choose system inputs and states as free and dependent variables, respectively, albeit other choices are possible. By substituting the expansion of $\Delta z$ into the augmented KKT system and multiplying by $Z^\top$, one obtains:

$$Z^\top \Phi Y \Delta z_y + Z^\top \Phi Z \Delta z_z + Z^\top C^\top \Delta \mu = -Z^\top r_d$$

Using the fact $CZ = 0$ and rearranging terms:

$$(Z^\top \Phi Z)\Delta z_z = -Z^\top r_d - Z^\top \Phi Y \Delta z_y$$

The matrix $Z^\top \Phi Z$ is referred to as the *reduced Hessian* and is positive definite if the problem is strictly convex. Moreover, the condition $H \succ 0$ is not necessary and can be relaxed to $H \succeq 0$ as long as $Z\Phi Z^\top \succ 0$ holds.

Finally, it should be noted that the term $Z\top\Phi Y \Delta x_y$ vanishes if a feasible point is known. Otherwise, it may be calculated as:

$$(CY)\Delta z_y = d - Cz.$$

From which it is clear that $r_e = 0$ implies $\Delta z_y = 0$. Now, if the first block equation in (4.15) is multiplied by $Y^\top$ (instead of $Z^\top$), one may calculate the Lagrange multipliers as:

$$(CY)^\top \Delta \mu = -Y^\top r_d - Y^\top \Phi \Delta x$$

Where $CY$ is square and full-rank.

The null-space method requires the explicit calculation of the reduced Hessian $Z^\top \Phi Z$ and the basis $Y$. In turn, it solves a reduced system when compared to (4.15). The reduced Hessian has dimension $T \cdot m$, and can be factorized using the Cholesky method with complexity $\mathcal{O}(T^3 m^3)$.

However, we shall note that null-space methods are not advantageous in the scope of interior point methods. First, even if the matrix $Z$ can be precomputed, the matrix-matrix product $Z^\top \Phi Z$ has to evaluated at every iteration (since $\Phi$ is not constant), with cubic complexity on the matrix dimensions [7]. Second, the reduced Hessian is a dense matrix even if $\Phi$ and $C$ are sparse. Hence, sparse factorization techniques cannot be used, as opposed to range space methods.

The popularity of the null-space method comes from its use in the scope of active set methods. In ASM, the equality constraints are eliminated via dense MPC formulation, and the matrix $C$ in (4.15) corresponds to the so-called *working set*, which is a subset of the inequality constraints. If there are many constraints in the working set, which is usually the case in MPC problems, the space of the free variables is small and the reduced Hessian is a small matrix, which justifies the use of the null-space method [13].

## Range Space Methods

The range-space method is also called the *Shur complement method*, because it requires forming the Shur Complement of the KKT matrix. Here, we will introduce the range-space method following the same approach used to present the null-space methods. The first block equation in (4.15) is eliminated via:

$$\Delta z = -\Phi^{-1} C^\top \Delta\nu - \Phi^{-1} r_d, \tag{4.16}$$

and the expression for $\Delta z$ is substituted into the second block equation, yielding:

$$\left(C\Phi^{-1}C^\top\right)\Delta\nu = r_e - C\Phi^{-1} r_d. \tag{4.17}$$

The matrix $C\Phi^{-1}C^\top$ is the Shur Complement of the augmented KKT matrix. Note that (4.17) is a system in $Tn$ variables, and thus can be solved with complexity $\mathcal{O}(T^3 n^3)$. Moreover, since we impose the condition $H \succ 0$ (from which $\Phi \succ 0$), the Shur Complement is positive definite, and thus a Cholesky factorization scheme can be used.

There are two main disadvantages in the range-space method. First of all, it expects the augmented Hessian $\Phi$ to be non-singular. This requires, in general, the condition $H \succ 0$, which is more restrictive than $Z^\top H Z \succ 0$. This is the reason why interior point solvers usually require all the penalty matrices to be positive definite: $R, Q, P \succ 0$. As opposed to this, general purpose solvers allow $Q, P \succeq 0$, since $R \succ 0$ is enough to guarantee that the reduced Hessian $Z^\top H Z$ is positive definite.

The second disadvantage of the range-space method is that the expression for the Shur Complement involves $\Phi^{-1}$. The augmented Hessian has dimension $T(n+m)$, and hence its factorization has complexity $\mathcal{O}(T^3(n+m)^3)$. We need to factorize $\Phi$ to obtain the product $C\Phi^{-1}C^\top$, rather than explicitely computing the inverse. Note that this

step is previous to solving the linear system of equations. After forming the Shur Complement, incurring in the aforementioned cost, the Shur Complement has to be factored itself.

The reason that justifies the utilization of a range-space method is that the matrices $\Phi$ and $C\Phi^{-1}C^\top$ are sparse and thus can be factored at a reduced cost. In particular, we will show that the range-space method allows to solve the KKT system (4.15) with complexity $\mathcal{O}(T(n^3+m^3))$, which is linear in the prediction horizon. If we discard numeric constants and drop lower order terms, the complexity of different solution methods for the KKT system in sparse formulation are listed in Table 4.1.

| Solution method | Direct factorization | Null-space | Range-space |
|---|---|---|---|
| Complexity | $\mathcal{O}(T^3(n+m)^3)$ | $\mathcal{O}(T^3m^3)$ | $\mathcal{O}(T(n^3+m^3))$ |

Table 4.1: Complexity of solving the KKT system in sparse MPC formulation

## Tailored Factorization of the augmented Hessian

MPC problems have a very special structure that allows for a tailored, efficient factorization of the matrices $\Phi$ and $C\Phi^{-1}C^\top$. The main work in this area is due to [21] and [22]. Their procedure will be reproduced here, since it is decisive for the chosen software desing.

First, we introduce the notation for block diagonal matrices as follows. We also introduce the notation for block vectors.

**Definition 13** (Block diagonal matrix)**.** A block diagonal matrix is described using the notation:

$$\begin{aligned} A &= \text{block}\{A_k\}_0\top \\ &= \begin{pmatrix} A_0 & 0 & \dots & 0 & 0 \\ 0 & A_1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & A_{T-1} & 0 \\ 0 & 0 & \dots & 0 & A_T \end{pmatrix} \end{aligned}$$

**Definition 14** (Block vector)**.** The block diagonal matrix notation is used to describe block vectors as well. If $x \in \mathbb{R}^{T\cdot n}$ has $T$ blocks of

dimension $n$, we write:

$$\begin{aligned} x &= \text{block}\{x_k\}_0 \top \\ &= \begin{pmatrix} x_1^\top & x_2^\top & \dots & x_T^\top \end{pmatrix}^\top \end{aligned}$$

The vectors $\nu$, $\lambda$ and $s$ are block-vectors, which are partitioned as follows:

$$\nu = \text{block}\{\nu_k\}_0^{T-1} = \begin{pmatrix} \nu_0^\top & \nu_1^\top & \dots & \nu_{T-1}^\top \end{pmatrix}^\top,$$

$$\lambda = \text{block}\left\{ \begin{pmatrix} \lambda_k^{(u)} \\ \lambda_k^{(x)} \end{pmatrix} \right\}_0^{T-1} = \begin{pmatrix} \lambda_0^{(u)} \\ \lambda_0^{(x)} \\ \lambda_1^{(u)} \\ \vdots \\ \lambda_{T-1}^{(u)} \\ \lambda_{T-1}^{(x)} \end{pmatrix},$$

$$s = \text{block}\left\{ \begin{pmatrix} s_k^{(u)} \\ s_k^{(x)} \end{pmatrix} \right\}_0^{T-1} = \begin{pmatrix} s_0^{(u)} \\ s_0^{(x)} \\ s_1^{(u)} \\ \vdots \\ s_{T-1}^{(u)} \\ s_{T-1}^{(x)} \end{pmatrix}.$$

where $\nu_i \in \mathbb{R}^n$, $\lambda_i^{(u)} \in \mathbb{R}^{\kappa_u}$, $\lambda_i^{(x)} \in \mathbb{R}^{\kappa_x}$ and $\lambda_{T-1}^{(x)} \in \mathbb{R}^{\kappa_t}$. The elements in $s$ have the same dimensions as the corresponding elements in $\lambda$. Finally, define the vector $d$ as:

$$d = \lambda./s = \begin{pmatrix} d_0^{(u)} \\ d_0^{(x)} \\ d_1^{(u)} \\ \dots \\ d_{T-1}^{(u)} \\ d_{T-1}^{(x)} \end{pmatrix}^\top,$$

where the operator $./$ represents the elementwise division. We also define $D \coloneqq \text{diag}(d)$.

Using the notation above, the augmented Hessian is formed as:

$$\begin{aligned}\Phi &= \text{block}\left\{ \begin{pmatrix} \Phi_{r,k} & 0 \\ 0 & \Phi_{q,k} \end{pmatrix} \right\}_0^{T-1} \\ &= \text{block}\left\{ \begin{pmatrix} R + F_u^\top D_k^{(u)} F_u & 0 \\ 0 & Q_k + F_{x,k}^\top D_k^{(x)} F_{x,k} \end{pmatrix} \right\}_0^{T-1}\end{aligned}$$

where

$$Q_k = \begin{cases} Q & k = 0, \ldots, T-2 \\ P & k = T-1 \end{cases}$$

$$F_{x,k} = \begin{cases} F_x & k = 0, \ldots, T-2 \\ F_t & k = T-1. \end{cases}$$

We can obtain the inverse of a block diagonal matrix by inverting each of its blocks, which produces a block diagonal matrix as well. This is used to evaluate the Shur Complement of the KKT matrix, which has a tri-diagonal structure:

$$S = \begin{pmatrix} S_{11} & S_{12} & 0 & \ldots & 0 & 0 \\ S_{21} & S_{22} & S_{23} & \ldots & 0 & 0 \\ 0 & S_{32} & S_{33} & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & S_{T-1,T-1} & S_{T-1,T} \\ 0 & 0 & 0 & \ldots & S_{T,T-1} & S_{T,T} \end{pmatrix}$$

Where

$$S_{11} = B\Phi_{r,0}^{-1} B\top + \Phi_{x,0}^{-1} \tag{4.18}$$

$$S_{ii} = A\Phi_{x,i-2}^{-1} A\top + B\Phi_{r,i-1}^{-1} B\top + \Phi_{x,i-1}^{-1} \tag{4.19}$$

$$S_{i,i+1} = S_{i+1,i}\top = -\Phi_{x,i-1}^{-1} A\top \tag{4.20}$$

To form the blocks (4.18) - (4.20), we need to factorize all the blocks in $\Phi$. This involves a computational complexity $\mathcal{O}(Tn^3 + Tm^3)$, which is much cheaper than the factorization of $\Phi$ as a whole; *i.e.*, $\mathcal{O}(T^3(m+n)^3)$.

In a similar way, the Shur Complement $S$ should not be factored as a whole. Rather than this, we will do a blockwise factorization. The matrix $S$ is positive definite by construction, and hence it admits the

Cholesky decomposition $S = LL^\top$, where $L$ is lower triangular. The factor $L$ has the form:

$$L = \begin{pmatrix} L_{11} & 0 & 0 & \dots & 0 & 0 \\ L_{21} & L_{22} & 0 & \dots & 0 & 0 \\ 0 & L_{32} & L_{33} & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & L_{T-1,T-1} & 0 \\ 0 & 0 & 0 & \dots & L_{T,T-1} & L_{TT} \end{pmatrix}$$

Where the blocks $L_{ii}$ are $n \times n$ lower triangular matrices with positive diagonal entries, and the blocks $L_{i+1,i}$ are general, dense $n \times n$ matrices. From $Y = LL^\top$ one obtains:

$$S_{11} = L_{11}L_{11}^\top \tag{4.21}$$

$$S_{i,i+1} = L_{i,i}L_{i+1,i}^\top \qquad 1 \leq i < T \tag{4.22}$$

$$S_{i,i} - L_{i,i-1}L_{i,i-1}^\top = L_{i,i}L_{i,i}^\top \qquad 2 \leq i \leq T \tag{4.23}$$

Note that both (4.21) and (4.23) involve the factorization of a $(n \times n)$ matrix, whereas (4.22) consists in $n$ backwards substitutions against a triangular matrix. All this operations have cubic complexity. Hence, if we drop numeric constants, the complexity of factoring the Shur Complement is $\mathcal{O}(Tn^3)$, which is much cheaper than the factorization of a dense matrix of equal size (with complexity $\mathcal{O}(T^3n^3)$).

Next, we will show how the blocks in the Shur Complement can be used to compute the right hand side of the system, thus avoiding the large matrix multiplication $C\Phi^{-1}$ in (4.16). First, note that the stationarity residual in (4.2) can be expressed block-wise and in terms of inputs and states. The definition of $r_c$ is reproduced here for convenience:

$$r_c = Hz + F^\top\lambda + C^\top\mu + h.$$

The vector $r_c$ may be subdivided in $T$ blocks, each of them for one time step. It may be further subdivided into input and state components, resulting in:

$$\begin{aligned} r_c &= \text{block}\left\{ \begin{pmatrix} r_{cu}[k] \\ r_{cx}[k] \end{pmatrix} \right\}_{k=0}^{T-1} \\ &= \text{block}\left\{ \begin{pmatrix} Ru[k] + F_u^\top\lambda_u[k] + B^\top\nu[k] \\ Qx[k] + F_x^\top\lambda_x[k] + A^\top\nu[k+1] - \nu[k] \end{pmatrix} \right\}_{k=0}^{T-1} \end{aligned}$$

Where the step dependance has been made explicit between brackets for clarity. Also, note that $r_{cx}[T-1]$ has a slightly different expression in term of the terminal penalty and the terminal constraints; namely:

$$r_{cx}[T-1] = Px[k] + F_t^\top \lambda_x[k] - \nu[k]$$

We perform the same analysis on $r_d$, whose definition is reproduced here for convenience:

$$r_d = r_c + F^\top S^{-1}(\Lambda r_i - r_s)$$

We can express $r_d$ blockwise and in terms of inputs and states.

$$\begin{aligned} r_d &= \text{block}\left\{ \begin{pmatrix} r_{du}[k] \\ r_{dx}[k] \end{pmatrix} \right\}_{k=0}^{T-1} \\ &= \text{block}\left\{ \begin{pmatrix} r_{cu}[k] + F_u^\top D_u[k] r_{iu}[k] - F_u^\top (\text{diag}(s_u[k]))^{-1} r_{su}[k] \\ r_{cx}[k] + F_x^\top D_x[k] r_{ix}[k] - F_x^\top (\text{diag}(s_x[k]))^{-1} r_{sx}[k] \end{pmatrix} \right\}_{k=0}^{T-1} \end{aligned}$$

Let the right-hand-side for the Shur Complement be denoted by $r_{sc}$, which is presented in (4.16) and reproduced here:

$$r_{sc} = r_e - C\Phi^{-1} r_d$$

We can express $r_{sc}$ blockwise as in the previous cases:

$$r_{sc} = \text{block}\big\{ r_e[k] - A\Phi_q^{-1}[k-1] r_{dx}[k-1] - B\Phi_r^{-1}[k] r_{du}[k] + \Phi_q^{-1}[k] r_{dx}[k] \big\}_{k=0}^{T-1} \tag{4.24}$$

Where the second term drops for $k = 0$. Here, it is imporant to note that the matrix products $A\Phi_q^{-1}[k]$ and $B\Phi_r^{-1}[k]$ have already been computed in the construction of the Shur Complement, and thus shall not be recomputed again. For this reason, the evaluation of $r_{sc}$ is inexpensive in spite of the apparently large matrix product $C\Phi^{-1}$.

## 4.3 Inclusion of Soft Constraints

A major concern in MPC applications is the feasibility of the generated optimization problem. If the problem is not feasible, the controller fails to provide an input for the controlled system, with potentially disastrous consequences. One approach to handle infeasible problems is to delegate to a back-up controller if MPC fails, such as a traditional

PID or even a human operator [6]. However, a much popular approach is *constraint softening*, which allows some constraints to be violated and thus enforces feasibility.

We say that a constraint is *hard* if it cannot be violated at all. As opposed to it, *soft* constraints might be violated at a penalty. The main idea is that there should exist no penalty if the soft constraint is fulfilled, whereas any violation of it shall be penalized in the cost function.

There are different ways of implementing constraint softening. One approach attempts to solve the problem with hard constraints, and if it is infeasible, constraints are softened one by one until feasibility is attained. However, this requires to iteratively solve a sequence of feasibility problems, followed by an optimization problem, which is computationally expensive. Hence, the most popular approach is to allow some constraints to be soft from the beginning.

In the MPC framework, there is a straightforward approach for choosing which constraints should be softened. Input constraints are usually handled as hard constraints, since they represents physical limitations of the system (*e.g.*, the opening of a valve shall not exceed 100%). On the other hand, state constraints usually refer to control goals, and thus it makes sense to soften them (*e.g.*, the temperature of the combustion chamber should not exceed 1450$^{o}$C, but it can definitely be surpassed if that is the only way of ensuring the stability of the system).

This distinction between input and state constraints has a further advantage. The set of input constraints does never hinder feasibility, since the inputs are the free variables and thus we can choose them without any additional restriction. On the other hand, the states shall obey the system dynamics and we cannot choose them freely. Infeasibility comes from state constraints, and thus it makes sense to let them be soft, which guarantees that the optimization problem is always feasible.

It is worth noting that there is no formal distinction between soft and hard constraints from a mathematical point of view. Every single constraint in an optimization problem is, by definition, a hard constraint. Soft constraints are not constraints, formally, since they do appear in the objective function. However, the distinction between hard and soft constraints makes full sense in the field of automatic control.

When implementing constraint softening, we would like to enforce that no soft constraint is violated unless it is strictly necessary. *Exact* constraint softening is the procedure that produces best results [11]. It

penalizes the violation of the soft constraints by using additional slack variables. For illustrative purposes, consider the following optimization problem, with a single constraint that we would like to soften:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \mathrm{f}(x) \\ \text{subject to} \quad & \mathrm{g}(x) \leq 0 \\ & x \in \mathbb{R}^n \end{aligned}$$

The introduction of a slack variable $\varepsilon \in \mathbb{R}$ allows for exact constraint softening:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \mathrm{f}(x) + \rho\varepsilon^2 \\ \text{subject to} \quad & \mathrm{g}(x) + \varepsilon = 0 \\ & \varepsilon \geq 0 \\ & x \in \mathbb{R}^n \end{aligned}$$

The scalar $\rho$ is a tuning parameter that allows us to decide how important is the violation of the soft constraint. If it is large, the solver will put emphasis in solving the problem without any constraint violation. In the limit $\rho \to \infty$, the problem is hard constrained.

Though this approach provides good results, given an adequate tuning of the parameter $\rho$, it has several disadvantages. First of all, it increases the total size of the problem by introducing a new variable (the slack $\varepsilon$) and a new constraint (since the slack variables must be enforced to be positive) for every soft constraint in the problem. In the MPC framework, this effect is scaled by $T$. Here we shall mention that the complexity can be limited by introducing at most one slack variable per prediction step [11]. As a drawback of this approach, the upper limit of one slack variable per prediction step requires using the $1$-norm or the $\infty$-norm, which makes the objective function non-smooth.

In addition, introducing slack variables destroys the block diagonal structure in the augmented Hessian $\Phi$, and hence the block tri-diagonal structure in the Shur Complement $C\Phi^{-1}C^\top$, which is the main advantage of the sparse MPC formulation.

A different, novel approach is proposed in [11]. The author suggests to use the *Kreisselmeier-Steinhauser* function to approximate the maximum violation of the soft constraints, which is then penalized. To do so, the following additional term is introduced in the cost function,

where $\mathrm{g}(x) = \{\mathrm{g_j}(x)\}_0^J$ is the set of all soft constraints:

$$\mathrm{KS}(\mathrm{g}(x)) = \frac{1}{\rho}\log\left(\sum_{j=0}^{J} e^{\rho\,\mathrm{g_j}(x)}\right)$$

Where $\mathrm{KS}(\mathrm{g}(x))$ is the Kreisselmeier-Steinhauser function, with the property:

$$\lim_{\rho\to\infty} \mathrm{KS}(\mathrm{g}(x)) = \max_j \mathrm{g_j}(x)$$

Because the KS function approximates the maximum violation of all soft constraints, penalizing it effectively disencourages constraints violation. In addition, the KS function is convex and hence does not perturb the convexity properties of the QP.

To include the KS term into the KKT system, both its gradient and its Hessian shall be calculated. Let the soft constrained problem be expressed as:

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & z^\top H z + \mathrm{KS}\Big(\tilde{F}z - \tilde{f}\Big) \\ \text{subject to} \quad & Fz \leq f, \\ & Cz = d, \end{aligned}$$

Which is entirely analogous to (2.17) but for the inclusion of the soft constraint $\tilde{F}z \leq \tilde{f}$ in the objective function. The same procedure presented in the last sections can be used with the sole modification:

$$\begin{aligned} \tilde{\Phi} &= \Phi + \rho\tilde{F}^\top \operatorname{diag}\Big(\hat{d}\Big)\tilde{F} \\ \tilde{r}_d &= r_d + \tilde{F}^\top \tilde{d} \end{aligned}$$

We use tilde in $\tilde{\Phi}$ and $\tilde{r}_d$ to refer to the augmented Hessian and the right-hand-side residuals after the inclusion of the soft constraints. The vectors $\hat{d}$ and $\tilde{d}$ are calculated as in [11]:

$$\begin{aligned} \tilde{d}_i &= \frac{e_i^+}{1+e_i^+}, && \forall i, \\ \hat{d}_i &= \frac{e_i^+}{(1+e_i^2)^2}, && \forall i \end{aligned}$$

Where the scalars $e_i^+$ are the exponential function of the soft constraint violations:

$$e_i^+ = e^{\rho\left(\tilde{F}_i z - \tilde{f}_i\right)}$$

Where $\tilde{F}_i$ is the $i$-th row in the matrix $\tilde{F}$.

This approach has several advantages. In the first place, it preserves the sparsity and structure of the sparse MPC formulation, which allows the efficient, sparse factoring and solving methods introduced before. In the second place, it is similar to the exact constraint softening (given that $\rho$ is sufficiently large) without the addition of extra variables. Finally, the computational cost is linear in the number of soft constraints (*i.e.*, $\mathcal{O}(\tilde{K})$, where $\tilde{K}$ is the total number of soft constraints).

The tailored solution for the KKT system, with the inclusion of the soft constraints, is depicted in Algorithm 2. Note that the Shur Complement is factored as soon as the blocks in the diagonal and sub-diagonal are available, instead of forming the whole matrix before factoring it. The reason for this is called *interleaving* and improves the execution of the code in modern computer architectures, as reported in [22]. The algorithm for the centering plus correction step is the same, but for the factorization of the Shur Complement (which is already available). For completeness, the centering plus correction step is depicted in Algorithm 3.

**Data:** $A$, $B$, $R$, $Q$, $P$, $r_c$, $r_e$, $r_i$, $r_s$
**Result:** $\xi^{\text{aff}} = \left(z^{\text{aff}}, \nu^{\text{aff}}, \lambda^{\text{aff}}, s^{\text{aff}}\right)^\top$
**Initialization:** $\xi = (z_k, \nu_k, \lambda_k, s_k)^\top$, $k = 0$

Form $S_{1,1}$ /* First block */
Form $r_{sc}[0]$
$L_{1,1} \leftarrow \text{cholfact}(S_{11})$
$\nu_0^{\text{aff}} \leftarrow L_{1,1} \backslash r_{sc}[0]$

**for** $k = 2, \ldots, T$ **do** /* Forward solve */

  Form $S_{k,k}$ and $S_{k,k-1}$ /* Shur Complement */
  Form $r_{sc}[k]$ /* residual as in (4.24) */
  $L_{k,k-1} \leftarrow L_{k-1,k-1} \backslash S_{k+1,k}^\top$
  $L_{k,k} \leftarrow \text{cholfact}\left(S_{k,k} - L_{k,k-1} L_{k,k-1}^\top\right)$
  $\nu_{k-1}^{\text{aff}} \leftarrow r_{sc}[k] - L_{k,k-1} \nu_{k-2}^{\text{aff}}$
  $\nu_{k-1}^{\text{aff}} \leftarrow L_{k,k} \backslash \nu_{k-1}^{\text{aff}}$
**end**

$\nu_{T-1}^{\text{aff}} \leftarrow L_{T,T}^\top \backslash \nu_{T-1}^{\text{aff}}$ /* Last block */
**for** *k = T-1,. . . ,1* **do** /* Backward solve */

  $\nu_{k-1}^{\text{aff}} \leftarrow \nu_{k-1}^{\text{aff}} - L_{k+1,k}^\top \nu_k^{\text{aff}}$
  $\nu_{k-1}^{\text{aff}} \leftarrow L_{k,k}^\top \backslash \nu_{k-1}^{\text{aff}}$
**end**

**Algorithm 2:** Solution of the KKT system for the affine step

**Data:** $\sigma$, $\mu$
**Result:** $\xi^{\text{cc}} = (z^{\text{cc}}, \nu^{\text{cc}}, \lambda^{\text{cc}}, s^{\text{cc}})^\top$
**Initialization:** $r_s^{\text{cc}} = S^{\text{aff}}\Lambda^{\text{aff}}e - \sigma\mu e$

Form $r_{sc}[0]$ /* Residual */
$\nu_0^{\text{cc}} \leftarrow L_{1,1} \backslash r_{sc}[0]$

**for** $k = 2, \ldots, T$ **do** /* Forward solve */

  Form $r_{sc}[k]$ /* Residual */
  $\nu_{k-1}^{\text{cc}} \leftarrow r_{sc}[k] - L_{k,k-1}\nu_{k-2}^{\text{cc}}$
  $\nu_{k-1}^{\text{cc}} \leftarrow L_{k,k} \backslash \nu_{k-1}^{\text{cc}}$
**end**
$\nu_{T-1}^{\text{cc}} \leftarrow L_{T,T}^\top \backslash \nu_{T-1}^{\text{cc}}$ /* Last block */
**for** *k = T-1,. . . ,1* **do** /* Backward solve */

  $\nu_{k-1}^{\text{cc}} \leftarrow \nu_{k-1}^{\text{cc}} - L_{k+1,k}^\top \nu_k^{\text{cc}}$
  $\nu_{k-1}^{\text{cc}} \leftarrow L_{k,k}^\top \backslash \nu_{k-1}^{\text{cc}}$
**end**

**Algorithm 3:** Solution of the KKT system for the centering plus correction step

# Chapter 5

# System Architecture

Interior Point Methods achieve convergence in a low, rather constant number of steps, at the cost of computationally expensive iterations. There are two means of accelerating the execution of an interior point algorithm applied to MPC. First, the Mehrotra Predictor-Corrector algorithm is used to produce search directions of better quality and hence reduce the require number of iterations. Second, the sparse factorization of $\Phi$ and $C\Phi^{-1}C^{\top}$ in the scope of the range-space method attempts to reduce to the minimum the complexity of each iteration. In a similar way, the introduction of soft constraints as in [11] pursuits maintaining the complexity of each iteration as low as possible.

In spite of all the theoretical properties of the proposed algorithm, an efficient software implementation is crucial to attain efficiency and low computation times. In this chapter, we explain the architecture of the software. Our design is highly modular, and attempts to exploit every simplificatory characteristic of the user's MPC problem. First, we will explain these singularities, and how we can capitalize on them. Next, we present the modular design in the programming language C++.

## 5.1 Strategies for computing and factorizing $\Phi$ depending on the Type of Constraints

We know from Chapter 4 that the most expensive steps in the solution of the KKT system are forming the Shur Complement $C\Phi^{-1}C^{\top}$ and factorizing it. These steps involve matrix-matrix products and

matrix factorizations, all of them with cubic complexity in the matrix dimensions.

The Cholesky factorization of the Shur Complement is independent of the MPC problem formulation, since $C\Phi^{-1}C^\top$ and $L$ always have the same structure. Contrarily, the procedure for computing $\Phi$ and $C\Phi^{-1}C^\top$ may be optimized out by exploiting some characteristics of the original MPC formulation. A comprehensive complexity analysis is available in [22, Section III]. Here, we present the results that are relevant for the software design.

Recall that the matrix $\Phi$ has the form:

$$\Phi = \text{block}\left\{ \begin{pmatrix} \Phi_r[k] & 0 \\ 0 & \Phi_q[k] \end{pmatrix} \right\}_{k=0}^{T-1}$$

To simplify the notation, we limit the following analysis to one single block in $\Phi$. Moreover, we reference just the input constraints block $\Phi_r$, where we drop the dependance of $k$. However, the analysis is valid for all the blocks $\Phi_r[k]$ and $\Phi_q[k]$ in $\Phi$.

The blocks in $\Phi_r$ are formed by adding two terms to the penalty matrix $R$. The first term corresponds to the hard constraints, whereas the second one corresponds to the soft constraints.

$$\Phi_r = R + \underbrace{F_u^\top D_u F_u}_{\text{Hard constraints}} + \underbrace{\tilde{F}_u^\top \operatorname{diag}\left(\hat{d}_u\right) \tilde{F}_u}_{\text{Soft constraints}}, \tag{5.1}$$

where $D_u = D_k^{(u)} = \operatorname{diag}\left(\lambda_k^{(u)} ./ s_k^{(u)}\right)$. In general, $F_u$ is a dense matrix with $\kappa_u$ rows and $m$ columns. Hence, forming every block in $\Phi_r$ requires an expensive matrix-matrix multiplication. However, we can do better in the presence of upper and lower bounds on the variables (as opposed to linear constraints).

Simple bounds are expressed in the form:

$$lb_u \leq u_k \leq ub_u.$$

Nontheless, we may express them as linear constraints $F_u u_k \leq f_u$, by rewriting them as:

$$\underbrace{\begin{pmatrix} I \\ -I \end{pmatrix}}_{F'_u} u_k \leq \underbrace{\begin{pmatrix} ub_u \\ -lb_u \end{pmatrix}}_{f'_u}$$

where $I$ is the identity matrix of appropriate dimension. The constraints matrix $F'_u$ arising from simple bounds is not dense, but very sparse and structured. If we divide $D_u$ in four blocks, and limit the constraints to be simple bounds, the second term in 5.1 simplifies to:

$$\begin{pmatrix} I & -I \end{pmatrix} \begin{pmatrix} \mathrm{diag}(d_1) & 0 \\ 0 & \mathrm{diag}(d_2) \end{pmatrix} \begin{pmatrix} I \\ -I \end{pmatrix} = \mathrm{diag}(d_1 + d_2) \tag{5.2}$$

In (5.2) we divide the diagonal of $D_u$ in two vectors $d_1$ and $d_2$. This motivates a distinction between the dual variables associated with linear constraints, and those associated with simple bounds. So, consider the vector $d_k \coloneqq d = \lambda_k./s_k$. It can be divided into input and state variables, originating the separation $d = \begin{pmatrix} d_u & d_x \end{pmatrix}^\top$. We can further subdivide it according to the type of constraints. Under such considerations, the vector $d_u$, associated with input variables, can be divided as:

$$d_u = \begin{pmatrix} d_{u1} \\ d_{u2} \\ d_{u3} \end{pmatrix}^\top$$

Where $d_{u1}$, $d_{u2}$ and $d_{u3}$ refer to the upper bounds, lower bounds and linear constraints, respectively. In matrix form, we have:

$$\begin{aligned} D_u &= \begin{pmatrix} \mathrm{diag}(d_{u1}) & 0 & 0 \\ 0 & \mathrm{diag}(d_{u2}) & 0 \\ 0 & 0 & \mathrm{diag}(d_{u3}) \end{pmatrix} \\ &= \begin{pmatrix} D_{u1} & 0 & 0 \\ 0 & D_{u2} & 0 \\ 0 & 0 & D_{u3} \end{pmatrix} \end{aligned} \tag{5.3}$$

The substitution of (5.2) and (5.3) into (5.1) yields:

$$\Phi_r = R + D_{u1} + D_{u2} + F_u^\top D_{u3} F_u + \hat{D}_{u1} + \hat{D}_{u2} + \tilde{F}_u^\top \hat{D}_{u3} \tilde{F}_u \tag{5.4}$$

Where $\hat{d}_u$ and $\hat{D}_u$ have been subdivided exactly as $d_u$ and $D_u$. This means that we can save some expensive matrix-matrix multiplications by handling simple bounds and linear constraints in a different way. The addition of a diagonal matrix has complexity $\mathcal{O}(n)$, whereas a matrix product has complexity $\mathcal{O}(n^3)$.

Nevertheless, there is more that we can do when forming the Shur Complement $C\Phi^{-1}C^\top$. For convenience, we reproduce here the expressions for the diagonal and subdiagonal blocks in S:

$$S_{k,k} = A\Phi_{q,k-1}^{-1}A^\top + B\Phi_{r,k}^{-1}B^\top + \Phi_{q,k}^{-1}$$
$$S_{k,k+1} = S_{k+1,k}\top = -\Phi_{q,k}^{-1}A^\top$$

Consider the products in the form $B\Phi_{r,k}^{-1}B^\top$, where $B$ and $\Phi_{r,k}$ are dense matrices and $\Phi_{r,k}$ is positive definite. The same reasoning applies to the products $A\Phi_{q,k}^{-1}A^\top$. Rather than explicitely computing $\Phi^{-1}$ and carrying out two matrix-matrix products, the standard procedure is to factorize the matrix $\Phi$ and then forward solve against $B$:

$$\Phi_{r,k} = L_{r,k}L_{r,k}^\top \tag{5.5}$$
$$X_{r,k} = L_{r,k}\backslash B^\top \tag{5.6}$$
$$B\Phi_{r,k}^{-1}B^\top = X_{r,k}^\top X_{r,k} = (BL_{r,k}^{-\top})(L_{r,k}^{-1}B^\top) \tag{5.7}$$
$$\Phi_{r,k}^{-1}B^\top = L_{r,k}^\top\backslash X_{r,k} = L_{r,k}^{-\top}L_{r,k}^{-1}B^\top \tag{5.8}$$

Steps (5.5) - (5.7) involve one matrix factorization, one forward substitution against a lower triangular matrix and a symmetric matrix-matrix product. Equation (5.8), which is needed for calculating the subdiagonal blocks, is a forward substitution againts a lower triangular matrix.

However, this is just the general case for dense $\Phi_r$. There are several scenarios which result in cheaper computations, which we shall exploit. In the following, we will reference just the matrices $\Phi_r$ and the products $B\Phi_r^{-1}B^\top$ (where, again, we omit the dependance with $k$), but the same reasoning applies to $\Phi_q$ and $A\Phi_q^{-1}A^\top$.

We need some new nomenclature for the next sections. First, we let the user define constraints on inputs, states and the final state, both hard and soft. These are referred as *hard input constraints* and *soft input constraints*, in the input case, and similarly for the state and terminal constraints.

Furthermore, we distinguish four families of constraints. By now, it sufficies to know that constraints are tagged as *unconstrained*, *simple bounds*, *linear constraints* and *full constraints*.

In addition, we need to tag the input constraints as a whole, taking both the hard and soft constraints into consideration. For this reason, unless we say otherwise, by the general term *input constraints* we refer to all the constraints on the inputs. (The same applies to state and terminal constraints.) The tag, or family, of these generalized constraints

depends on the specific combination of the hard and soft constraints. All possible combinations are listed in Table 5.1.

| Hard \ Soft | **Unconstrained** | **Simple Bounds** | **Linear Constraints** | **Full Constraints** |
|---|---|---|---|---|
| **Unconstrained** | Unconstrained | Simple Bounds | Linear Constraints | Full Constraints |
| **Simple Bounds** | Simple Constraints | Simple Constraints | Full Constraints | Full Constraints |
| **Linear Constraints** | Linear Constraints | Full Constraints | Linear Constraints | Full Constraints |
| **Full Constraints** | Full Constraints | Full Constraints | Full Constraints | Full Constraints |

Table 5.1: Overall constraints classification depending on the hard and soft constraints

## No Constraints

The first complexity scenario is also the simplest one. Consider the case when the inputs are unconstrained (*i.e.*, there are neither hard constraints, nor soft constraints, according to Table 5.1). In such scenario, the matrix $\Phi_{r,k}$ is constant for all steps, and also across all iterations. For this reason, we pre-compute $\Phi_r^{-1}$ and $B\Phi_r^{-1}B^\top$ in the initialization of the algorithm and reuse them whenever needed.

## Simple Bounds

It is rather common to find problems with simple bounds on the inputs, but no linear constraints. In this case, we can avoid expensive matrix-matrix products and evaluate $\Phi_r$ as in (5.4). Furthermore, consider the case when the penalty matrix $R$ is diagonal (which is a typical design choice, given the difficulty of tuning a dense penalty matrix). If the penalty matrix is diagonal and the constraints are simple bounds, then the matrix $\Phi_r$ is diagonal and the procedure (5.5) - (5.8) can be avoided. Instead, the product $B\Phi_r^{-1}B^\top$ can be calculated directly, since the inverse of a diagonal matrix is explicitely available as the inverse of the diagonal elements.

In addition, note that, instead of doing two matrix products, $B\Phi_q^{-1}B^\top$ may be calculated as $m$ rank one updates on a zero matrix, which is

more efficient computationally. By expanding the product $B\Phi_q^{-1}B^\top$, we observe:

$$
\begin{aligned}
B\Phi_q^{-1}B^\top &= \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{pmatrix} \begin{pmatrix} \phi_{11} & 0 & \dots & 0 \\ 0 & \phi_{22} & \dots 0 & \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \phi_{nm} \end{pmatrix} \begin{pmatrix} b_{11} & b_{21} & \dots & b_{n1} \\ b_{12} & b_{22} & \dots & b_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ b_{1m} & b_{2m} & \dots & b_{nm} \end{pmatrix} \\
&= \sum_{i=1}^{m} \phi_{ii} \cdot \begin{pmatrix} b_{1i} \\ b_{2i} \\ \vdots \\ b_{ni} \end{pmatrix} \begin{pmatrix} b_{1i} & b_{2i} & \dots & b_{n1} \end{pmatrix} \\
&= \sum_{i=1}^{m} \phi_{ii} \cdot b_{:,i} b_{:,i}^\top
\end{aligned}
$$

where $b_{:,i}$ denotes the $i$-th column in $B$. Consider now the complexity of each rank-1 operation.

Let $x \in \mathbb{R}^n$. Then, the rank-1 operation $\alpha x x^\top$, for some scalar alpha, needs $n$ products to form $\alpha x$, and $n(n+1)/2$ floating point operations (*flops*) for the symmetric product $(\alpha x)x^\top$. Note that, since it is symmetric, we only need to reference the lower or the upper half. We have to repeat the rank-1 operation $m$ times; hence, the total cost is $mn(n+3)/2$ flops.

Next, consider the flop count for the full matrix-matrix multiplications. The product $\Phi_q^{-1}B^\top$ needs a total of $m \cdot n$ scalar multiplications. Then, the non-symmetric product $B \cdot (\Phi_q^{-1}B^\top)$ costs $n(n+1)m$ flops if just one half of the matrix is referenced. There are $mn(n+2)$ flops in total, which is around two times more than using rank-1 updates. In addition to it, the matrix-product method is less stable numerically, since the full product is not guaranteed to be symmetrical in presence of numerical inaccuracies.

## Linear Constraints

Assume that there are no bounds on the inputs, but there is a certain number $\kappa_u$ of linear constraints on them. We may compute the sum $R + F_u^\top D_u F_u$ as $\kappa_u$ rank-1 updates on $R$ and then factorize the result as in (5.5) - (5.8). However, there is a well known procedure for updating the Cholesky factor of a matrix subject to a rank-1 operation, without actually computing such operation.

In math: let $A \in \mathbb{R}^{n \times n}$ be a positive definite matrix, whose Cholesky decomposition $A = LL^\top$ we know. Let $x$ be a vector of dimension $n$. Then, the Cholesky decomposition of $\tilde{A} = A + \alpha x x^\top$, for any $\alpha > 0$, can be calculated without actually computing $\tilde{A}$. That is: we can obtain the factorization $\tilde{A} = \tilde{L}\tilde{L}^\top$ directly from $L$ and $x$. This procedure has a complexity of $\mathcal{O}(n^2)$, as opposed to the complexity $\mathcal{O}(n^3)$ of a recomputation of the Cholesky factorization from scratch.

In the case of a linearly constrained problem, with no bounds, the penalty matrices can be factored in the initialization of the algorithm, and the Cholesky factorizations of $\Phi$ in every iteration can be computed as rank-1 factor updates.

## Summary

In order to form the Shur complement we follow two steps. First, we obtain the augmented Hessian $\Phi_r$ and $\Phi_q$, and factorize it. Later, we use the factorization of the augmented Hessian to form $B\Phi_r^{-1}B^\top$ and $A\Phi_q^{-1}A^\top$. The complexity cases for the first step are summarized below for the constrain tags.

1. **Unconstrained:** $\Phi_r$ and $B\Phi_r^{-1}B^\top$ are constant. We evaluate them at the initialization of the algorithm.

2. **Simple Bounds:** We obtain $\Phi_r$ via an inexpensive diagonal matrix addition. Regarding the factorization of $\Phi_r$, there are two different scenarios:

   (a) **Diagonal Penalty Matrix:** We do not factorize $\Phi_r$, since it is diagonal and hence the inverse is also diagonal and straightforward to compute.

   (b) **Dense Penalty Matrix:** In this case, we need to do the full Cholesky factorization of $\Phi_r$. There is no efficient factor update routine for the addition of a diagonal matrix.

   (Note that this distinction also holds for the Unconstrained problem. However, it is less important in that case, since we perform the computations only once.)

3. **Linear Constraints:** We factorize $R$ at the initialization of the algorithm. In every iteration, we obtain the Cholesky decomposition of $\Phi_r$ by updating the factorization of $R$.

4. **Full Constraints:** If there are both bounds and linear constraints, then we shall compute $\Phi_r$ as in (5.4) and then factorize it. Recall that the terms in the form $F_u^\top D_u F_u$ shall be added as rank-1 operations in the rows of $F_u$, rather than doing the full matrix-matrix product.

The output of this step is the matrix $\Phi_r$, if it is diagonal, or the factorization $\Phi_r = L_r L_r^\top$ if it is dense. Note that $\Phi_r$ is positive definite by construction (due to $R \succ 0$ and $\lambda_i, s_i > 0, \forall i$). Also, note that we do not needed to compute $\Phi_r$ if it is dense. In such case, the Cholesky factorization $\Phi_r = L_r L_r^\top$ is all we need.

The next step is forming the products $B\Phi_r^{-1}B^\top$ and and $A\Phi_q^{-1}A^\top$, which are building blocks in the Shur complement. There are just two cases here:

1. $\Phi_r$ **is diagonal:** We compute $B\Phi_r^{-1}B^\top$ as $m$ rank-1 updates.

2. $\Phi_r$ **is dense:** We follow the procedure (5.6) - (5.8).

Whether $\Phi_r$ is diagonal or dense depends on the type of constraints and the penalty matrices. The different cases are depicted in Table 5.2.

| **Constraint type** | $\boldsymbol{R}$ **is diagonal** | $\boldsymbol{R}$ **is dense** |
|---|---|---|
| **Unconstrained** | $\Phi_r$ diagonal | $\Phi_r$ dense |
| **Simple Bounds** | $\Phi_r$ diagonal | $\Phi_r$ dense |
| **Linear Constraints** | $\Phi_r$ dense | |
| **Full Constraints** | $\Phi_r$ dense | |

Table 5.2: Structure of $\Phi_r$ as a function of the constraints and penalty types.

## 5.2 Enumeration of Use Cases

In our attempt to create an efficient MPC optimization solver, we would like to take advantage of all of these simplifications. However, note that a finely-tuned algorithm comes at a cost. In the following analysis, we reference only the input variables and the augmented Hessian $\Phi_r$, without any loss of generality.

First, consider just the hard input constraints. According to previous discussion, there are four different types of hard input constraints,

and each on them is implemented in a different way. However, that implementation also depends on the soft input constraints. For example, assume that the hard constraints are linear. In that case, we would like to do a Cholesky factor update, rather than adding them to the augmented Hessian and then factorizing it. Nevertheless, if the soft constraints are simple bounds (or full constraints), we shall not do factor update, but add the linear constraints in the standard way.

Hence, we have $4^2$ different use cases only for the input constraints. Moreover, $3$ of them are tagged either as "unconstrained" or "simple bounds" (Table 5.1). In these cases, there is an important difference depending on whether the penalty matrix is diagonal or dense. Taking it into account, there are $4^2 + 3$ use cases.

Now consider that the user has the freedom to choose input constraints, state constraints and terminal constraints independently. If they are taken all together, there are $(4^2 + 3)^3 = 6859$ possible specializations of the algorithm.

Naturally, we do not have to enumerate all the use cases. We can explore the tree using inexpensive `if-else` and resolve to the appropriate function or piece of code. However, this approach has several disadvantages.

In first place, the software must carry out these `if-else` comparisons in every iteration of the algorithm. Even though comparisons are cheap when compared to matrix factorizations, this adds some overhead to the process, which we would like to avoid.

In second place, excesive branching is, in general, a bad coding practice. Even though it does not affect the quality of the code at first glance, it makes the code hard to understand and, more important, to maintain. Any single change in code may affect any branch, and hence all of them must be checked systematically to avoid new problems.

Finally, and most importantly, excesive branching translates into a larger memory footprint. The reason is dual. To begin with, since we are dynamically resolving to the appropriate branches, all of them must exist at compile time. (In other words: even if we do not explore some branches, the corresponding code must exist, and thus take some memory.) Secondly, assume that we use static memory allocation[1],

[1]Static memory allocation means the memory required for all variables and vectors is allocated when the program starts. The variables size is fixed and must be known when the program is created (*i.e.*at compile time). In turn, the total code size is known a priori and we can make sure that it fits into an embedded device.

which is required in some industrial applications. Then, we must allocate memory for all the possible use cases, even if we do not use it due to the `if-else` resolution.

One possible solution to this problem is to code all the $6859$ use cases independently. However, even if we reuse code, this is a titanic task and an unrealistic goal. Debugging and maintaining the code would become cumbersome as well.

## 5.3 Policy-Based Design for a Catch-All Algorithm with no Branches

The correct and efficient implementation of the proposed algorithm is a Computer Science question. In order to have a customizable algorithm without `if-else` branching, we have chosen to follow the *Policy-Based Design* principle. As a result, our numerical solver is a *polymorphic* object, that can modify its *behaviour* at *compile time* according to some *inheritance* chains.

In short, the behaviour of the solver is the branch of the algorithm that better suits the problem formulation. The key is that the resolution to the correct branch is accomplished at compile time. Hence, there is no `if-else` branching at run time. We do not have to code every instance of the algorithm either, since the different branches are coded independently.

A proper description of the Policy-Based Design principle requires the explanation of the concepts of "polymorphism", "inheritance" and "generic programming". The interested reader is refererd to Appendix A for a introduction of them. A thorough explanation of inheritance and Policy Based Design is available under [25], [26].

In the Policy-Based Design paradigm, we decompose our algorithm into policies, each of them implementing a different behaviour. An *inheritance diagram* is used to depict how the policies interact with each other. In this type of diagrams, an arrow from `A` to `B` means that `B` inherits from `A`. Figure 5.1 depicts the decomposition of our algorithm into policies.

Each box (apart from the bottommost "Solver") names a type of policy. For instance, the blue box on the left requires a policy capable of managing the input constraints of an MPC problem. We can think of Figure 5.1 as a diagram of empty boxes, which needs to be filled with

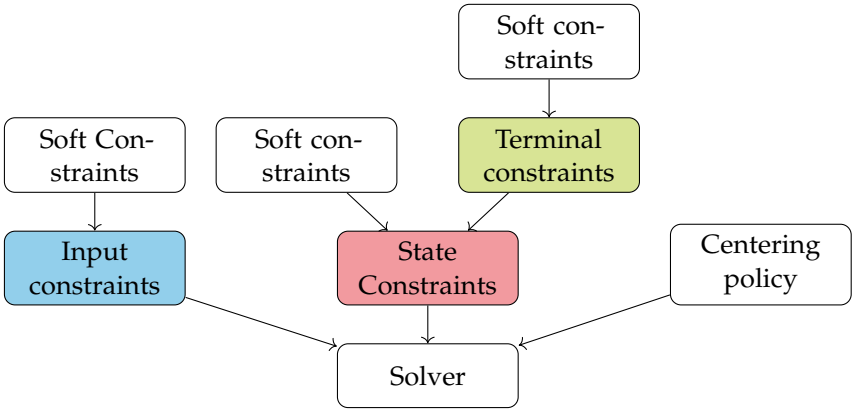


Figure 5.1: Inheritance diagram for our software.

appropriate policies. When all the boxes are filled, a particular instance of the algorithm is coded. In the following, we may refer to the box "Solver" as the *shell* of the algorithm, since it works as a host for the branches that are implemented via policies.

We have highlighted in color the three main types of constraints that we have in our design. The terminal constraints are a particular case of state constraints, and for that reason we let the object `State constraints` handle both of them. For each of the three coloured objects, we will define four policies, one for each complexity category (Unconstrained, Simple Bounds, Linear Constraints, Full Constraints). The same goes for the object `Soft constraints`: we will define four different policies that may implement it.

For example, the shell will not add the input constraints to the augmented Hessian itself. Instead, the shell will *delegate* the task of augmenting the Hessian to whatever policy is inserted in the blue box. Similarly, the blue box inherits from a constraint softening policy. Instead of adding the soft constraints to the augmented Hessian itself, the `Input Constraints` policy will delegate that task to whatever policy is inserted in the white box it inherits from.

Assume that the user has an MPC problem where the inputs are hard bounded. That requires that we insert the policy "Simple Bounds" in the blue box, and the policy "Unconstrained" in the white box it inherits from. In addition, assume that the user wants to have soft constraints for the states: simple bounds for the normal states, full constraints for the terminal state. Policies capable of managing that combination have to be chosen. The result is depicted in Figure 5.2.

Note that we use *italics* for the tags in the boxes. The reason is that these tags are not the type of the policy (as in Figure 5.1), but the concrete policy itself. Whereas Figure 5.1 depicts the generic inheritance diagram, Figure 5.2 shows a particular instance of the algorithm.

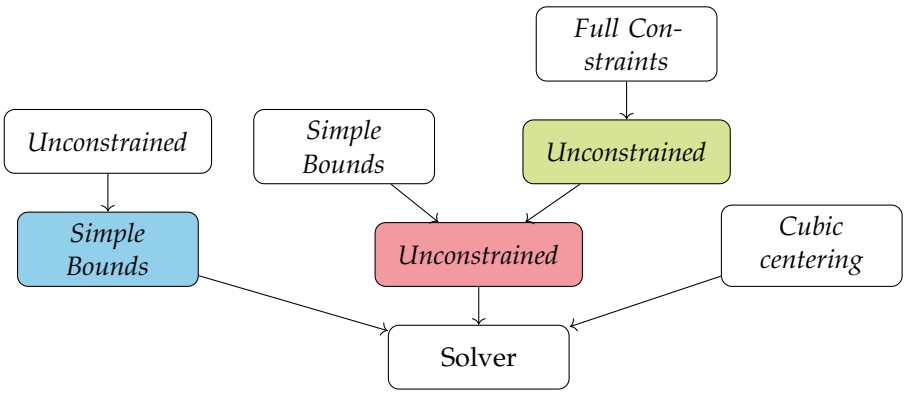


Figure 5.2: Inheritance diagram for our software.

The `Solver` in the bottom of Figure 5.2 implements the aforementioned MPC problem with no `if-else` comparisons and no code branching. And, at the same time, it exploit the particularities of each type of constraints for best and fastest execution. Note that we have included an additional parent object, called `Centering Policy` (Figure 5.1), so that we can choose between different heuristics for the centering parameter $\sigma$. In the example Figure 5.2, we use the cubic heuristics that Mehrotra introduced in his original paper.

By changing the policy that we insert in each box in Figure 5.1, we can customize the algorithm for each of the `6859` possible MPC formulations. Because the resulting piece of software is re-compiled every time that a policy is removed or added to the solver, the branching occurs at compile time, and hence no `if-else` branching occurs at run time.

We chose the programming language `C++` for our implementation. There are numerous reasons for this choice. In brief, `C++` is well-suited for embedded applications and allows to a very fast execution of the code (if not the fastest among all programming languages). In contrast with `C`, which was the other obvious choice, `C++` is much more appropriate for implementing advanced Computer Science principles such as Policy-Based Design.

In `C++`, our solver is coded as:

```
1 template <class InputPolicy, class StatePolicy,
2           class CenteringPolicy>
3 class Mehrotra : public InputPolicy, StatePolicy,
4                           CenteringPolicy {
5     // General architecture of the solver
6 };
```

Code 1: Policy-Based Design in our software.

Where `Mehrotra` is the name that we give to our software, due to the base algorithm that we use.

The three `class`'es in angular brackets are the policies the solver is templated against. Notice that these policies are nor concrete ones (such as "Unconstrained" or "Cubic Centering"), but abstract, templated ones. Because they are templated, they can resolve to any policy of our choice at compile time (see Appendix A for an introduction to templates in `C++`).

Last, note that the architecture in Code 1 does not faithfully represent the inheritance diagram in Figure 5.1. The policies handling the terminal and the soft constraints are not present. The example has the objective of illustrating the PBD principle.

## 5.4 Our Architecture

In our design, we use the inheritance diagram depicted in Figure 5.1. The policies are built with policies themselves (*e.g.*, `InputCons` inherits from `SoftCons`). For that reason, the shell is not templated against policies, but against templated policies. We will explain this in brief.

Before, we should note that we exploit generic programming beyond the scope of Policy Based Design. In the first place, we template the solver against a generic *floating value type*, that allows us to use the solver with different precision (*e.g.*, we can choose between single (32 bits) and double (64 bits) precision). Moreover, the solver is also templated against an *integer value type*, that allows to customize the type that we want to use for integer numbers.

Finally, we define two very simple matrix classes: `DiagMat`, for diagonal matrices, and `PDMat`, for positive definite matrices. These light-weight classes have the necessary information about the matrix dimensions and the data, and a few basic operations such as getters

and setters. We use these classes to differentiate the matrices we use in our algorithm. Because there are three different penalty matrices in our MPC problem, the solver is templated againts three generic *matrix types*, that the user can customize to either diagonal or positive definite matrix.

In code, the solver class is declared as:

```
template <class real_t, class int_t,
          template <class,class> class R_t,
          template <class,class> class Q_t,
          template <class,class> class P_t,
          template <class,class> class InputPolicy,
          template <class,class> class StatePolicy,
          template <class,class> class CenteringPolicy>
class Mehrotra : public InputPolicy<real_t,int_t>,
                         StatePolicy<real_t,int_t>,
                         CenteringPolicy<real_t,int_t> {
    // General architecture of the solver
};
```

Code 2: Declaration of the solver object.

In Code 2, note that some template arguments are templated themselves. In fact, they are templated against an integer value type and a real value type. This way, all the matrix light-weight classes and the policies can be customized for the integer and real value types chosen for the solver.

The policies `InputPolicy` and `StatePolicy` require a more detailed explanation. First, we shall look at the declaration of some input policies (Code 3).

In Code 3, we report the declaration of the policies that handle the cases where there are no hard constraints or the inputs, or these constraints are simple bounds. The policies `LinearCons` and `FullCons` are declared analogously. Note that they inherit from a policy capable of handling the soft constraints, as we depicted in Figure 5.1.

However, note that the template argument `InputPolicy` in Code 2 is templated against two type parameters (`real_t` and `int_t`). In constrast, the classes defined in Code 3 are templated against four arguments. To circumvent that, we use helpers as in Code 4.

Assume that we have a problem with hard bounds on the inputs, soft linear constraints on them and diagonal penalty matrices on the

```
namespace InputCons{

template <class real_t, class int_t,
          template <class,class> class R_t,
          template <class,class> class SoftCons>
class Unconstrained : public SoftCons<real_t,int_t> {
    // General architecture of the input policy
    // that handles the unconstrained scenario
};

template <class real_t, class int_t,
          template <class,class> class R_t,
          template <class,class> class SoftCons>
class SimpleBounds : public SoftCons<real_t,int_t> {
    // General architecture of the input policy
    // that handles the scenario of simple hard bounds
};

} // End of namespace
```

Code 3: Declaration of some input policies.

inputs. Then, we create the customized solver as in Code 4. In that piece of code, the dots substitute the other template arguments. We are not interested in them here, since we do not need them to show how helpers are employed.

```
template <class real_t, class int_t>
using InputHelper = InputCons::SimpleBounds<real_t,int_t,
                        matrix::DiagMat,SoftCons::LinearCons>;

// Create our customized solver
Mehrotra<double,int,...,InputHelper,...> MySolver();
```

Code 4: Usage of helpers.

We implement the polcicy `StatePolicy` in a similar way. The reader can check the details in the source code.

In Algorithms 1, 2 and 3, the shell delegates the execution of the algorithm to the appropriate policies. The explicit inclusion of policies results in Algorithm 4.

**Data:** $x$, $u$, $\nu$, $\lambda$, $s$, $x_0$
**Result:** $\Delta\nu$, $\Delta x$, $\Delta u$, $\Delta\lambda$, $\Delta s$
**Initialization:** $x_0$

**for** *k=1,2,...T* **do**
  InputPolicy: : fs$(L_{k,k}, L_{k,k-1}, \nu_k, \Delta\nu_k, k)$
  StatePolicy: : fs$(L_{k,k}, L_{k,k-1}, \nu_k, \Delta\nu_k, k)$
  $\Delta\nu_k \leftarrow L_{k,k}\backslash(\Delta\nu_k - M_{k,k-1}\Delta\nu_{k-1})$ `/* Forward solve */`
**end**

**for** *k=T,T-1,...1* **do**
  $\Delta\nu_k \leftarrow L_{k,k}\top\backslash(\Delta\nu_k - M_{k+1,k}\top\Delta\nu_{k+1})$ `/* Backsolve */`
  InputPolicy: : bs$(\nu_k, \Delta\nu_k, k)$
  StatePolicy: : bs$(\nu_k, \Delta\nu_k, k)$
**end**

**Algorithm 4:** Execution is delegated to the appropriate policies.

In Algorithm 4, $L_{k,k}$ and $L_{k,k-1}$ represent the $k$-th block in the diagonal and subdiagonal of the Cholesky factor of the Shur Complement (4.21) - (4.23), respectively. Note that, once the factorization is available, the affine vector $\Delta\nu$ is calculated in the shell via forward- and backwards substitution.

However, forming and factorizing the Shur Complement depends on the type of constraints and penalties. For that reason, the shell forwards to the policies `InputPolicy` and `StatePolicy` the responsability of forming and factoring the Shur Complement. In the first loop, the method `fs`$(\cdot)$ (forward solve) is called for the input and state policies. Depending on the policy, the Shur Complement is handled in one way or another. The same holds for the method `bs`$(\cdot)$ (backwards solve) in the second loop.

## 5.5 Linear Algebra

Our algorithm does not need many sofisticated linear algebra operations (such as eigenvalue or singular value decomposition). However, there are some expensive matrix operations that we need to perform. Among the operations with complexity $\mathcal{O}(n^3)$, we make use of the

Cholesky decomposition and the forward and backwards substitution of a triangular matrix versus a multiple right-hand side. Additionally, we use many typical $\mathcal{O}(n^2)$ and $\mathcal{O}(n)$ complexity operations, such as rank-1 updates, linear transformations, vector additions and the vector dot product.

To implement the necessary basic liner algebra operation, we follow the principles of generic programming and polymorphism that inspire this thesis. We create a generic class, `MatOps`, that is templated against a real value type (Code 5). That class wraps the linear algebra operations that we need. For example, if the dot product is defined for the real type that we are working with, we can use the dot product as in Code 6.

```
1  template <class real_t>
2  class MatOps;
3
4  // Define the specialization for double
5  template <> class MatOps<double>{
6    /* Define the required linear algebra operations here
7    *
8    *
9    */
10
11   /* Define the dot product r = x'y */
12   static double dotproduct(const int n, const real_t* x,
13   const real_t* y){
14       // Implementation of the dot product operation
15   }
16 };
```

Code 5: `MatOps` class declaration.

The benefits from this framework are dual.

The first benefit is that the framework allows to customize the linear algebra operations for any data type that the user wishes to have. In our software prototype, we provide support for single and double precision arithmetic (*i.e.*, `float` and `double` in `C++`). However, the user can easily expand the library to his or her customized data type by writing the corresponding template specialization, as in Code 5. We can think, for example, on implementing fixed precision arithmetic operations for embedded devices without floating point support. The modularity of the design allows for this kind of expansion at a low marginal cost.

```
1  int main(){
2    int n{3};                          // Vector dimension
3    double[3] x{1,2,3}, y{2,4,6};      // Vectors
4    double r;
5
6    /* Evaluate the dot product of x and y and store the result
7    in r */
8    r = MatOps<double>(n,x,y);
9
10   return 0;
11 }
```

Code 6: Usage of the `MatOps` wrapper.

The second benefit is that this framework allows to define the linear algebra operations in different ways. Different implementations may be advantageous in different scenarios. In our software prototype, we make use of open source BLAS[2] ans LAPACK[3] libraries. In particular, we use OPENBLAS [27], which is an optimized BLAS and LAPACK library. However, the user may prefer to use other libraries (*e.g.*, specialized BLAS libraries for Intel microprocessors). In that case, he or she may replace the calls to OPENBLAS by calls to his or her customized library.

Moreover, it may be the case that we need to avoid external libraries to reduce the memory footprint of the final program. In that case, we may as well define our own linear algebra operations, which may be less efficient but will also result in a smaller memory footprint.

## 5.6 Memory Allocation

Last, we would like to talk about memory allocation in our software. In Computer Science, we speak of *static* memory allocation when the memory that the variables need is reserved when the program starts. Note that, in turn, static memory allocation requires that the size of all variables is known when the program is compiled. As opposed to it, we speak of *dynamic* memory allocation when the memory is allocated at runtime (normally, whenever it is needed, but not before that). We

[2]Basic Linear Algebra Subprograms
[3]Linear Algebra Package

usually resort to dynamic memory allocation whenever, when the program starts, we do not know yet how much memory the variables will need.

The natural advantages of dynamic memory allocation is that it prevents memory wastage. Memory wastage arises in the scope of static memory allocation, when the memory required by our variables is uncertain and we reserve extra memory space to be on the safe side. For example: if we know that some input matrix will range from a $2 \times 2$ to a $200 \times 200$ matrix, we reserve memory for the latter. This way, we guarantee that we always have enough memory to run our program. However, if the input matrix is smaller (say, a $2 \times 2$ matrix), most of the allocated memory will not be used. Naturally, we can easily circumvent this disadvantage if we dynamically allocate the exact required memory at run time, once that we know the size of the matrix that we are going to read.

Another way of preventing this problem is to modify the source code for each problem instance, customizing it for the particular size of the input data. However, this approach requires systematically modifying and compiling the source code, which is time-consuming and definitely less convenient than letting the program allocate the memory at run time (without touching the source code).

For those reasons, many application prefer dynamic memory allocation. In particular, think about code libraries. Libraries are pieces of compiled code, and hence dynamic memory allocation is the only option in every case but if we can tightly estimate the memory consumption of the user.

In our software prototype, we rely on dynamic memory allocation for the reasons described above. However, static memory allocation has a few advantages that, though irrelevant in many applications, are crucial in some others.

The main disadvantages of dynamic memory allocation are that you can run out of memory, and that memory corruption can occur, during the execution of the program. Both problems may arise at runtime, which is not admissible in most embedded, real-time applications. Moreover, most safety-critical applications prohibit the usage of dynamic memory allocation.

Our current design uses dynamic memory allocation. However, future expansions of the software will allow the user to choose from static and dynamic memory allocation.

# Chapter 6

# Numerical Experiments

In this chapter, we report some performance measurements of the designed software. For benchmarking purposes we will compare against qpOASES [28], a parametric active set solver, and Gurobi, a general purpose comercial solver. Our numerical experiments put emphasis in two main characteristics of optimization solvers: effectivity and efficiency.

We measure the effectivity of our software as the accuracy of the solutions it provides. To do so, we compare the solution reported by our solver with the solution reported by Gurobi. The reason for this is that Gurobi is solid, well-test commercial package.

Efficiency refers to the time that the software requires to solve a given optimization problem (to a given accuracy). Here, the choice is qpOASES for several reasons. First, it is an active set solver, and hence allows to compare against a completely different algorithmic approach. Second, qpOASES is one of the best accepted open-source QP solvers in the optimization community. It is common to see qpOASES in the benchmark of any new optimization software (see *e.g.*the excellent benchmarking in [13]). And, finally, qpOASES is open source and hence we can easily make use of it.

## 6.1 Benchmarking Problem: the Oscillating Masses

Our focus in in the optimization problems that arise from an MPC structure. Hence, it would have been possible to use randomly generated MPCs. However, in our benchmarking we will apply our optimization

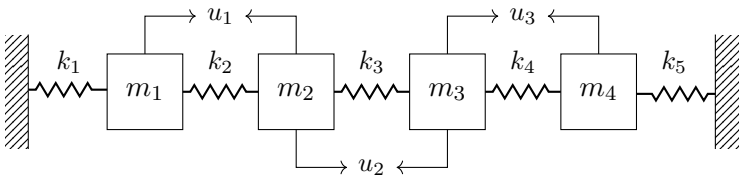


Figure 6.1: Oscillating masses system

solver to a real MPC problem. The control problem that we are attempting to solve is a mechanical system of masses interconnected by springs, as depicted in Figure 6.1. This setup is a popular benchmarking problem for optimization in the scope of Model Predictive Control (see [13], [21], [22]).

In the system, there are $N$ masses, interconnected by $N + 1$ springs and connected to the walls at both ends of the chain. In the resting position of the masses, the springs exert no forces upon any of them. The position of the $i$-th mass is denoted by $x_i(t)$, where $x_i(t) = 0$ indicates that the mass lays on its resting position.

The control inputs are external forces that act upon the masses, and we add them as an independent term to the dynamic equations of the masses upon which they act. If we denote the inputs by $u$, the dynamic equations for the system depicted in Figure 6.1 are:

$$m_1\ddot{x}_1 = -k_1x_1 + k_2(x_2 - x_1) + u_1, \tag{6.1}$$
$$m_2\ddot{x}_2 = -k_2(x_2 - x_1) + k_3(x_3 - x_2) - u_1 + u_2, \tag{6.2}$$
$$m_3\ddot{x}_3 = -k_3(x_3 - x_2) + k_4(x_4 - x_3) - u_2 + u_3, \tag{6.3}$$
$$m_3\ddot{x}_3 = -k_4(x_4 - x_3) - k_5x_4 - u_3, \tag{6.4}$$

where $m_i$ is the $i$-th mass, $k_i$ is the mechanical constant of spring $i$ and the dependency of $x_i$ with time has been omitted for clarity. Note that the system can be easily expanded to any number of masses.

To use this model in our MPC simulations, we have to derive a state-space model and discretize it. Details on this process are available in Appendix B. Here it suffices to mention that the resulting state-space model has as many inputs as actuators on the masses, and twice as many states as masses in the system (since both positions and velocities are incorporated to the state vector).

## 6.2 Test Procedure

In order to make the efficiency tests on an even basis, we enable the MPC option in qpOASES, which results in faster execution of the algorithms. In Gurobi, we do not use any special option, since we are not interested in computation speed but in the accuracy of the solution. We performed all the tests in C++. The reason for this is that both qpOASES and our solver source code is written in C++. In the case of Gurobi, we use the C++ wrapper provided with the package.

We have performed all the numerical experiments in a server with $1.2$ terabytes of RAM and $32$ Intel cores E5-2687W and $3.10$ GHz. We limited the computations to just one core. We use a sampling time of $0.5$ seconds and the simulations last for $500$ time steps. Additionally, we reproduced every simulation $50$ times, and the smallest solution time for every QP was recorded. This way, we attempt to eliminate the influence of other programs running on the computer. Finally, since we are interested in online MPC, where every optimization problem must be solved before the next sampling interval, the worst execution time for every simulation is reported.

In other words: every QP in the simulation is solved $50$ times, and the smallest solution time is recorded. Then, among all the QPs in the simulation, the worst computation time is reported. For completeness, we als report the median of the solution time, the number of required iterations and the accuracy of the solutions.

It is also important to mention that we include a random disturbance in our tests. To make sure that we generate the same QPs in every simulation, we always use the same *seed* for generating the random numbers.

## 6.3 Test 1: MPC Dimensions Influence

In this section, we study how the dimensions of the system under control (number of states $n$ and inputs $m$) and the prediction horizon $T$ influence the performance of the designed software. The base case is depicted in Figure 3:

It is a system with $N = 6$ masses, 7 springs and $5$ actuators, along with a prediction horizon of $20$ time steps. This produces a state-space model with $n = 12$ states and $m = 5$ inputs. All the masses in the

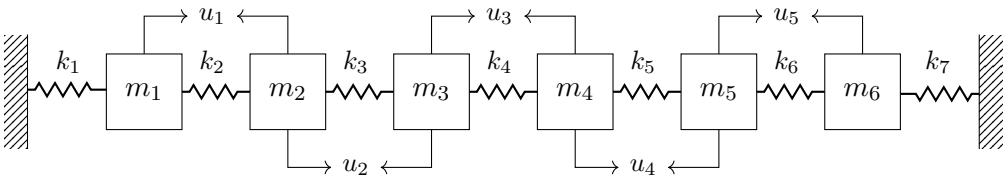


Figure 6.2: Base benchmarking problem

system weigth $1$kg, and all the spring constants are equal to $1$ kg/s.

In the initial state, all masses lay on their resting position with zero velocity. The goal is to control the system to the resting position of all the masses when random disturbances come in. To do so, the system is subject to some constraints on displacements, velocities and inputs. In particular, the restrictions are:

$$\begin{aligned} |x_i[k]| &\leq 1 && \forall i, k, \\ |v_i[k]| &\leq 1 && \forall i, k, \\ |u_i[k]| &\leq 0.5 && \forall i, k, \end{aligned}$$

where $|\cdot|$ is the absolute value operator.

The random disturbance $d \in \mathbb{R}^N$ acts upon every mass and has a uniform distribution $[-0.357, 0.357]$. We chose those limits such that none of the arising optimization problems becomes infeasible. Finally, the penalty matrices are chosen as $R = I$ and $Q = P = I$, where $I$ is the identity matrix of appropriate dimensions.

## Dimensions of System under Control

First, we will show how the system dimensions influence the performance of our software. To do so, we will solve the base case with a varying number of masses, ranging from $N = 4$ to $N = 20$. One actuator is added for every additional mass, such that there are always $m = N - 1$ inputs to the system (which guarantees controllability under the given disturbances). The results are depicted in Figure 6.3.

We would like to make a few remarks here.

In first place, we shall notice that the designed software and qpOASES are comparable regarding maximum compuation time, for all problem dimensions. The largest QP problem in the series has $20$ masses and $19$ actuators. When spawned along $T = 20$, it results in a QP with $800$ variables and $380$ inputs, all of which are lower and upper bounded.

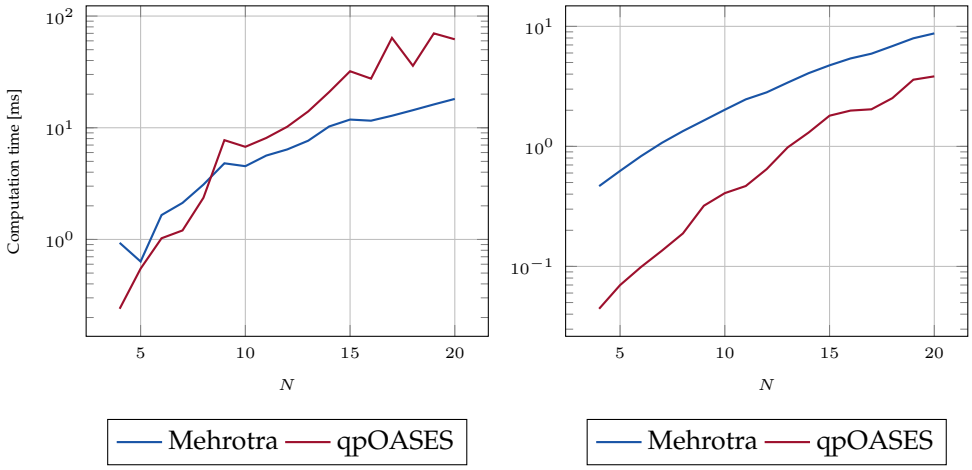


Figure 6.3: Maximum and mean computation time for varying number of masses

Even in that case, the computation time lies below $100$ ms, which is significantly smaller than the sampling time of $500$ ms.

In second place, qpOASES shows a clear advantage regarding the mean computation time. This is easy to understand by looking at Figure 6.4. qpOASES is an active set method and, hence, requires very few iterations when a good initial guess is available. Since we are simulating a linear system with a rather small disturbance, this is usually the case. qpOASES reports an average number of iterations below $1$ for most of the problems. Our interpretation is that, for the majority of the QPs in the simulation, the set of active constraints at the optimum does not change. Under such circumstances, any active set method does not require any iteration, since it converges to the optimum in exactly one Newton step. Hence, it is natural to expect a low mean computation time.

On the other hand, the iterations required by Mehrotra are rather constant for all problem dimensions. This is advantageous regarding the maximum number of iterations. There is an increased optimization times for large problems, but we explain it in terms of the computational effort required for the factorization of large matrices, as opposed to requiring more steps to converge. That is the case of qpOASES. However, the minimum iterations required by our software is rarely below $3$. One possible explanation is that interior point methods need to reach the central path before finding a new optimal value. Assume

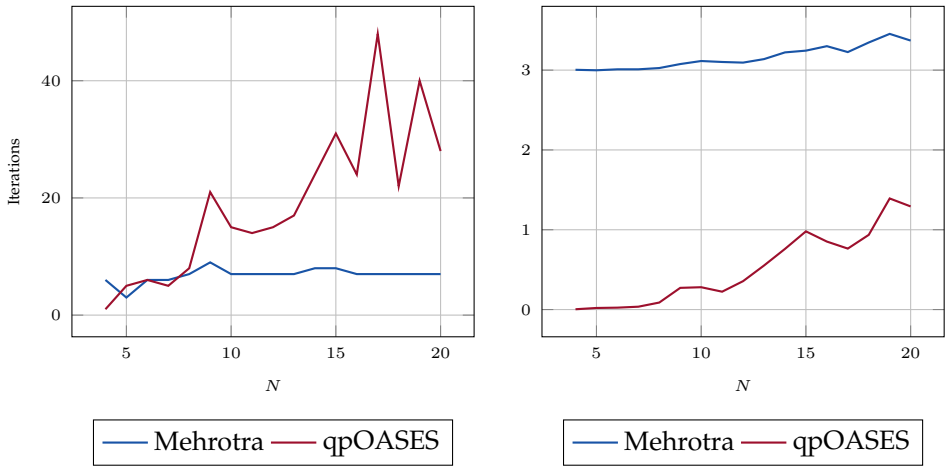


Figure 6.4: Maximum and mean iterations for varying number of masses

that the solutions of two consecutive QPs lie in the same set of active constraints, which is often the case. Whereas qpOASES converges in one Newton step, our solver needs to step aside the boundary, reach the central path and converge to the new optimum. The warmstart of interior point methods is still an open question, with few satisfactory approaches in the literature.

To sum up, it is important to remark that, for our purposes, the maximum computation time is what matters. Out goal is to solve MPC problems in real time and, hence, we need to solve the given optimization programs in a fixed, maximum time. However, it is worth to mention that a low mean computation time is probably advantageous in other aplications where the total computation time matters (*e.g.*in simulations).

## Dimensions of the Input Vector

Next, we show how the ratio of system inputs to system states affects the performance of our solver. For that purpose, we fix the number of masses to six, and the prediction horizon length to twenty time steps.

In addition, we create MPC problems with a varying number of actuators, ranging from $m = 3$ to $m = 11$ (*i.e.*the input space is always smaller than the state space). The first five actuators are the same as in the previous experiment. To reduce the number of inputs, we remove the left-most actuators; *i.e.*the actuator between masses $1$ and $2$, first,

and the one conected to masses $2$ and $3$, later. To increase the number of inputs, we add small actuators that act on a single mass.

These actuators are constrained to a $20\%$ of the main actuators; *i.e.*they are upper and lower bounded by $0.1$ kg.m/s$^2$. The first additional input acts upon mass $6$, the second one acts upon mass $5$, and so on.

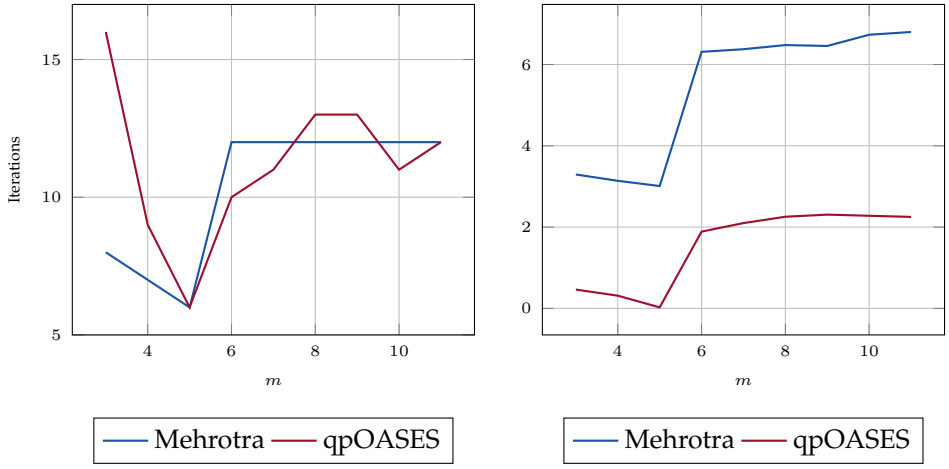


Figure 6.5: Maximum and mean iterations for varying number of inputs

These experiments are interesting because the show the consequences of choosing a dense of sparse MPC formulation. Recall from Chapter 3 that the dense MPC formulation results in a QP with $Tn$ variables and no equality constraints. In contrast, the sparse MPC results in a QP with $T(m+n)$ variables and additional equality constraints.

In theory, interior points methods tend to perform a small, rather constant number of expensive iterations when compared to active set methods. This agrees with the results we report in Figure 6.3 and Figure 6.4: whereas qpOASES performs a much larger maximum number of iterations, the maximum computation time is similar for qpOASES and Mehrotra.

In Figure 6.5 we observe something different. The maximum number of iterations is approximately equal for out interior point implementation and qpOASES. In theory, this should result in a much better performance of the active set method, since its iterations tend to be cheaper. However, Figure 6.6 shows that the maximum computation time is also approximately equal.

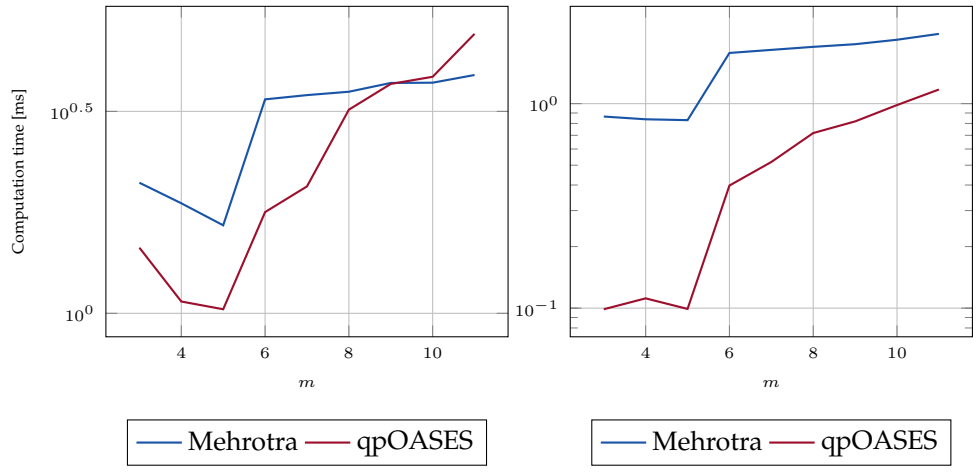


Figure 6.6: Maximum and mean computation time for varying number of inputs

The reason for this is that any algorithm that solve the MPC in dense formulation is specially sensitive to the input space dimension. A sparse MPC formulation is less sensitive to this parameter, as can be seen in Figure 6.6.

## Prediction Horizon Length

Finally, we will show how our software behaves with regard to the prediction horizon length. Note that a larger number of prediction steps may be due to a larger prediction time, but also due to a smaller sampling time. In our experiments, we fix the number of masses to six and the number of actuators to five (*i.e.*$N = 12$ states and $m = 5$ inputs). Then, we reproduce the simulations for prediction horizons that range from 10 to 100 steps.

The results depicted in Figure 6.7 agree with the theory. Whereas qpOASES has a better maximum computation time for smaller prediction horizons, our solver performs notably well under long prediction horizons. The maximum computation time for Mehrotra grows linearly with the prediction horizon, which is in accordance with the complexity analysis in Chapter 4. (Note that a linear function has the typical form of a logarithmic function when plotted against a logarithmic axis.)

Even more interesting, the maximum and mean number of iterations required by our solver is constant for all prediction horizons (Figure 6.8).

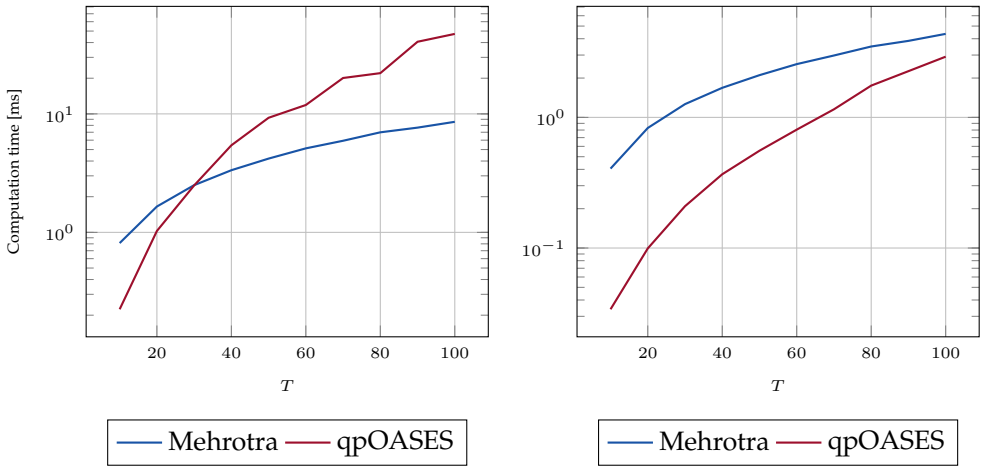


Figure 6.7: Maximum and mean computation time for varying prediction steps

This is one big advantage of interior point methods, at the cost of a high minimum number of iterations required.

## 6.4 Test 2: Accuracy of the Solution

The goal of this work is to solve the quadratic programs efficiently, but not at the expense of obtaining a correct solution. From our simulations, we can conclude that our software provides rather accurate optimal values, since the system is controlled succesfully. However, quantifying the accuracy of the solution is not a simple task. Furthermore, we noticed in the course of our experiments that there was some discrepancy between the solutions provided by Mehrotra and qpOASES to a series of critical QPs.

We use a third optimization software, Gurobi, to check the correctness of the optimal values and variables. To do so, we reproduced the same batch of experiments from Test 1, using the same seeds for the random number generator such that the results are comparable. Also, we assume in this section that Gurobi provides the correct solution, and thus we compare against it.

In our software, we set both the *feasibility tolerance* and the *optimality tolerance* to $1e-3$, for all performance and accuracy tests. Recall that the feasibility tolerance is the minimum treshold for the norm of the

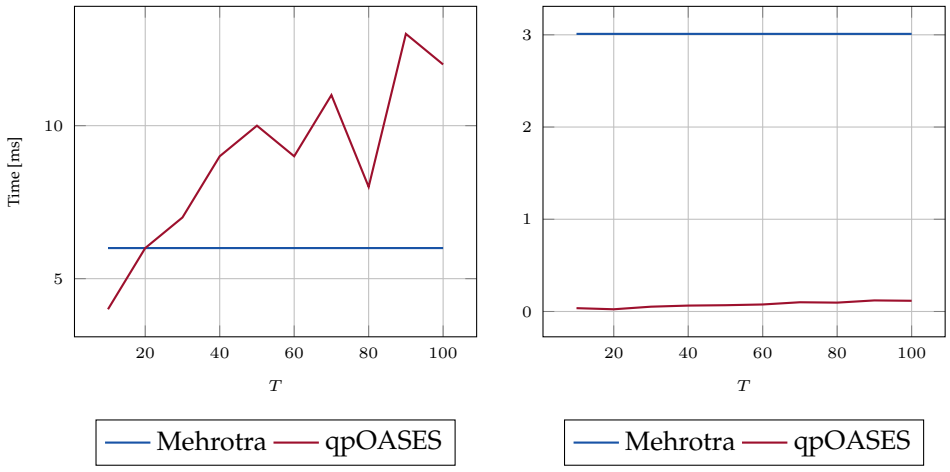


Figure 6.8: Maximum and mean iterations for varying prediction steps

equality residual (4.8). That is: we assume that the solution is feasible with respect to the equality constraints if the norm of $r_e$ is smaller thatn the feasibility tolerance. Similarly, the optimality tolerance is a minimum treshold for the duality gap.

In our accuracy tests, we vary the number of states, inputs and prediction steps as we did in the previous section. In Figure 6.9 we show the maximum offset between the optimal value (provided by Gurobi) and the solutions of Mehrotra and qpOASES for a varying number of masses. To do so, we have simulated the system for $500$ steps and report here the maximum and the mean offsets (in percentage). From now on, *offset* refers to the quantity $f(x) - f(x^*)$, where $f(x^*)$ is the optimal value as reported by Gurobi and $f(x)$ is the optimal value of the solver under consideration (Mehrotra, qpOASES).

First of all, we want to remark the accuracy of the solutions computed by Mehrotra. Though the blue line in the plot is hard to read, the maximum error incurred in by Mehrotra lies below $0.01\%$, which is satisfactory.

Second, it is important to remark that qpOASES also does a good job when it comes to the correctness of the solution. The mean error, when compared with Gurobi, lies below $0.06\%$ in our numerical experiments. However, we can observe that, for some critical QPs in the simulations, the error commited by qpOASES reaches values above $100\%$.

In fact, qpOASES provides reports whether a particular QP has

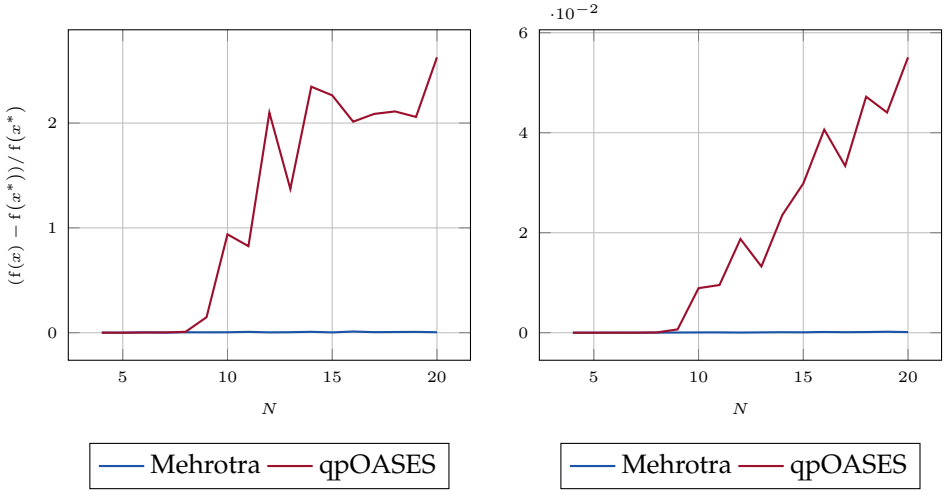


Figure 6.9: Maximum and mean error in the optimal value when compared against Gurobi.

been solved or not. Based on that information, we have elaborated Table 6.4. We report there how often qpOASES failed to solve a QPs in the simulation series. These data are important because, as we can read from the qpOASES reference manual, enabling the MPC flag sacrifices accuracy and robustness in exchange for performance. Hence, it is even more remarkable that Mehrotra provides a fast solution to the arising QPs, yet does not give up on accuracy on exchange.

To clarify the meaning of these numbers, we also provide the evolution of the optimal values for all the QPs in the problem scenario with $20$ masses. For most of the time, the optimal value reported by the three solvers is indistinguishable from one another. Nonetheless, for certain, critical QPs, qpOASES clearly fails at computing the optimal value. Though we did not deepen in this analysis, this may suggest that employing qpOASES as an optimization backend could lead this particular system to instability.

We carried out the same correctness verification for the other two batches of numerical experiments: varying number of inputs, and varying number of the prediction horizon length. Since the results are less relevant, we will comment them briefly.

In Figure 6.12 we can observe that both Mehrotra and qpOASES provide high quality solutions to all the QPs in the series of varying number of inputs. In particular, the maximum error when compared against Gurobi lies always below $0.5\%$. Notice that, though qpOASES

| $N$ | % unsolved QPs |
|---|---|
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| 9 | 0 |
| 10 | 2.2 |
| 11 | 1.8 |
| 12 | 1.8 |
| 13 | 1.8 |
| 14 | 2.0 |
| 15 | 3.2 |
| 16 | 4.6 |
| 17 | 3.4 |
| 18 | 6.2 |
| 19 | 6.4 |
| 20 | 6.6 |

Table 6.1: Percentage of unsolved QPs in the simulations corresponding to different number of masses.

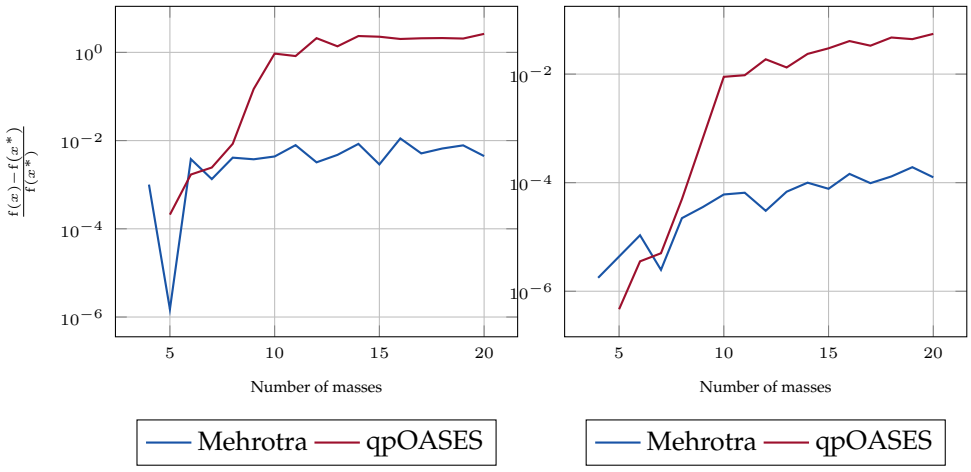


Figure 6.10: Maximum and mean error in the optimal value when compared against Gurobi. (Log version)

proves to be more accurate than our software in this particular batch of problems, the difference is way less significant than in the previous series.

Finally, we report the correctness of our solver for problems of different prediction steps. Figure 6.13 shows a similar error treshold to the previous one (Figure 6.12). Hence, we can conclude that our software is capable of performing efficient optimization procedures while keeping a high degree of accuracy. While qpOASES can certainly outperform our solver in some cases (mainly during steady-state operation), it does not guarantee the correctness of the solution to the same extent that we do. Finally, it is worth to remark that qpOASES did solve all the QPs in the two last problem batches.

## 6.5 Test 3: Constraint Softening Implementation

In this test, we report the impact that constraint softening has in the performance of out method. We implement soft constraints as we explained in Chapter 4.

Here, we solve the exact same batch of problems that we solved in Test 1, but we let the constraints be soft. Naturally, the soft constraints are never violated, since otherwise some of the QPs in Test 1 would

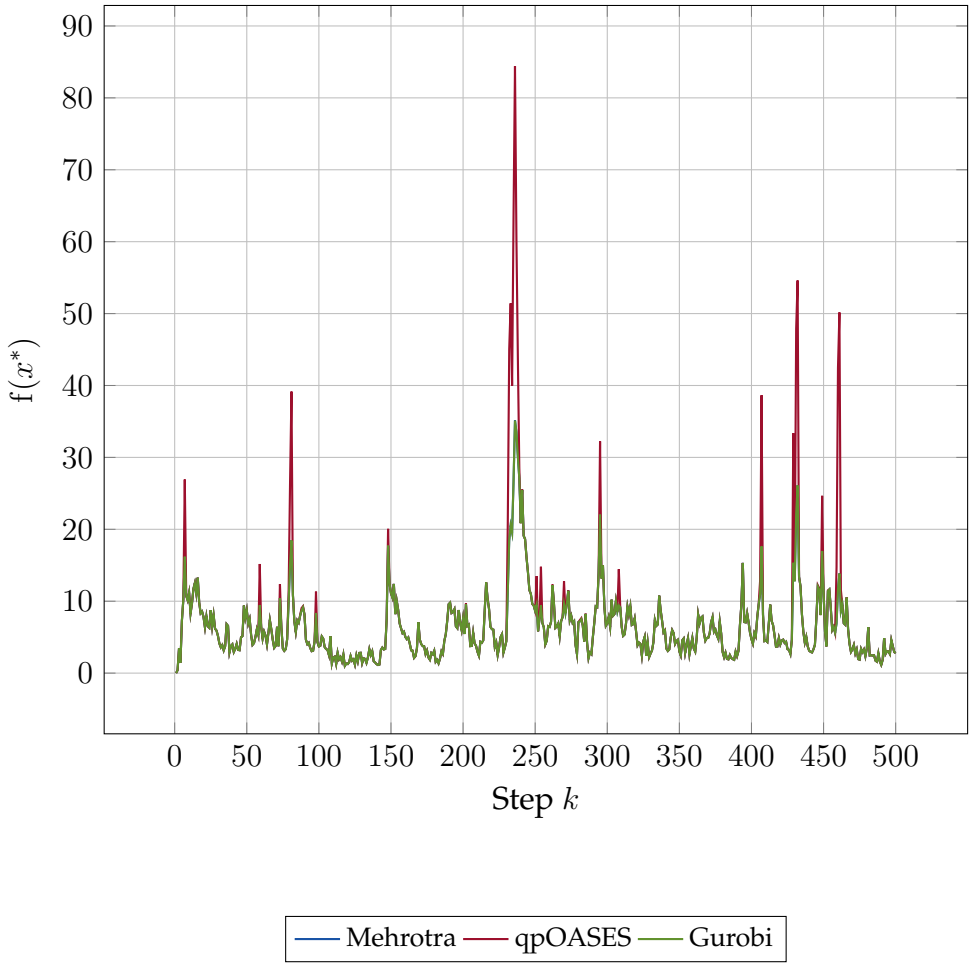


Figure 6.11: Optimal value for each simulation step, as reported by the Mehrotra, qpOASES and Gurobi.

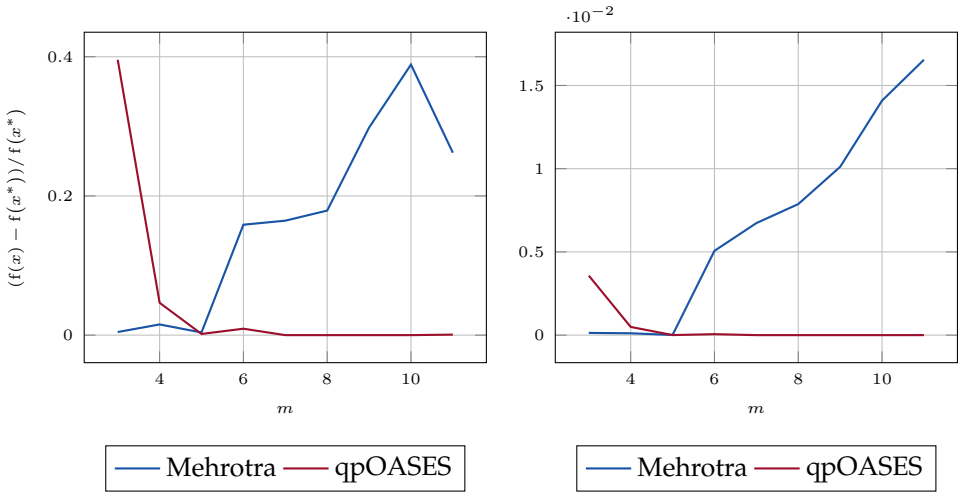


Figure 6.12: Maximum and mean error in the optimal value when compared against Gurobi. Varying number of inputs.

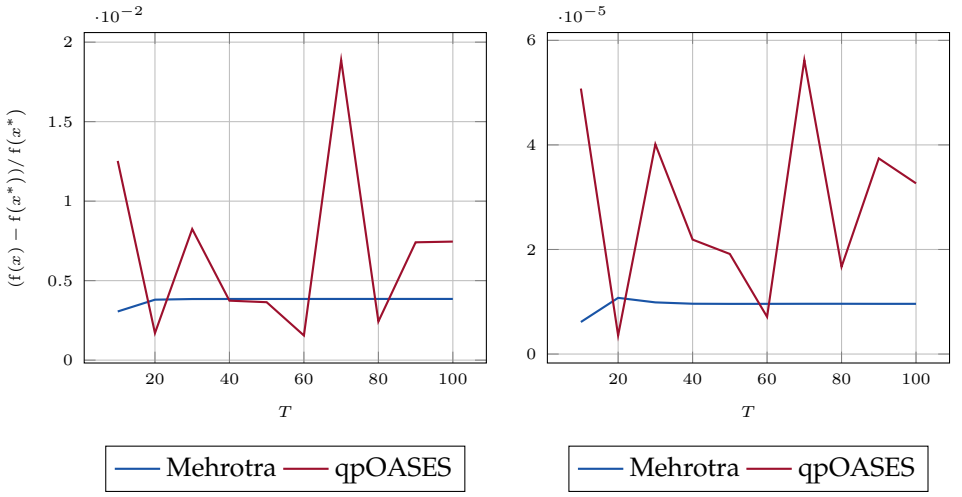


Figure 6.13: Maximum and mean error in the optimal value when compared against Gurobi. Varying number of prediction steps.

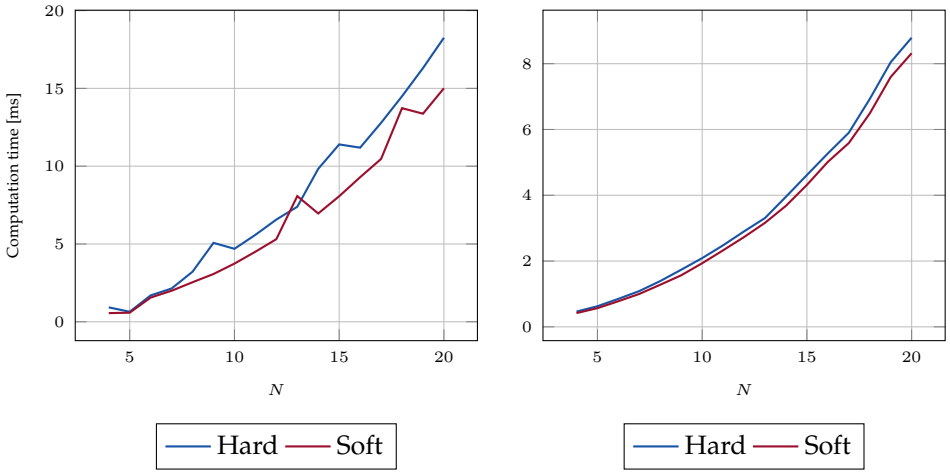


Figure 6.14: Maximum and mean computation time when solving the problem with hard and soft constraints, for different number of masses.

have been infeasible. For a comparison of our soft constraints method versus the standard method, we refer the reader to Test 4 and Test 5.

Figures 6.14 and 6.16 are interesting, since they show an *a priori* counter-intuitive result. The exact same problem is solved faster if soft constraints are enabled. The difference is not great, but certainly the addition of soft constraints does not degradate the performace.

We have a possible explanation for this behaviour. Whether we use hard or soft constraints, the search direction is computed at the exact same cost. Moreover, during the simulations we rarely hit the state constraints, whereas we hit the input constraints quite often. Hence, the linesearch also yields the same results.

Furthermore, we save one linesearch, since we do not take into account any hard constraints. Furthermore, we do not have to calculate slack variables and lagrange multipliers associated with the hard constraints. These are minor aspects of the algorithm, but they do contribute to explain this unexpected result.

Another question that we should ask ourselves is: if soft constraints reduce the computational burden, shall we just soften all the constraints? There are two main disadvantages in that approach.

First, it may be, depending on the penalties, that the optimal solution consists in violating a soft constraints, even if a feasible solution is available. Then, the feasible solution is suboptimal and we disregard it. Whereas this is acceptable for state constraints, it would be very

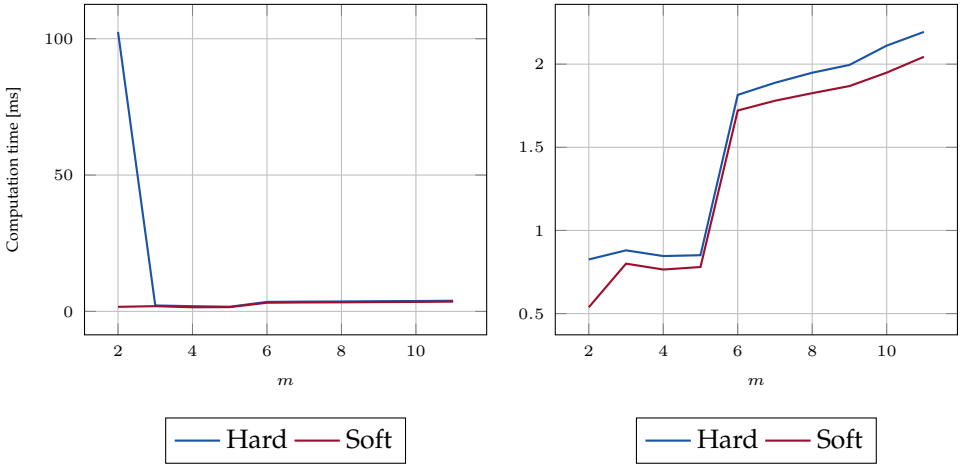


Figure 6.15: Maximum and mean computation time when solving the problem with hard and soft constraints, for different number of inputs.

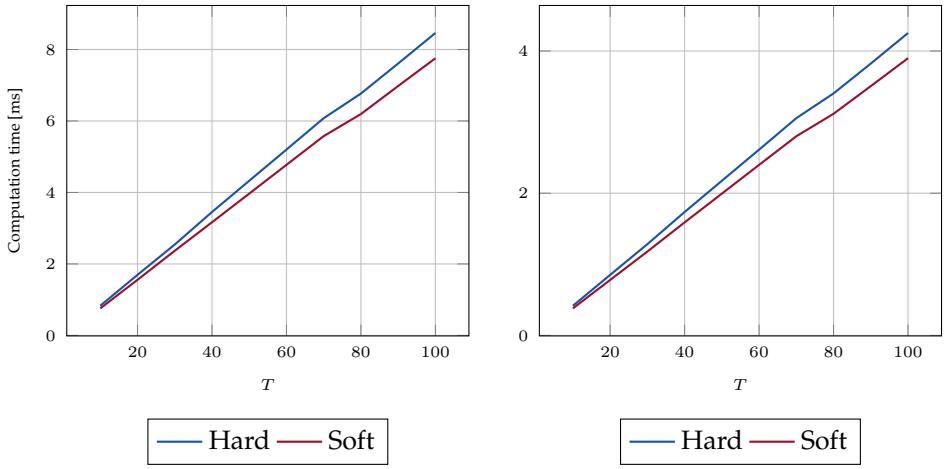


Figure 6.16: Maximum and mean computation time when solving the problem with hard and soft constraints, for different number of prediction steps.

dangerous in the case of input constraints. (Recall that state constraints are normally control goals, but input constraints relate to physical limitations of the system.)

Second, we lose information about the problem whenever we substitute hard constraints by soft constraints. For instance: if all constraints were soft, we could not perform exact linesearch. Instead, we should rely on some backtracking linesearch [21] and the application of the Mehrotra algorithm is unclear. We did not investigate this direction, but we shall remark that choosing full constraint softening is not as advisable as Figures 6.14 and 6.16 may suggest.

Finally, we would like to pull the attention to Figure 6.15. Notice that we included an additional simulation with respect to Figure 6.6, the one corresponding to $m = 2$. We did so to illustrate the meaning of constraint softening. In the simulation with $N = 6$ masses and $m = 2$ inputs, the optimization problem becomes infeasible at some simulation step. In that case, the hard-constrained solver fails to provide a solution and iterates until reaching the maximum iterate count.

For the rest of the proposed systems (*i.e.*, for $m$ ranging from $3$ to $11$), the results are in accordance with the previous discussion.

## 6.6 Test 4: Constraint Softening Performance

Next, we raise the disturbance to the interval $[-1, 1]$, which guarantees that some of the QPs in the series are infeasible. For this setup, we include qpOASES in the analysis.

We want to warn the reader against the meaning of the following results. We will observe in the graphs that qpOASES is up to two orders of magnitude slower than our software. However, this is not really the case. It is the fact that qpOASES fails to converge in some of these problems (which are feasible by definition). In those cases, it iterates until reaching the maximum iteration count, which we set to $1000$, and hence the solve times that are meaningless.

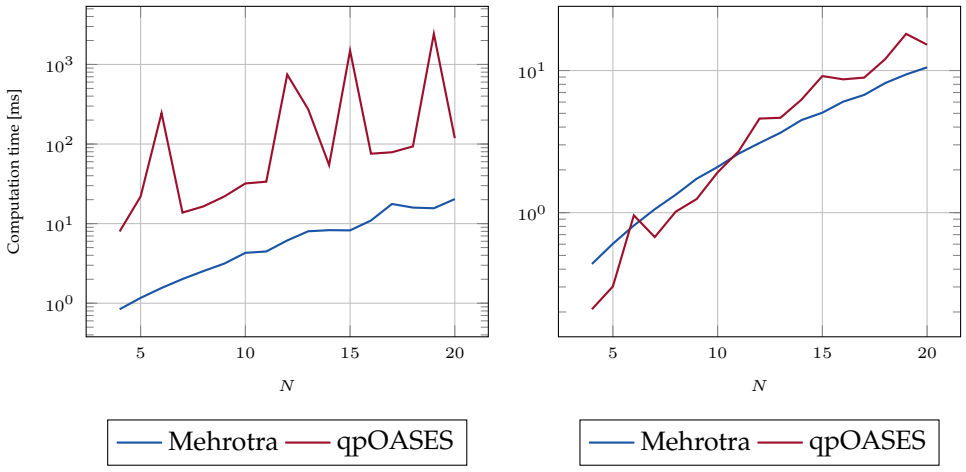


Figure 6.17: Maximum and mean computation time for soft constrained problems, for a varying number of masses.

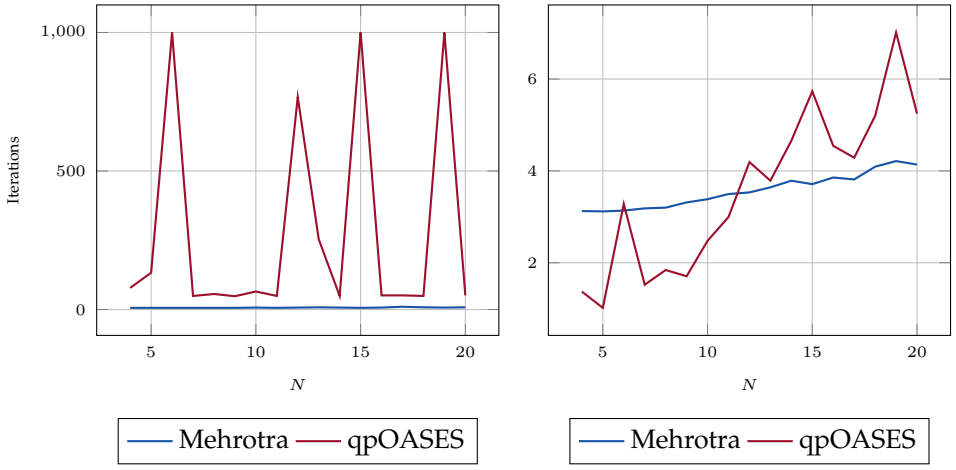


Figure 6.18: Maximum and mean number of iterations for soft constrained problems, for a varying number of masses.

Consider the performance of the solver for a varying number of masses, which is depiceted in Figures 6.17 and 6.18. First, we can observe the peaks that we just mentioned: in some series, qpOASES encounters a QP problem that it cannot solve (*i.e.*, the series for $N = 6$, $12$, $15$ and $19$). In those cases, the iteration count raises up to $1000$ (Figure 6.18), which is reflected in the maximum computation time (Figure 6.17).

Instead, we would like to focus on the other cases. Notably, our

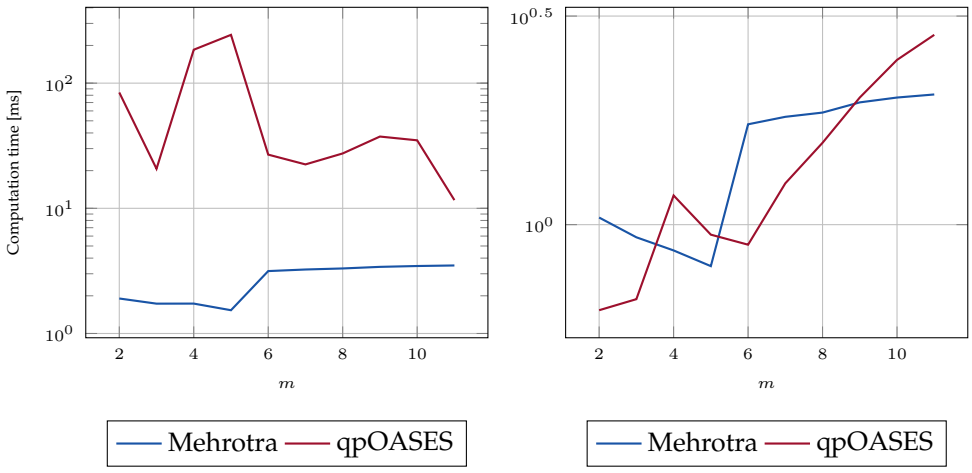


Figure 6.19: Maximum and mean computation time for soft constrained problems, for a varying number of inputs.

software outperforms qpOASES by one order of magnitude regarding maximum computation time. Similarly, the maximum and mean number of iteration rapidly grows with respect to the hard constrained experiment. All these results speak in favour of our software.

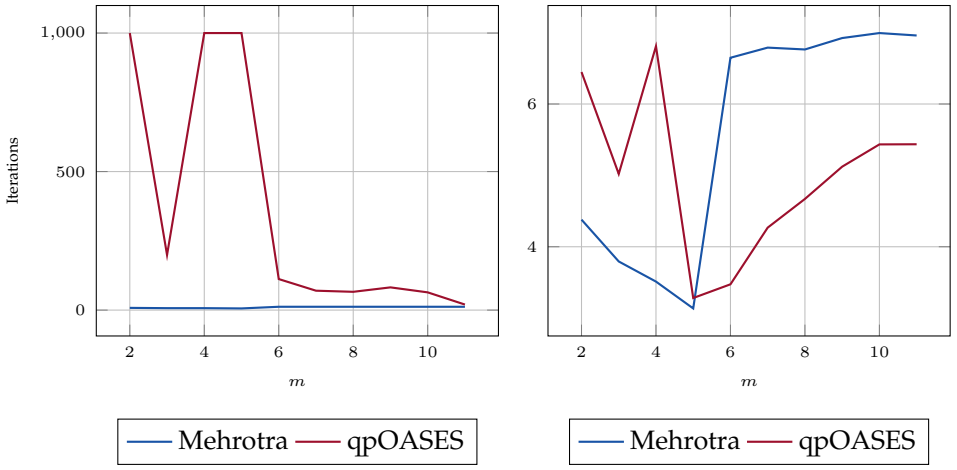


Figure 6.20: Maximum and mean number of iterations for soft constrained problems, for a varying number of inputs.

The results for different number of inputs and prediction steps are

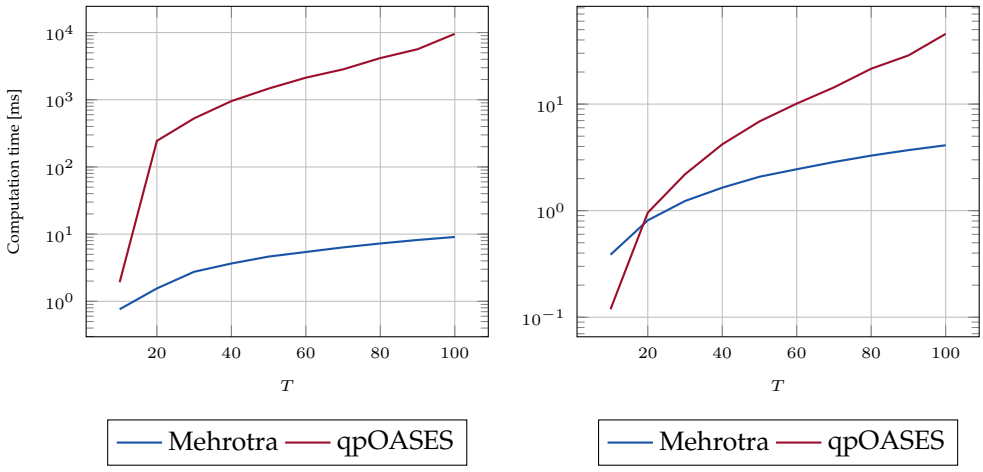


Figure 6.21: Maximum and mean computation time for soft constrained problems, for a varying number of prediction steps.

also reported for completeness. However, the previous analysis still holds without any modification.

Interestingly, qpOASES systematically fails for all prediction horizons larger than $T = 20$. That is: from the $500$ QPs solved during the simulation, there is always at least one that qpOASES cannot solve within the maximum number of iterations.

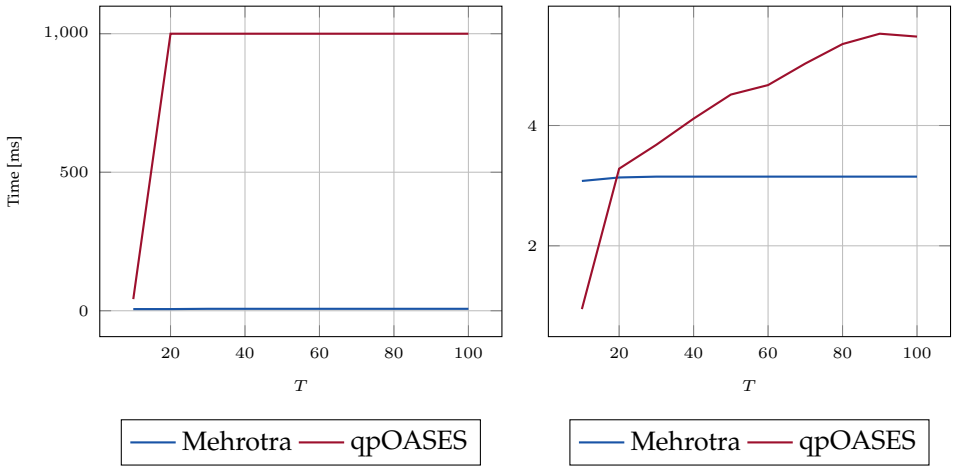


Figure 6.22: Maximum and mean number of iterations for soft constrained problems, for a varying number of prediction steps.

## 6.7 Test 5: Accuracy of Constraint Softening

Finally, one open question remains. We know that qpOASES fails at solving some of the problems with softened constraints. However, is our software solving them correctly? To answer this question, we compare against Gurobi as we did in Test 2.

The setup is as follows. We solve the same batch of problems that we solved in Test 4. Instead of running every simulation $50$ times, we solve them only once and we do not measure times, since we are not interested in performance. We feed the exact same dense QP to qpOASES and Gurobi. For Mehrotra, we use or particular constraint softening implementation.

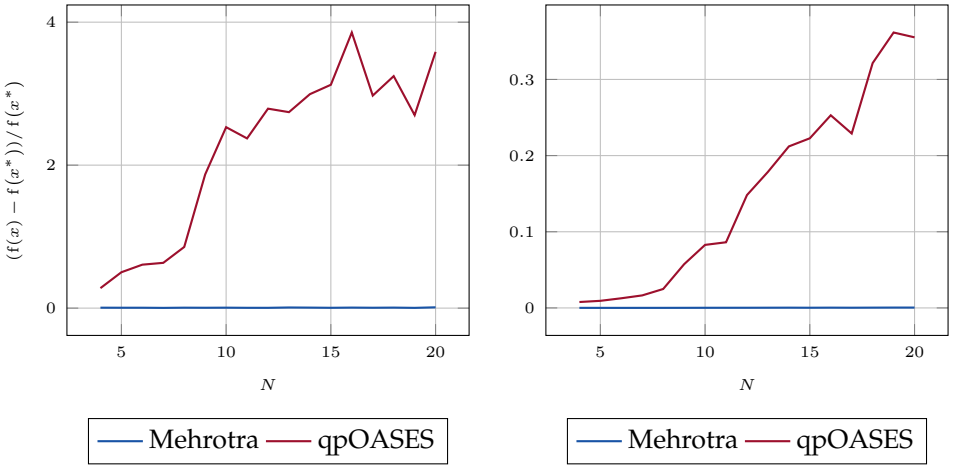


Figure 6.23: Accuracy of the solutions for the soft-constrained problem, for varying number of masses.

In Figures 6.23, 6.24 and 6.25, we show that maximum error of Mehrotra and qpOASES at computing the optimal value, when compared against Gurobi. We can observe that our solver always provides rather accurate solutions, with the maximum error always below $1\%$. In contrast, qpOASES exhibits notable accuracy errors.

Note that even the mean error is large, going above $10\%$ for $N > 12$. This time, the maximum errors reaches values has high as $200\%$, which is probably inacceptable for most applications. There is one outsider in Figure 6.24, the one corresponding to $6$ masses and $4$ inputs. The worst case for qpOASES in that scenario is an optimal value which is

100 times larger than the optimal one. The mean error goes up to 140% in that simulation, which means that qpOASES systematically fails at solving the optimization problems that we randomly generate.

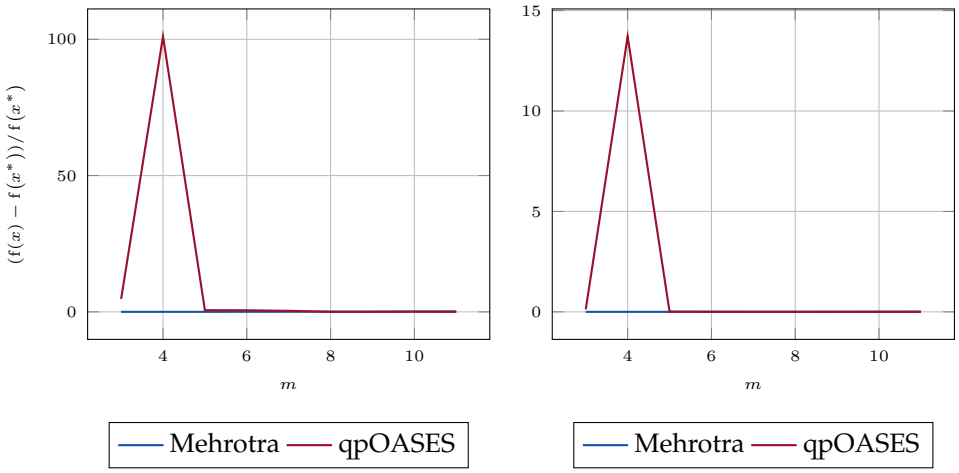


Figure 6.24: Accuracy of the solutions for the soft-constrained problem, for varying number of inputs.

The results follow the same trend if we look at Figure 6.25, where we iterate over the number of prediction steps. Important here is that our software always provides an accurate solution, while keeping it up with qpOASES in terms of efficiency.

For better understanding of the numbers we reported here, we will show how the optimal value varies along some simulations. First, we show in Figure 6.26 a simulation where qpOASES is not that off the correct value. Still, we can observe some critical steps in the gap 350-400 where the performance of the controller may be degradated.

In Figure 6.27, we show the simulation for 12 masses. This simulation has a maximum error of 279% for qpOASES (0.4% for Mehrotra), and a mean error of 14% for qpOASES (0.02% for Mehrotra). Even though qpOASES shows a good accuracy for most simulation steps, in some critical ones it completely fails at identifying the optimum.

Finally, we show optimal value for the QPs in the simulation with $N = 6$ and $m = 4$. In this case, qpOASES would just not suffice as an optimization engine for our MPC controller.

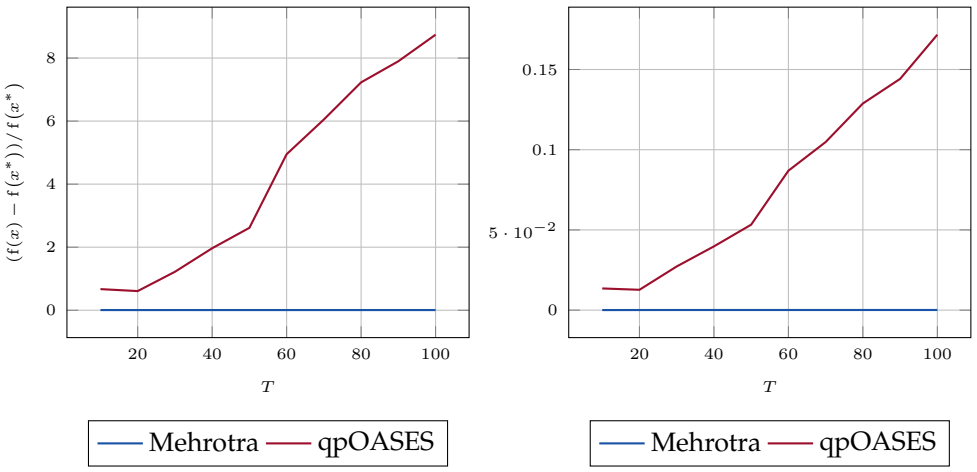


Figure 6.25: Accuracy of the solutions for the soft-constrained problem, for varying number of prediction steps.

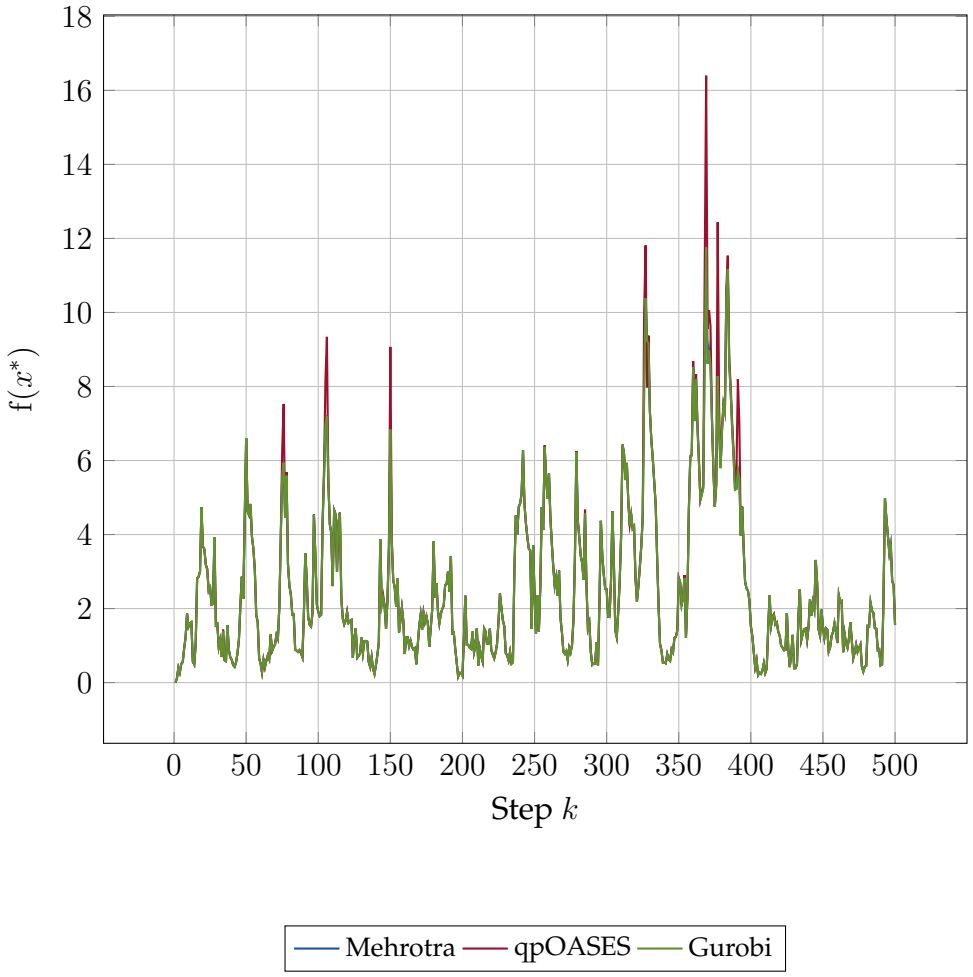


Figure 6.26: Optimal value for each iteration, as reported by each solver. Problem with $5$ masses, $4$ inputs and $20$ prediction steps.

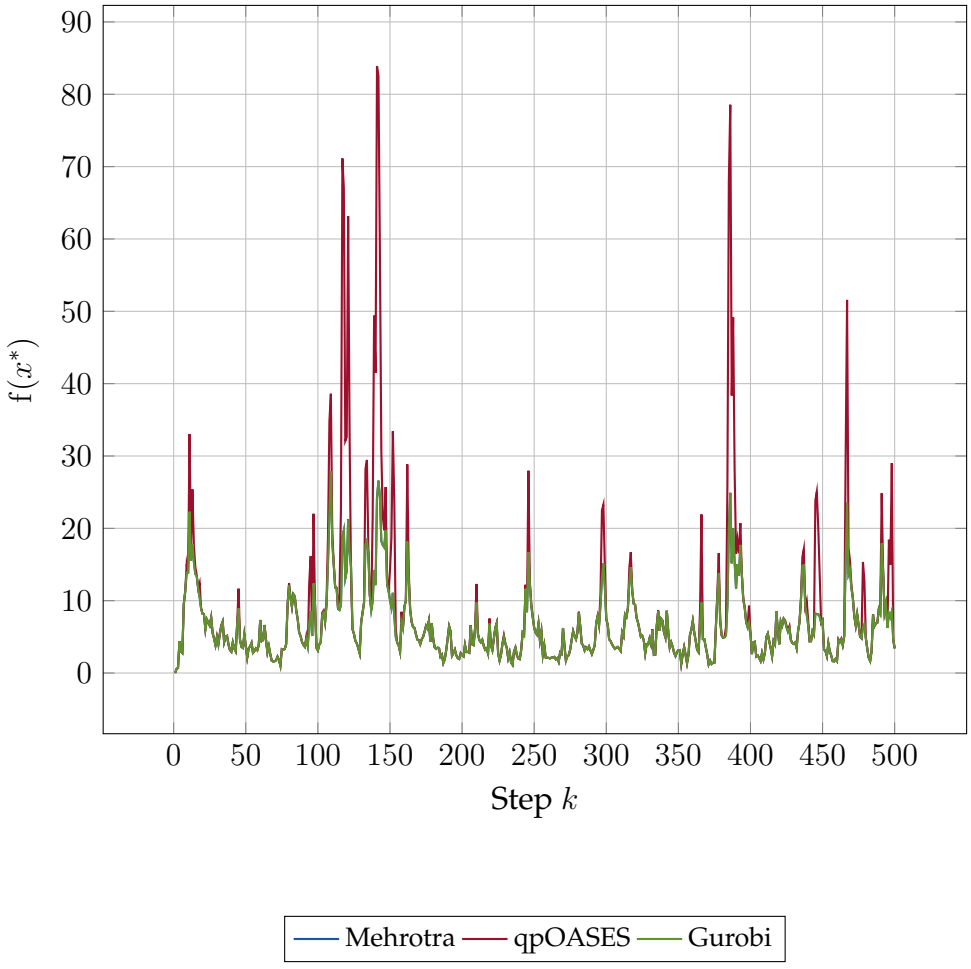


Figure 6.27: Optimal value for each iteration, as reported by each solver. Problem with 12 masses, 11 inputs and 20 prediction steps.

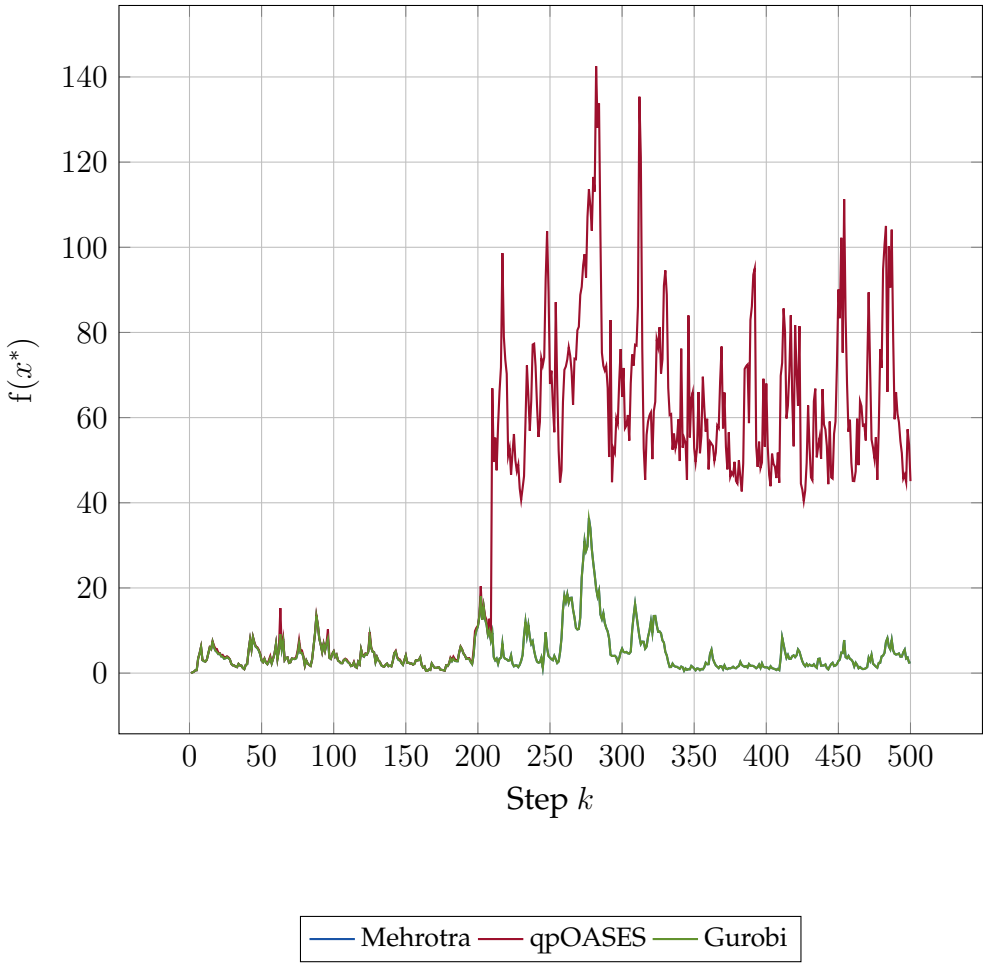


Figure 6.28: Optimal value for each iteration, as reported by each solver. Problem with $6$ masses, $4$ inputs and $20$ prediction steps.

# Chapter 7

# Conclusions

## 7.1 Summary

This thesis adresses the implementation of an efficient optimization method for embedded MPC applications.

First, we reviewed the theoretical properties of MPC and the arising convex QPs (Chapter 2). Then, we reviewed the main algorithms present in the literature, and chose one among them taking into account the particularities of our optimization problem (Chapter 3).

In our software, we used the algorithm proposed by Mehrotra [24], which excels in practical applications, and the sparse KKT system scheme proposed in [21], [22] (Chapter 4). Furthermore, we exploited some features of `C++` to optimize the execution of the code as much as possible (Chapter 5). This allows us to have a software capable of exploiting every nuance in the MPC formulation to speed up the process, yet avoiding code branching and reducing the memory footprint. These are desirable properties in embedded devices, with limited computational power and memory.

Finally, we reported experimental results of a realistic MPC benchmarking problem. We compared our implementation against an state-of-the-art active set method, qpOASES [28]. qpOASES is open-source, is widely used in practical applications and is also commonly used for benchmarking purposes.

In our studies, we analyzed different performance measurements. We focused primarily in worst-case computation time, mean computation time and accuracy of the solution.

For a comprenhensive benchmarking, we simulated the control of

different systems, with different set-ups. We run simulations of systems with varying number of states, inputs and prediction horizons. We also included random disturbances in our system, since MPC controllers usually have to deal with model inaccuracies and disturbances of any kind. Finally, we explored how our solver behaves with the inclusion of soft constraints, which is an utmost desirable property in industrial MPC applications.

## 7.2 Results

Our results are in general in accordance with the theory. We can observe that qpOASES shows, in general, much lower mean computation times. We did expect these results, since active set methods have very good warm-start capabilities. If there is little or no variation at all in the optimal active set, qpOASES can attain the optimum in zero iterations.

Since this is the most usual case in steady-state operation of the controller, we could also expect that qpOASES would show the lowest mean computation time. In contrast, our solver shows a minimum of three iterations per optimization problem, which seriously impacts the mean computation time.

Nonetheless, the worst-case computation time is similar for both solvers in small problems, and much for our implementation in large problems. We did also expect this behaviour. We know that active set methods have exponential complexity in the number of variables and constraints. If the initial guess is not good, the number of required iterations rapidly increases. In addition, we also know that interior point methods typically behave better in large optimization problems.

These results are of great importance for real-time applications. The mean computation time is important in simulations, where we aim to reduce the computation time as much as possible. In contrast, the worst-case execution time is the real bottleneck for real-time systems. According to our results, our software is, at least, comparable with qpOASES in terms of performance. According to our performance tests, we favour qpOASES in small problems, and favour our solution in large problems.

The inclusion of soft constraints has a big impact in the simulation results. Our software includes constraint softening natively. In contrast, qpOASES does not have an option to soften constraints, and we have

to do it manually.

There are different ways of implementing constraint softening. It is possible to introduce one slack variable per constraint, which is normally disencouraged due to big amount of computational effort. A popular alternative is to introduce one slack variable per prediction step, which penalizes the maximum constraint violation in that precise step. Our software implements constraint softening in the latter way. For the sake of consistency, we followed the same approach for qpOASES, manually modifying the problem matrices to include the slack variables in the formulation.

With this set-up, we observed that qpOASES becomes about one order of magnitude slower (when compared to the scenario with no soft constraints). This results in comparable mean computation times. Also, the worst-case computation time clearly favours our implementation, even in small problems.

Finally and foremost, we found important accuracy problems in qpOASES with the inclusion of soft constraints. After detecting that our implementation and qpOASES were reporting fairly different solutions to the QPs, we included a third optimization package in our tests. We have used Gurobi for that purpose, which is a commercial, general purpose solver. Whereas Gurobi is not competitive in terms of execution time, it is rendered robust and accurate for well-conditioned optimization problems.

For most QPs, we observe that our software and qpOASES provide the same correct result. However, in some critical QPs (roughly, those with a large change in the optimal basis), qpOASES greatly deviates from the optimal solution. This is illustrated in Figures 6.26, 6.27 and 6.28.

Again, this result was not completely unexpected. In their documentation, the authors of the qpOASES package warn that using the MPC option may weaken the robustness of their solver. Nevertheless, according to our experiments, we favour our software whenever soft constraints have to be included in the design.

## 7.3 Fulfillment of the Project Goals

In Section 1.2 we stated the goals of the project. We will review them here, one by one.

1. **Effectivity.** Throughout our experimentation, we have proven that our solver is effective (*i.e.*, it correctly solves the proposed problems). In spite of small mismatches, our solver always reported the same solution as Gurobi (within a tiny error margin). Both with and without soft constraints, we never encountered a problem instance that our solver could not solve accurately.

2. **Robustness.** We have not completely attained robustness in our software. We do not provide any support against infeasible problems, and we do not handle almost any degeneracy (for example, we require the penalty matrices to be strictly positive definite).

   However, we provide a consistent constraint softening feature. By taking advantage of such, the user can enforce every single problem to be feasible, which is sufficient for most practical MPC applications.

3. **Flexibility.** In our initial design, we were aiming to tackle as many different MPC formulations are possible. This requirement motivates the choice of a modular software design. Whereas we have covered a wide set of formulations (unconstrained and simple bounded variables, both as hard and soft constraints), the software prototype is still incomplete. Due to time limitations, we have not implemented the modules capable of handling linear and full constraints.

   Another limitation is that we do not support defective bounds; *i.e.*, if the input vector is bounded, all the inputs must be lower and upper bounded.

   Finally, as we stated before, we do not support positive semidefinite penalty matrices. However, due to our modular design, we hope that this features can be implemented in the future at a low marginal cost.

4. **Efficiency.** For the implemented features, we have carried out extensive testing against one state-of-the-art solver, namely qpOASES. Though we report larger mean computation times, we also report a similar speed in worst-case scenarios. qpOASES is certainly superior to our software in small to medium, hard constrained problems. Nevertheless, we have also found use cases were we report better solution times than qpOASES, with potential for practical use of our software.

Even though not all the goals have been fully attained, we hope that we can reach them in the future, as explained in the next section.

## 7.4 Future Work

Regarding the future work, we would like to make a distinciton between the items related to the algorithm, and the items related to the software itself. The first refers to a theoretical (and possibly empirical/-experimental) investigation, whereas the second refers to the implementation of the code.

Regarding the algorithm that we use, future research may include:

1. **Improved warm start** We have seen that, unlike active set methods, our approach requires a minimum number of iterations for every QP. This holds even if two consecutive QPs have very close solutions (*i.e.*, the solutions belong to the same active set). Warm-start of interior point methods is still an open question in the research community. Any strategy (or heuristics) capable of advantageously warm starting the algorithm would have a very significative impact in the mean computation time of the solver.

2. **Inconsistent linesearch strategies** In linear optimization, interior point methods typically use different step sizes for primal and dual variables. Here, we are constrained to using the same step size for both sets of variables. The reason for this is that the primal and dual variables are coupled via the Hessian matrix in the stationarity condition 2.13. Using different step sizes would result in not fulfilling the first block equation in 4.8 in the next iterate. However, using different step sizes also means taking longer steps, which may overcome the drawbacks of producing a larger residual $r_c$ (under certain circumstances).

3. **Stepsize scaling parameter** As discussed in Chapter 4, regarding Algorithm 1, the final step size is not applied entirely but scaled by a constant $\eta < 1$. Researchers [7], [12], [23] have found this scaling useful in practice. Roughly, the parameter $\eta$ prevents the pairs $\lambda$ - $s$ from becoming too unbalanced, which may lead to numerical instabilities in the KKT matrix.

   However, there is no general agreement regarding the best choice of $\eta$, which relies on heuristics, and varies between values as

different as $0.9$ and $0.9999$ in real applications. Moreover, to our knowledge there is no theoretical analysis about the influence of this parameter. Hence, we consider of interest studying the influence of the scaling factor $\eta$ in the overall performance of the algorithm, either theoretically or empirically.

Regarding the final software, future development may consist on:

1. **Static memory allocation support** As explained in Section 5.8, static memory allocation poses important advantages in certain applications. Because the aim of the project is developing a solver that can run in embedded devices, static memory allocation is a main concern, and a future development of the software should take this into account.

2. **Defective bounds** The solver, as we conceived it, only allows for *complete* bounds. That means that, if we decide that the system state is bounded, then all the states must be upper and lower bounded. Though apparently restrictive, this is a rather typical scenario if we consider real systems, in which signals are rarely allowed to grow towards infinity or minus infinity.

   Nonetheless, even if that is the case, the user may easily circumvent this problem by providing a large bound, that we know that the system will never reach. That makes the variable, in practice, behave as an unbounded variable. Nevertheless, we can think, design and develop a new policy that handles defective bounds explicitely, which would easen the design task of the user.

3. **Crossed input-state penalties** In our design, we are restricted to penalties in the form

   $$u_k^\top R u_k + x_k^\top Q x_k$$

   for positive definite penalty matrices $R$ and $Q$. This is a rather general formulation, and most linear MPC formulations fall into this category. However, it is also possible to penalize the crossed products input-state, as in:

   $$u_k^\top R u_k + x_k^\top Q x_k + 2 u_k^\top S x_k.$$

   The inclusion of the crossed penalty term $S$ in the sparse KKT matrix factorization has already been explored in [22]. Adapting

our software to using crossed penalties would enable the user to even more design freedom.

4. **Nonlinear MPC wrapper** A popular nonlinear optimization strategy is to solve a second order approximation of the optimization problem at each iterate, which provides the new search direction. That is the case of sequential quadratic programming, but also of some interior point methods (see LANCELOT, [8]). Hence, there a few nonlinear optimization algorithms that rely on solving a sequence of QPs with varying matrices. Our algorithm adapts relatively well to varying problem matrices, since we do not reuse matrix factorizations from one iteration to another (as ASMs do). For that reason, designing a nonlinear MPC solver that wraps around our current optimization engine is a possible future development direction.

# Bibliography


[1] C. R. Gutvik, T. A. Johansen, and A. O. Brubakk, "Optimal decompression of divers [applications of control]", *IEEE Control Systems*, vol. 31, no. 1, pp. 19–28, Feb. 2011. DOI: 10.1109/mcs.2010.939141.

[2] T. I. Bø and T. A. Johansen, "Dynamic safety constraints by scenario based economic model predictive control", *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 9412–9418, 2014. DOI: 10.3182/20140824-6-za-1003.00582.

[3] B. J. T. Binder, D. K. M. Kufoalor, A. Pavlov, and T. A. Johansen, "Embedded model predictive control for an electric submersible pump on a programmable logic controller", in *2014 IEEE Conference on Control Applications (CCA)*, IEEE, Oct. 2014. DOI: 10.1109/cca.2014.6981402.

[4] D. K. Kufoalor and T. A. Johansen, "Reconfigurable fault tolerant flight control based on nonlinear model predictive control", in *2013 American Control Conference*, IEEE, Jun. 2013. DOI: 10.1109/acc.2013.6580635.

[5] S. Vazquez, J. I. Leon, L. G. Franquelo, J. Rodriguez, H. A. Young, A. Marquez, and P. Zanchetta, "Model predictive control: A review of its applications in power electronics", *IEEE Industrial Electronics Magazine*, vol. 8, no. 1, pp. 16–31, Mar. 2014. DOI: 10.1109/mie.2013.2290138.

[6] T. A. Johansen, "Toward dependable embedded model predictive control", *IEEE Systems Journal*, vol. 11, no. 2, pp. 1208–1219, Jun. 2017. DOI: 10.1109/jsyst.2014.2368129.

[7] J. Nocedal and S. Wright, *Numerical Optimization (Springer Series in Operations Research and Financial Engineering)*. Springer, 2000, ISBN: 0-387-98793-2. [Online]. Available: https://www.

amazon.com/Numerical-Optimization-Operations-Financial-Engineering/dp/0387987932?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0387987932.

[8] A. R. Conn, G. I. M. Gould, and P. L. Toint, *Lancelot*. Springer Berlin Heidelberg, Dec. 6, 2010, 356 pp., ISBN: 3642081398. [Online]. Available: https://www.ebook.de/de/product/13905597/a_r_conn_g_i_m_gould_p_l_toint_lancelot.html.

[9] J. Maciejowski, *Predictive Control with Constraints*. Prentice Hall, Jan. 11, 2002, ISBN: 0201398230. [Online]. Available: http://www.ebook.de/de/product/3241503/jan_maciejowski_predictive_control_with_constraints.html.

[10] F. Borrelli, A. Bemporad, and M. Morari, *Predictive Control for Linear and Hybrid Systems*. Cambridge University Press, 2017, ISBN: 978-1107652873. [Online]. Available: https://www.amazon.com/Predictive-Control-Linear-Hybrid-Systems/dp/1107652871?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1107652871.

[11] A. Richards, "Fast model predictive control with soft constraints", *European Journal of Control*, vol. 25, pp. 51–59, Sep. 2015. DOI: 10.1016/j.ejcon.2015.05.003.

[12] S. Boyd and L. Vandenberghe, *Convex Optimization, With Corrections 2008*. Cambridge University Press, 2004, ISBN: 0-521-83378-7. [Online]. Available: https://www.amazon.com/Convex-Optimization-Corrections-2008-Stephen/dp/0521833787?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0521833787.

[13] O. Santin, "Numerical algorithms of quadratic programming for model predictive control", en, pp. -, 2016. DOI: 10.13140/rg.2.2.23784.88321.

[14] A. Bemporad, "Embeddded convex quadratic optimization for model predictive control", Sep. 8, 2014, [Online]. Available: cse.lab.imtlucca.it/~bemporad/talks/Embedded_QP4MPC.pdf.

[15] P. Patrinos, A. Guiggiani, and A. Bemporad, "A dual gradient-projection algorithm for model predictive control in fixed-point arithmetic", *Automatica*, vol. 55, pp. 226–235, May 2015. DOI: 10.1016/j.automatica.2015.03.002.

[16] D. Axehill and A. Hansson, "A dual gradient projection quadratic programming algorithm tailored for model predictive control", in *2008 47th IEEE Conference on Decision and Control*, IEEE, 2008. DOI: 10.1109/cdc.2008.4738961.

[17] P. Patrinos, P. Sopasakis, H. Sarimveis, and A. Bemporad, "Stochastic model predictive control for constrained discrete-time markovian switching systems", *Automatica*, vol. 50, no. 10, pp. 2504–2514, Oct. 2014. DOI: 10.1016/j.automatica.2014.08.031.

[18] H. J. Ferreau, S. Almer, H. Peyrl, J. L. Jerez, and A. Domahidi, "Survey of industrial applications of embedded model predictive control", in *2016 European Control Conference (ECC)*, IEEE, Jun. 2016. DOI: 10.1109/ecc.2016.7810351.

[19] R. Milman and E. J. Davison, "A fast MPC algorithm using nonfeasible active set methods", *Journal of Optimization Theory and Applications*, vol. 139, no. 3, pp. 591–616, May 2008. DOI: 10.1007/s10957-008-9413-3.

[20] I. Griva, S. G. Nash, and A. Sofer, *Linear and Nonlinear Optimization, Second Edition*. Society for Industrial Mathematics, 2008, ISBN: 0-89871-661-6. [Online]. Available: https://www.amazon.com/Linear-Nonlinear-Optimization-Second-Griva/dp/0898716616?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0898716616.

[21] Y. Wang and S. Boyd, "Fast model predictive control using online optimization", *IEEE Transactions on Control Systems Technology*, vol. 18, no. 2, pp. 267–278, Mar. 2010. DOI: 10.1109/tcst.2009.2017934.

[22] A. Domahidi, A. U. Zgraggen, M. N. Zeilinger, M. Morari, and C. N. Jones, "Efficient interior point methods for multistage problems arising in receding horizon control", in *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, IEEE, Dec. 2012. DOI: 10.1109/cdc.2012.6426855.

[23] S. J. Wright, *Primal-Dual Interior-Point Methods*. Society for Industrial and Applied Mathematics, Jan. 1997. DOI: 10.1137/1.9781611971453.

[24] S. Mehrotra, "On the implementation of a primal-dual interior point method", *SIAM Journal on Optimization*, vol. 2, no. 4, pp. 575–601, Nov. 1992. DOI: 10.1137/0802028.

[25] S. B. Lippman, J. Lajoie, and B. E. Moo, *C++ Primer (5th Edition)*. Addison-Wesley Professional, 2012, ISBN: 978-0-321-71411-4. [Online]. Available: https://www.amazon.com/Primer-5th-Stanley-B-Lippman/dp/0321714113?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0321714113.

[26] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001, ISBN: 0-201-70431-5. [Online]. Available: https://www.amazon.com/Modern-Design-Generic-Programming-Patterns/dp/0201704315?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0201704315.

[27] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 BLAS performance optimization on loongson 3a processor", in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, IEEE, Dec. 2012. DOI: 10.1109/icpads.2012.97.

[28] H. J. Ferreau, C. Kirches, A. Potschka, H. G. Bock, and M. Diehl, "qpOASES: A parametric active-set algorithm for quadratic programming", *Mathematical Programming Computation*, vol. 6, no. 4, pp. 327–363, Apr. 2014. DOI: 10.1007/s12532-014-0071-1.

# Appendix A

# Polymorphism and Generic Programming

This Appendix gives insight into the concepts of *polymorphism* and *generic programming*, which are neeeded for Chapter 5 if the reader is not familiar with them. Although these are generic Computer Science principles, we will use the programming language `C++` to introduce them. The reason is that `C++` is the language of choice for the implementation of our software.

## A.1 Polymorphism

In Computer Science, we say that an object is polymorphic when it *inherits* behaviour from other objects. The object you inherit from is the *Parent*, and the object that inherits is called *Child*.

What is special about inheritance is that childs share the behaviour of all its parents (in case that multiple inheritance is allowed, you can inherit from various parents at the same time). This is where the name polymorphism comes from, since childs behave as if they were several objects at once. Exploiting this principle, parents can be used to customize the behaviour of their children.

We shall illustrate this with an example. Assume that we have an object (a `class` in `C++`) that stores a vector and provides a function to evaluate the norm of the vector. (Object functions are called methods in `C++`, and we shall use this name from now on.) The minimal interface of such object is depicted in Code 7. The method `norm` return the norm of the vector stored in the class.

```
1  class Vector{
2  private:
3    int n;              // Vector dimension
4    double* vector;     // Array to store the vector
5
6  public:
7    double norm();      // Function that evaluates the norm
8    };
```

Code 7: Vector class with a method for evaluating the norm.

However, we may want to give the user the possibility of choosing between different types of norms. Depending on the application, the user may need to use the 1-norm or the 2-norm. Assume also that they are mutually exclusive: if the 1-norm is used, there is no need for the 2-norm.

One approach would be to implement two different methods (*e.g.*, `norm1` and `norm2`), and let the user choose the appropriate one. Nevertheless, then the user must change the signature in every piece of code. We may, instead, define a single method that takes a boolean argument: `0` for the 1-norm, `1` for the 2-norm (Code 8).

```
1  class Vector{
2  private:
3    int n;              // Vector dimension
4    double* vector;     // Array to store the vector
5
6  public:
7    double norm(bool x){
8      if (x==false)
9        return norm1();
10     else
11     return norm2();
12   }
13   double norm1();     // Returns the norm-1 of the vector
14   double norm2();     // Returns the norm-2 of the vector
15   };
```

Code 8: Vector class with a method for evaluating two different norms.

This way, the user may define the type of norm he or she wants to use at the beginning of the code, and always use the same signa-

ture. An example is provided in Code 9. Note that the user can easily change every instance of the method `norm` by changing the value of the constant variable `x` in the first line.

```
1  constexpr bool x = false;    // Every instance of "norm" returns
2                               // the 1-norm
3
4  int main(){
5    int n{3};                  // Vector dimension
6    double[3] v{1,2,3};        // Create the array [1, 2, 3]
7
8    Vector MyObject(n,v);      // Initialize the object with
9                               // dimension and data
10   MyObject.norm(x);          // Returns the 1-norm of the vector
11
12   return 0;
13 }
```

Code 9: Main code for using the class Vector.

However, note that this design has two of the main drawbacks that we mentioned before. In first place, it unnecesarily expands the code footprint, since the code for both the 1-norm and the 2-norm is available, even if only one of them is used. In this small examples, this is not important, but it can rapidly scalate for more complex software.

In second place, there is one `if-else` comparison (Code 8, line 8) that makes the code slower and harder to read, debug and maintain.

We can, instead, define the behaviour of the method `norm` using inheritance. First, we create two parent objects: one handles the 1-norm, the other one handles the 2-norm.

```
class norm1{
public:
  static double do_work(int n, double* x){
    double r{0};
    for (int i = 0; i < n; i++)
      r += abs(x[i]);
    return r;
  }
};

class norm2{
public:
  static double do_work(int n, double* x){
    double r{0};
    for (int i = 0; i < n; i++)
      r += x[i]*x[i];
    return sqrt(r);
  }
};
```

Code 10: Parent objects.

Then, we provide our `Vector` class with the desired behaviour, by inheriting from the appropriate parent.

```
class Vector : public norm1{  // Inherits from norm1
  private:
    int n;                    // Vector dimension
    double* vector;           // Array to store the vector

  public:
    double norm(bool x)
      norm1::do_work(n,vector);    // The parent implements the
                                   // desired behaviour
     };
```

Code 11: Child object.

We say that we delegate to the parent class the execution of the method. This design avoids code `if-else` branching, saves execution time and does not increase the memory footprint unnecessarily. Then, the user can safely use the 1-norm in his or her main program:

```
int main(){
  int n{3};
  double[3] v{1,2,3};

  Vector MyObject(n,v);    // Initialize the object with
                           // dimension and data
  MyObject.norm();         // Returns the 1-norm of the vector

  return 0;
}
```

Code 12: Main code for using the class Vector using the inheritance design.

Note that we do not have to specify what type of `norm` we want to use, since the parent class will handle it by itself.

The polymorphism principle does help us to avoid any code branching and reduce the memory footprint to a minimum. However, notice that this approach has a huge disadvantage that makes it difficult to use. To switch from one policy to another, we have to modify the inheritance chain. Although this is doable in small problems, it becomes cumbersome with larger problems. For instance, think about the algorithm presented in Chapter 5, which has `6859` possible inheritance chains.

## A.2 Generic programming

In Computer Science, the paradigm of generic programming allows us to circumvent this problem of hard-coding every inheritance chain that we want to use. According to the generic programming principle, we do not have to create specialized code for each particular instance of a family of problems.

In `C++`, generic programming is called *template programming*. According to [25]: "*A template is a blueprint or formula for creating a class or a function*". As the defination says, a template is not a piece of software itself; but a set of instructions for creating it. We will clarify this with an example.

Assume the following definition of a regular function for computing the square of an integer numer in `C++`:

```
1 int square(int x){
2   return x*x;
3 }
```

We can pass any object of type `int` to the function, but it will not accept other types such as `double` or `float`. In other words: given the problem of calculating the square of a number, the previous functions solves just a particular instance (the one of integer input data). Using template programming, we can give instructions to form the appropriate code:

```
1 template <class T>
2 T square(T x){
3   return x*x;
4 }
```

Here, `square` is a *templated function*. It is not a piece of code yet, since the type `T` is not known *a priori*. However, we can use the function in our code. We may call `square` on any data type that supports the product operation ("`*`") returning an object of the same type. This is the case of all basic `C++` types: `int`, `float`, `double`, `uint_16t`, etc.

Whenever we compile the source code, the compiler will generate the necessary functions, and only the necessary functions. For instance: if we call `square` on objects of type `double`, it will generate a regular `C++` function, that takes `double` as an argument and returns `double`. As opposed to it, the compiler will generate no code for the square function for integers, since it is not used.

Interestingly, if the function is not used, the compiler will just ignore it. In plain English: the compiler will work as if the templated function never was typed.

Template programming offers great potential when it comes to savings in the size of both the source code and the compiled code. The combination of polymorphism (to particularize behaviour) and generic programming (to specify the policy that we want to use without modifying the source code) is called Policy-Based Design.

# Appendix B

# Oscillating masses problem

Consider the oscillating masses setup depicted in Figure 6.1, described by equations (6.1) - (6.4). The Figure and the equations are reproduced here for convenience.

$$
\begin{aligned}
m_1\ddot{x}_1 &= -k_1x_1 + k_2(x_2 - x_1) + u_1,\\
m_2\ddot{x}_2 &= -k_2(x_2 - x_1) + k_3(x_3 - x_2) - u_1 + u_2,\\
m_3\ddot{x}_3 &= -k_3(x_3 - x_2) + k_4(x_4 - x_3) - u_2 + u_3,\\
m_3\ddot{x}_3 &= -k_4(x_4 - x_3) - k_5x_4 - u_3,
\end{aligned}
$$

Let $x = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 \end{pmatrix}^\top$ be the vector of positions, let $v = \begin{pmatrix} \dot{x}_1 & \dot{x}_2 & \dot{x}_3 & \dot{x}_4 \end{pmatrix}^\top$ be the vector of velocities and let $u = \begin{pmatrix} u_1 & u_2 \end{pmatrix}^\top$ be the input vector. The state-space representation of the system is given by:

$$
\begin{pmatrix} \dot{x} \\ \dot{v} \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & I \\ K & 0 \end{pmatrix}}_{A_c} \begin{pmatrix} x \\ v \end{pmatrix} + \underbrace{\begin{pmatrix} 0 \\ U \end{pmatrix}}_{B_c} u
$$

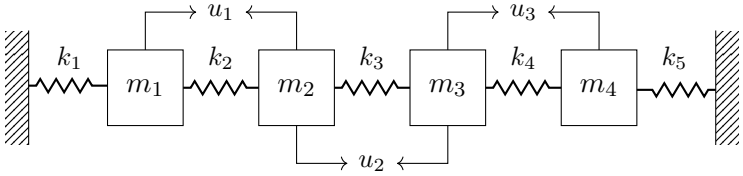


Figure B.1: Oscillating masses system (Figure 6.1).

where $0$ and $I$ are the zero and the identity matrices of appropriate dimensions, and

$$K = \begin{pmatrix} -\frac{k_1+k_2}{m_1} & \frac{k_2}{m_1} & 0 & 0 \\ \frac{k_2}{m_2} & -\frac{k_2+k_3}{m_2} & \frac{k_3}{m_2} & 0 \\ 0 & \frac{k_3}{m_3} & -\frac{k_3+k_4}{m_3} & \frac{k_4}{m_3} \\ 0 & 0 & \frac{k_4}{m_4} & -\frac{k_4+k_5}{m_4} \end{pmatrix}$$

$$U = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix}$$

Given a sampling time $T_s$, we discretize the system as follows:

$$\begin{pmatrix} x[k+1] \\ v[k+1] \end{pmatrix} = A \begin{pmatrix} x[k] \\ v[k] \end{pmatrix} + Bu[k], \tag{B.1}$$

where the exact discretization method is used:

$$A = e^{A_c T_s}$$
$$B = \int_0^{T_s} e^{A_c \tau} \mathrm{d}\tau B_c$$

Here, exact discretization means that, if $x[k] = x(t)$ and $u(t+\tau) = u[k]$ for $\tau \in [0, T_s)$, then necessarily $x[k+1] = x(t+T_s)$. In other words: the discretized system is identical to the continuous system at the sampling points, albeit they may differ for the time in between.

TRITA TRITA-EECS-EX-2018:655