Code Generation for Model Predictive Control of Embedded Systems

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

von

Juan Pablo Menendez Zometa

geboren am 29. Mai 1980 in Santa Tecla

genehmigt durch die Fakultät für Elektrotechnik und Informationstechnik

der Otto-von-Guericke-Universität Magdeburg

Gutachter:

Prof. Dr.-Ing. Rolf Findeisen

Prof. Dr. Daniel Limon

eingereicht am 23. August 2016

Promotionskolloquium am 12. Mai 2017

Acknowledgements

This thesis is the result of the work done as a research assistant at the Systems Theory and Automatic Control group of Prof. Rolf Findeisen, at the Otto-von-Guericke University in Magdeburg, Germany.

This work would have not been possible without the support of innumerable people. Special thanks go to Prof. Findeisen for his supervision and support, and to friends, colleagues and family for their encouragement.

Contents

Abstract

The internet of things is a novel paradigm based on networked physical devices that interact with their environment. This paradigm enables the improvement of existing technologies, and is likely to serve as the basis for yet unforeseen technological advancements. An internet-of-things device works on a computer network, and typically integrates sensors and actuators. At the core of any device is an embedded computer, commonly with very low computational capabilities. Model predictive control is an advanced control method likely to play a role in future internet-of-things applications as a way to improve the performance of networked cyber-physical systems. Therefore, the easy implementation of model predictive control on embedded hardware is of interest.

Model predictive control is based on the repeated solution of an optimization problem that allows to naturally handle multi-input multi-output systems subject to constraints. At every sampling time, inputs to the system are determined that optimize the predicted state evolution up to a certain future time instance. For this, an optimization problem must be solved in real time at each sampling period. This makes the implementation of model predictive control on embedded systems challenging. The difficulty on the implementation arises due to the combination of two factors: the limited computational capabilities of today's embedded hardware and the relatively high computational demands of solving the necessary optimization problem. The efficient and reliable implementation of model predictive control requires a tailored implementation.

A starting point for the efficient implementation is that for a given model predictive control formulation, the optimization problem belongs to a often narrow problem class (e.g. a quadratic program) with a particular structure. This structure does not change for many applications, and from one sampling time to the next only the problem data is different. The few available tools often exploit the special multistage structure of model predictive control problems. This leads to code whose computational complexity grows linearly with the horizon length by exploiting the sparsity of the problem. An alternative are, especially for short horizon lengths, condensed formulations.

We present a universal code generation tool that exploits condensed parametric formulations tailored for linear time-invariant discrete-time model predictive control. As base we introduce a simple modeling language that presents an easy to formulate model predictive control problems. Based on this language, we outline how one can automatically generate C code that is tailored towards dense solution approaches, which are especially suitable for low-performance and memory-constrained embedded processors. We furthermore implement an optimization algorithm tailored for model predictive control that explicitly takes into account the limitations of embedded hardware.

The presented modeling language is intuitive and easy to use, and offers great flexibility in the problem formulation. We demonstrate via several examples that the outlined code generation approach allows for fast solutions even on low-cost embedded platforms.

Deutsche Kurzfassung

Das Internet der Dinge ist ein neues Paradigma, welches auf vernetzten physikalischen Geräten basiert, die mit ihrer Umgebung interagieren. Dieses Paradigma ermöglicht die Verbesserung von existieren Technologien und wird wahrscheinlich als Grundlage für noch unvorhergesehene technologische Fortschritte dienen. Ein Internet-der-Dinge-Gerät arbeitet auf einem Computer und bezieht typischerweise Sensoren und Aktoren mit ein. Im Kern eines jeden Gerätes befindet sich ein eingebetteter Computer, welcher für gewöhnlich über eine geringe Rechenleistung verfügt. Modellprädiktive Regelung ist eine erweiterte Methode in der Regelungstechnik, die wahrscheinlich eine zukünftige Rolle in der Anwendung des Internets der Dinge spielen wird. Sie stellt einen Weg dar, die Leistung von vernetzten cyber-physikalischen Systemen zu verbessern. Deshalb ist die einfach umsetzbare Implementierung von modellprädiktiver Regelung auf eingebetteter Hardware von Interesse.

Modellprädiktive Regelung ist ein fortschrittliches Regelungsverfahren, basierend auf der wiederholten Lösung eines Optimierungsproblems, mit dem auf einfache Weise auch Mehrgrößensysteme und Systeme mit Beschränkungen behandelt werden können. Zu jedem Abtastzeitpunkt wird ein Eingang für das System berechnet, welcher die prädizierte Systementwicklung optimiert. Somit muss zu jedem Abtastzeitpunkt wiederholt und in Echtzeit ein Optimierungsproblem gelöst werden. Die wiederholte Lösung dieser Probleme macht die Implementierung der Modellprädiktiven Regelung auf eingebetteten Systemen zu einer Herausforderung. Die Schwierigkeiten, die bei der Umsetzung entstehen sind auf eine Kombination von zwei Faktoren zurückzuführen: die begrenzte Rechenleistung eingebetteter Systeme sowie die relativ hohen Rechenanforderungen zur Lösung der Optimierungsprobleme. Daraus folgt, dass eine effiziente Umsetzung der Modellprädiktiven Regelung eine maßgeschneiderte problemspezifische Implementierung erfordert. Für ein gegebenes Modellprädiktives Regelungsproblem gehört das entsprechende Optimierungsproblem oftmals zu einer speziellen Klasse (z.B. Quadratisches Programm) mit spezieller Struktur. In vielen Anwendungen bleibt diese Struktur

von einem Abtastzeitpunkt zum nächsten konstant, es ändern sich lediglich die Problemdaten. Die wenigen, verfügbaren Software-Tools nutzen die inhärente Struktur der Modellprädiktiven Regelung aus, was zu einem linearen Wachstum der Rechenkomplexität mit der Länge des Prediktionshorizonts führt. Diese Implementierungen nutzen oft dünnbesetzte Matrizen zur Beschleunigung der Lösung des Optimierungsproblems aus. In einigen Fällen, insbesondere für Probleme mit kurzen Prädiktionshorizonten auf eingebetteten Systemen, sind allerdings kondensierte Formulierungen besser geeignet. In dieser Arbeit wird ein Software-Tool zur Codegenerierung vorgestellt, welches auf einer parametrischen Formulierung der linearen zeitinvarianten diskreten Modellprädiktive Regelung beruht. Hierzu wird zunächst eine einfache Modellierungssprache vorgestellt, welche erlaubt modellprädiktive Regelungsprobleme einfach zu formulieren. Auf Basis dieser Sprache wird umrissen, wie automatisch C-Code generiert werden kann, der die Verwendung kondensierter Lösungsansätze ermöglicht. Darüber hinaus wird ein Optimierungsalgorithmus implementiert, der die Einschränkungen eingebetteter Systeme berücksichtigt. Die vorgestellte Modellierungssprache ist intuitiv und bietet große Flexibilität bei der Problemformulierung. Anhand einiger Beispiele wird demonstriert, dass der vorgestellte Ansatz der Codegenerierung zur schnellen Lösung von Modellprädiktiven Regelungsproblemen, auch auf preisgünstigen eingebetteten Systemen, ermöglicht.

1. Introduction

Despite many conceptual and theoretical advances, the systematic use of advanced control approaches for constrained system in industrial applications is still limited [1]. Often simpler unconstrained methods like a proportional-integral control might be sufficient to stabilize a system with constraints. In such cases, a more complex algorithm that can explicitly take into account the constraints may be, due to technical and economic reasons, hard to justify even if it delivers superior closed-loop performance [1]. A notable exception is the process industry, where model predictive control (MPC) [2, 3, 4, 5, 6, 7, 8] has been consistently applied for nearly four decades [9]. This is due several factors, including that the processes are slow and that operation near state constraints offers great economic benefits.

In recent years, the use of MPC has been applied to a broad range of applications, e.g. path-following in robotics [10], building climate control [11], control of water canals [12], and biomedical applications such as artificial pancreases [13].

In the case of fast embedded systems subject to constraints, MPC has been traditionally considered difficult to apply due to the high computational demands of the MPC algorithm relative to the computational capabilities of embedded computers. Therefore, a widespread use of MPC on embedded applications has been limited in the past and has only gained momentum in recent years. The constantly decreasing cost/performance ratio of computers, and, most importantly, the recent improvement in MPC-tailored optimization algorithms has made the implementation of MPC in embedded hardware possible [14, 15, 16, 17], albeit still challenging on low-cost devices. This work attempts to further simplify the application of MPC on low-cost embedded systems by introducing a new code generation software tool for MPC that explicitly considers the limitations of low-cost embedded hardware. The two main components of the generated code are an efficient optimization solver and data structures tailored for embedded applications.

1.1. Linear model predictive control for embedded systems

Model predictive control is an advanced control method that takes into account system constraints, and is based on repeated optimization using information of a prediction model. According to [7, Ch. 1], model predictive control is the only generic control method that can routinely deal with constraints. Furthermore, [8] states that MPC is "perhaps, the most general way of posing the process control problem in the time domain". The fact that it is generic implies that a broad range of applications that deal with the control of constrained systems can be considered within the MPC framework. A further appeal of MPC is that it is conceptually intuitive, although its inner operation rely on complex and less intuitive concepts. Many types of MPC formulations exist. What all of them have in common are the explicit use of a mathematical model to make predictions and a receding horizon optimization [18]. MPC has been developed at least in three major streams: the industrial or classical setting, the one based on adaptive control, and the synthesis approach [8]. We focus here on the synthesis approach. Two of the main characteristic of this approach are the use of state-space models and stability guarantees [18].

To this end, we consider the linear discrete-time dynamical system

$$x^+ = Ax + Bu + h(w)$$

subject to input and state constraints

$$u \in U,\ x \in X,$$

with x the current state, x^+ the successor state, u the input to the system and $h(w)$ a function affine in the parameter w. This parameter represents a reference trajectory or a known disturbances (or a combination of them), for example. The input constraints U are often determined by actuator limits. Examples are the minimum and maximum voltage that can be applied to an electric motor, or the maximum aperture of a valve. These are usually hard constraints, i.e. they are enforced by the hardware and, under normal operation, cannot be violated. The state constraints X are usually determined by several factors, such as safety, comfort, and efficiency. These are commonly artificially

introduced by design. For example the acceleration of a vehicle can be artificially limited for comfort reasons, or the temperature or pressure of a chemical process can be limited for economic and safety reasons. These types of constraints are often considered as soft. In practice they may be violated, e.g. due to several sources of uncertainty in the control loop like disturbances and model-plant mismatch. In other words, soft constraints are desired limits that, in case of not being respected, the controller can relax.

To control this type of constrained system, we use the following linear MPC problem formulation:

$$\begin{aligned}
\underset{\mathbf{u}}{\text{minimize}} \quad & \sum_{j=0}^{N-1} \ell(x_j, u_j, w_j) + \ell_N(x_N, w_N) & \\
\text{subject to} \quad & x_{j+1} = Ax_j + Bu_j + h(w_j) & j = 0, ..., N-1 \\
& u_j \in U & j = 0, ..., N-1 \\
& x_j \in X & j = 0, ..., N-1 \\
& x_N \in X_f \subseteq X & \\
& x_0 = x. &
\end{aligned} \tag{1.1}$$

The stage cost $\ell(x_j, u_j, w_j)$, the terminal cost $\ell_N(x_N, w_N)$ and the terminal constraint set X_f constitute the *three ingredients* that help establish stability of the closed-loop system, [19]. The horizon length is given by N. The stage cost is commonly used to determine the desired behavior by way of weighting matrices. Both, the stage and terminal costs, are commonly quadratic functions. The MPC formulation is an optimization problem that needs to be solved at each sampling time. The solution to the problem for the state x at the current sampling time is used to compute the current control input u (see [3, Ch. 2] for further details).

Note that the stage cost $\ell(x_j, u_j, w_j)$ is repeated for $j = 0, \cdots, N-1$. Thus, in this work we would refer to MPC as belonging to the broader class of *multistage* problems. This classification also includes moving horizon estimation problems [3, Ch. 3].

We are particularly interested in MPC problems like (1.1) with a quadratic cost and affine inequality constraints and affine equality constraints (i.e. the linear dynamics). Such problems lead to quadratic programs (QP). To find a solution using iterative algorithms, the MPC problem is often brought into a special form. We distinguish between two types of problem formulations: sparse and condensed.

On the one hand, sparse formulations exploit the multi-stage nature of MPC to achieve a computational complexity that grows linearly with the horizon length, instead of cubically in the case of non-tailored interior point methods [20]. On the other hand, condensed problem formulations use dense matrices without a directly exploitable structure. The computational complexity of a condensed solver typically grows cubically or quadratically with the horizon length. In general, condensed formulations are preferred in cases where the horizon length is short. In embedded applications, the horizon length is often selected to be as short as possible, while still delivering suitable performance. Thus, condensed formulations can be of computational advantage for embedded applications. One specific problem that benefits from a condensed formulation is the case when the number of states is much larger than the number of inputs. For example, stochastic MPC formulations based on polynomial chaos expansion show this property [21].

One often cited drawback of MPC is that its computational requirements are significantly higher in comparison to simpler unconstrained controllers. This is particularly relevant when implementing MPC on an embedded computer. An embedded computer in general provides only a small fraction of the computational performance of a modern desktop computer. Implementing MPC on an embedded computer is in general challenging. Although nowadays very powerful off-the-shelf microprocessors are cheaply available, in some applications, for technical or economical requirements, low-cost resource-limited embedded processors need to be used. Technical limitations may include low energy consumption, small footprint, etc. Furthermore, in cost-sensitive applications, the cheapest processor that can do the job is typically selected.

A great amount of research has been performed recently that aims at speeding up MPC implementations. Of particular interest are automatic code generation methods for MPC for embedded systems. [22, 23, 24, 25]. The current state of research on this field can be split into two branches: tailored solvers and automatic code generators.

One way to speed up finding a solution is the use of *explicit* MPC [26]. In this approach, the solutions are computed off-line and stored in a lookup table. This approach has high memory requirements and is in general limited to problems of small size [27]. In the following, we restrict our attention to MPC approaches that require solving an optimization problem on-line. We focus on optimization algorithms that iteratively search for a solution to the optimization problem. That is, the algorithm looks for a point that minimizes the value of the cost function and that, at the same time, satisfies all constraints.

A large variety of literature exist in the field of optimization [28, 29, 30]. Of particular interest in this work are *convex* optimization problems [31]. Although convex problems can be solved reliably, an off-the-shelf general purpose solver might not be fast enough for an MPC application. Furthermore, such a solver might be difficult to use on an embedded platform due to software dependencies and hardware limitations. This has motivated the development of solvers that take into account the MPC characteristics and limitations of embedded applications. Among the most commonly exploited characteristics are the MPC structure and warm starting. The former commonly refers to the exploitation of the banded structure of the problem matrices arising due to the predictive nature of MPC. Warm start refers to the use of the solution to the MPC problem at the previous sampling time as the initial guess for the solution of the problem at the current sampling time.

One of the first reported on-line algorithms that fully exploited the MPC structure for linear systems was presented in [20], which used a primal-dual interior point method (IPM). This has been the base for recent code generation tools tailored for MPC like the ones presented in [22, 23]. Primal-dual IPMs spends the first iterations finding the so-called central path. Once in that path, the algorithm can quickly converge to a solution. Hence, these methods do not typically make much progress on improving the quality of the solution on the very first iterations. Furthermore, primal-dual IPM are difficult to warm start [27]. This might restrict the application of tailored primal-dual IPMs on embedded applications where acceptable performance can be attained by rough solutions and only very limited time is available to compute the solutions. Alternatively, a tailored primal barrier IPM that can be warm started was discussed in [32]. This method is in general not as fast and accurate as primal-dual methods, but may offer a good trade-off between speed an accuracy for embedded applications. Finally, active-set methods have also been tailored to efficiently solve optimization problems arising from MPC, with perhaps the most popular being the *online active set strategy* [27], and its implementation, called qpOASES [33].

In general, the theoretical computational bounds are much larger than the ones observed in practice for interior point methods [31, 34] and active set methods [29, 35]. In safety-critical applications, error bound certification might be required. First-order methods in general have tighter bounds, and may be preferred for such applications. Some theoretical contribution in this direction have been made in [36, 37, 38].

The second branch of research aiming to speed up MPC has been automatic code

generation tools. These tools commonly consist on a parser of a high-level domain-specific language, and a code generator of a tailored optimization solver. Typically the C programming language [39] is used for the generated code. Commonly, the generated code does not depend on external software libraries, i.e. an appropriate C compiler (e.g. [40]) suffices to generate executable binaries for a hardware target platform.

Several parsers for general purpose optimization problems exist, e.g. YALMIP [41], AMPL [42], and CVX [43]. However, these are mainly used to feed data to general purpose optimization routines, i.e. the solver does not take into account many of the characteristics of each problem at hand, which makes them in general less efficient. In contrast, CVXGEN, [44] generates *embedded solvers* for convex quadratic programs. In the CVXGEN context, embedded means that, given a problem with certain characteristics, solvers that exploit these characteristics are generated. CVXGEN reports several orders of magnitude improvements in the solution time compared to CVX [44]. Nevertheless, the C code generated by CVXGEN grows very quickly in size (see Subsection 3.2.1 for further details). While CVXGEN can handle MPC problems [45], the generated code does not exploit the structure inherent in MPC. The combination of QCML and ECOS as presented in [24], the former being a parser and the latter being a stand alone general purpose second-order cone program solver [46], is based on similar ideas as CVXGEN. Both CVXGEN and QCML do not eliminate the equality constraints due to the system dynamics, leading to a sparse formulation.

In [47] two code generation tools are presented. One is FiOrdOs, which based on first-order methods that do not exploit structure inherent in MPC [48]. The other one is FORCES, a structure-exploiting primal-dual interior point method [22]. Parsers specific to nonlinear MPC have also been developed. The ACADO toolkit [49] is a C++ software tool that provides C-code generation capabilities for continuous-time nonlinear MPC problems [50]. Although it is possible to use the ACADO toolkit for discrete-time linear systems as well, the generated code is bulkier than code tailored for linear discrete-time MPC problems.

1.2. Contribution

This work addresses the computational drawback of MPC without compromising its advantages. The main contribution is a free MPC software tool called *μAO-MPC* that

simplifies the deployment of MPC controllers, while at the same time taking into consideration the computational limitation inherent to embedded processors. Because *µAO-MPC* relies on a flexible problem formulation, it can be used to deploy moving horizon estimators as well. In this sense, we retain the main advantages of the MPC framework, namely its intuitiveness and generality, while at the same time addressing one of its main cited drawbacks, i.e. prohibitive computational requirements for embedded computers.

Broadly speaking, *µAO-MPC* consist of three parts: a parser for a simple MPC problem specification, a C code generator and a tailored optimization algorithm. The typical workflow is as follows: the user specifies an MPC problem using a language that closely resembles the way MPC problems are expressed mathematically. This specification is parsed and transformed into an optimization problem, in particular a quadratic program (QP) with dense matrices. Finally, C code is generated that contains the QP together with a first-order optimization algorithm that can approximately solve the optimization problem efficiently.

In this work, we follow an approach similar to CVXGEN in the way of how MPC problems are expressed (i.e. simple text-based mathematical representation). We focus however on code generation of embedded MPC solvers for computationally limited systems using first-order methods, similar to FiOrdOs. At the moment, we have focused on solving QPs and our results show that *µAO-MPC* generates code of much smaller size than CVXGEN. One key difference from our approach and all other existing parser/code generators for linear systems is that we eliminate the equality constraints to derived a condensed formulation. Furthermore, the solver we use finds control inputs appropriate for certain types of embedded control faster than CVXGEN. This control inputs are based on rough approximate solutions, which are nevertheless in many applications good enough for control. We do not compare our results against FiOrdOs, but we would expect that *µAO-MPC* and FiOrdOs deliver similar performance in many situation. A full comparison with many optimization algorithms is beyond the scope of this work. The main contribution is therefore not in the field of convex optimization, but rather on the implementation aspects of automatic code generation tools for model predictive control on embedded systems.

In this work we extend the results presented in [25] by introducing a method and a parser that automatically generates easily usable code for the efficient on-line solution based on a condensed formulation. This approach, which is specially tailored for linear time-invariant MPC problems, allows (besides QP formulations) the consideration of

linear programs (arising e.g. in min-max robust MPC [51]), quadratically constrained quadratic programs (arising in robust MPC by adding ellipsoidal constraints [52]) and second-order cone programs (arising in stochastic MPC [53]). We show how to systematically go from the MPC description to the condensed formulation using the proposed method. Thus, we extend *µAO-MPC* capabilities two-fold: first, the presented approach allows to generate data tailored for linear programs, quadratically constrained programs and second order cone programs in dense form. Second, we extend the family of problems that can be considered by introducing a multistage modeling language. This allows not only to consider a broader range of multistage problems, but also moving horizon estimation problems and special filtering problems. Note that we focus here on the generation of the dense problem data, and not on a particular solver. To the best of our knowledge, this is the first tailored tool for linear time-invariant discrete-time MPC that can automatically parse, condense the data, and generate easily usable code. Loosely related, but tailored towards continuous-time nonlinear systems is the ACADO Toolkit [49, 50].

1.3. Outline

This work is organized as follows. In Chapter 2 we describe the overall setup, namely digital control of embedded systems and the MPC algorithm. We discuss the properties of embedded systems most relevant to MPC applications. In particular, we discuss the computational limitations of embedded systems. Furthermore, we present the theoretical foundations of convex optimization applied in particular to convex quadratic programs.

In Chapter 3 we discuss optimization algorithms tailored for embedded MPC. We briefly discuss some state-of-the-art approaches. The chapter focuses on a novel optimization algorithm tailored for embedded MPC applications. The algorithm is based on an augmented Lagrangian method combined with Nesterov's fast gradient method (the ALM+FGM algorithm).

In Chapter 4 we introduce a general MPC language specification. We show how some types of MPC problems specified using this language can be transformed into condensed parametric quadratically constrained quadratic programs in a structured way.

Chapter 5 presents *µAO-MPC*, an automatic code generation tool for MPC written in Python. The software generates portable C-code for MPC problems specified in

the language introduced in Chapter 4. By default the generated condensed QPs are solved using the tailored ALM+FGM algorithm discussed in Chapter 3. We show with some simulation examples *µAO-MPC*'s suitability for embedded applications, taken into account the properties discussed in Chapter 2.

In Chapter 6 we present two application examples that again demonstrate that *µAO-MPC* is appropriate for a specific type of embedded applications, namely those that can be warm started, are well conditioned and where rather rough approximate solutions deliver good closed-loop performance.

In Chapter 7 we present our conclusions and discuss some ideas for future work.

2. Model predictive control for embedded systems

Nowadays, most closed-loop control schemes are implemented on digital systems. This is mainly due to the flexibility provided by software implementations, and the ubiquity of microprocessors. Furthermore, there is a strong body of control theory and software tools that covers the design of digital control systems. Particularly well developed is the theory of linear unconstrained single-input single-output systems [54, 55]. Usually, the analysis is performed in the frequency domain (i.e. *z-transform* for discrete-time systems). In the case of unconstrained multi-input multi-output (MIMO) systems, modern control methods exist that are well established, such as $\mathcal{H}_2$ and $\mathcal{H}_\infty$ control [54, Ch. 40]. Optimal control methods together with state-space representations are commonly used for the control synthesis of MIMO systems [56, 57]. In the case of *constrained* MIMO systems, model predictive control (MPC) is the most widespread technique [7, 8].

This chapter lays the theoretical foundations needed for a basic understanding of the main characteristics of MPC for embedded systems.

2.1. Embedded digital control systems

This section describes the most relevant characteristics of digitally implemented control systems.

2.1.1. Digital control of linear dynamic systems

We start this section with a few definitions motivated by [55]. We expand later on some concepts that are key for the further development of this topic.

Figure 2.1 shows the general scheme of a digital control system. We first focus on the right part of the figure, which represents the analog (or continuous) part of the control

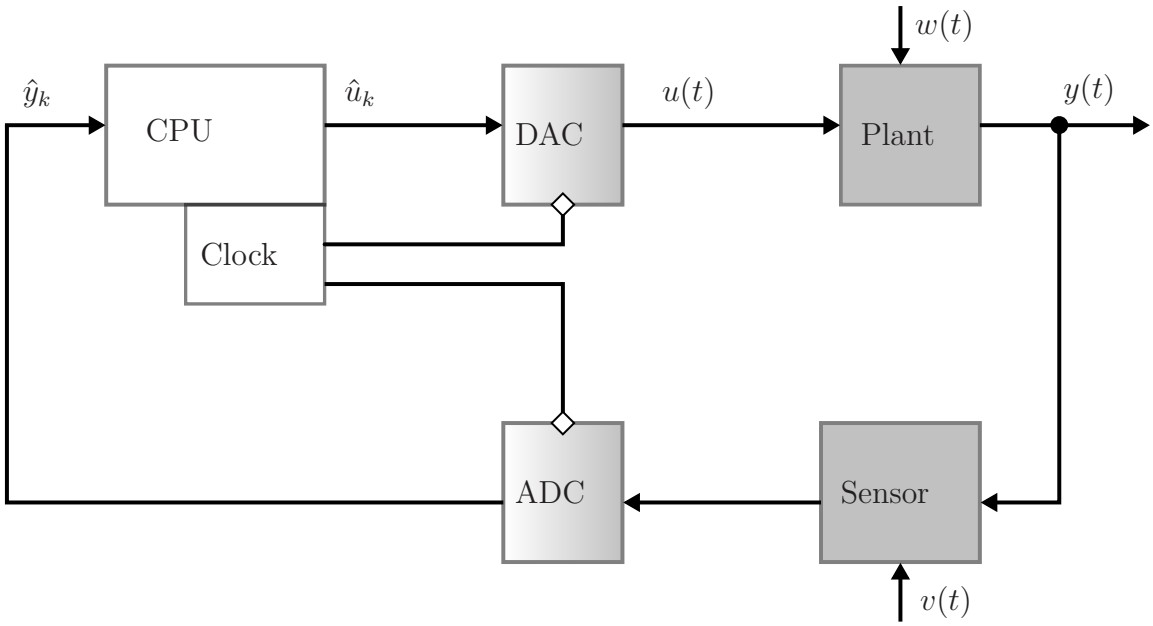

Figure 2.1.: Block diagram of a basic digital control system, adapted from [55].

system. These are represented with grey boxes. We consider the *plant* to be any physical process that needs to be controlled. Most of the physical plants of interest are dynamical systems which are continuous in nature. We want to modify the behavior $y(t)$ of the plant with a control input $u(t)$. The variable $w(t)$ represents disturbances. Finally, we capture the magnitude of physical variables of the plant via sensors. We use $v(t)$ to denote disturbances or noise in the sensor. All these variables change continuously in time.

Let us now turn our attention to the middle part of Figure 2.1, which represents the bridge between the digital and the analog part of the control system, represented by the grey/white boxes. The analog-to-digital converter (ADC) samples a physical variable and then converts this value into a digital number representation. We assume that these samples are taken at constant time intervals, which are driven by the clock. We call the interval between any two samples the *sampling period*. The signals generated by the ADC vary only at discrete times, more precisely at every $t = kT$, with T the sampling period, and $k = 0, 1, \ldots$. This type of signals are called *discrete signals*. For simplicity of notation, we write any discrete signal $y(kT)$ in the form y_k. A system that has an interface to continuous signals (e.g. $y(t)$) and discrete signals (e.g. y_k) is known as a *sampled-data system* [54, Ch. 12]. Additionally, the ADC represents the value of y_k

as a digital number with a limited number of binary digits. This introduces the effect known as *quantization*, which refers to the effect of representing y_k as a value $\hat{y}_k$ that has been approximated to a certain precision. Digital computers work, therefore, with *digital signals*, i.e. signals that are both quantized and discrete.

In the middle part of Figure 2.1 the digital-to-analog converter (DAC) is also depicted, which performs the opposite function of the ADC. The same concepts apply. The main difference is that the DAC usually implements a *zero-order hold* to keep the analog signal constant during a whole sampling period (the ADC also does this, but only for a fraction of the sampling period).

Finally, on the left side of the figure we have the digital part of our control system, represented by white boxes. The central processing unit (CPU) is in charge of computing a control action $\hat{u}_k$ based on the current sensor information $\hat{y}_k$ via a control algorithm.

Two of the most important factors that affect the performance of sampled data systems are quantization and the sampling period. If the sampling period is sufficiently small, and the quantization effects are negligible, a sampled-data control system might behave in some cases almost identically to a continuous-time control system [54, Ch. 15]. Usually, in cost-sensitive embedded control systems the sample period and quantization effects are not negligible and need to be taken into account in the design of the control system. We further expand the concepts and effects of quantization and the sampling period in the following.

Quantization

Loosely speaking, quantization is an effect where a digital number with limited precision approximates a real number [54, Ch. 15]. Consider a signal $z \in \mathcal{Q}_z \subseteq \mathbb{R}$ in the interval $[\underline{z},\ \overline{z}]$ that is represented by the quantized value $\hat{z}$. The set $\mathcal{Q}_z$ can represent the values of a continuous signal (e.g. $\mathcal{Q}_z = \mathbb{R}$) or of a digital signal (i.e. z is itself quantized). The bounds $\underline{z}$ and $\overline{z}$ represent, for example, actuator or sensor limits. In this context we define quantization $\Phi : \mathcal{Q}_i \to \mathcal{Q}_o$ as a nonlinear map from the set $\mathcal{Q}_i = \{z \in \mathcal{Q}_z \subseteq \mathbb{R} \mid \underline{z} \leq z \leq \overline{z}\}$ to the set $\mathcal{Q}_o = \{\hat{z} \in \mathbb{R} \mid \hat{z} = \underline{z} + s\rho(\overline{z} - \underline{z})\}$, where $\rho = 2^{-l}$, with $l \in \mathbb{Z}$ is the number of bits available to represent the digital number and $s \in \mathbb{Z}$, $0 \leq s \leq (2^l - 1)$ is the full range of raw binary values. Note that $\mathcal{Q}_o$ is defined by an affine function of integer numbers. Other definitions are also possible, e.g. using logarithmic functions.

The quantization Φ is usually defined in either of the following two ways. The first is truncation, which means mapping a real number in $\mathcal{Q}_i$ to the largest previous number in $\mathcal{Q}_o$. The second way is round-off, which is more commonly used in control computers. Round-off simply maps any number in $\mathcal{Q}_i$ to the nearest number in $\mathcal{Q}_o$. We assume in the following that the quantization function uses round-off.

We define the *maximum quantization error* β for Φ as

$$\beta = \frac{1}{2}\rho(\overline{z} - \underline{z}). \tag{2.1}$$

Furthermore, for the mapping $\Phi : z \mapsto \hat{z}$ we have $\beta \leq |z - \hat{z}|$. For example, consider the quantization happening in the ADC and the DAC. In the former case, we have $\Phi : y(kT) \mapsto \hat{y}_k$ with $\mathcal{Q}_z = \mathbb{R}$. In the latter case, the main difference is that the input set $\mathcal{Q}_i$ is now also defined for a quantized signal, with $\mathcal{Q}_z$ possibly defined similarly as $\mathcal{Q}_o$ (i.e. quantized).

Quantization have different effects in the CPU than in the ADC or the DAC. In many applications, the round-off errors for floating-point operations performed by the CPU can be neglected. However, floating-point arithmetic can be sometimes replaced by *fixed-point* arithmetic. The notation $Qa.b$ denotes binary fixed-point numeric representation using $a + b + 1$ bits, with a integer bits, b fractional bits and 1 sign bit. For example, $Q13.18$ denotes 32-bit binary fixed-point numeric representation, with 13 integer bits plus 1 sign bit and 18 fractional bits. Fixed-point arithmetic operations are performed in general more efficiently (in terms of energy and speed) than floating-point operations. The main drawback of fixed-point arithmetic is an increase in round-off errors due to the lower precision when compared to floating-point arithmetic. See [58] for a discussion of fixed-point arithmetic used in model predictive control.

In contrast to the CPU, quantization effects play a major role on the DAC and ADC, because these components use shorter word sizes (commonly between 8 and 16 bits). The maximum quantization error β may be large enough to noticeably degrade performance in cost-sensitive applications, where 8-bit or 10-bit DACs and ADCs are typically used.

Sampling period

The selection of the sampling rate is commonly based on a rule of thumb and a trade-off between cost and performance. The first thing to consider is that the sample rate ω_s must be at least twice the bandwidth ω_b of the closed-loop. The ratio $\frac{\omega_s}{\omega_b} > 2$ is given by

the Nyquist-Shannon sampling theorem, which states that a signal that is a bandlimited by ω_b is uniquely characterized by its uniform samples taken with a frequency $\omega_s > 2\omega_b$, see [54, Ch. 16]. In practice, higher ratios are required to have a smooth time response. The following rule of thumb is commonly used [55]:

$$20 < \frac{\omega_s}{\omega_b} < 40.$$

In [54, Ch. 16] other possible heuristic rules are also discussed.

The next consideration in determining the sampling rate is cost. In general, higher sampling rates imply better closed-loop performance. If cost is no concern, a powerful processor can be used that allows a very high sampling rate (e.g. $\frac{\omega_s}{\omega_b} > 40$). For cost-sensitive applications, the sampling rate is selected as the lowest that meet all performance requirements: fast disturbance rejection, smooth time behavior, etc.

Although there exist applications that may benefit from a variable sampling period, we focus here on systems with constant sampling period. Furthermore, we only deal with continuous-time systems discretized via zero-order hold on the inputs, i.e. we consider discrete-time systems of the form $x_{k+1} = f(x_k, u_k)$. As a side note, when using very fast sampling rates an *incremental* model of the form $dx_k = f(x_k, u_k, T)$ [59] might be necessary for the closed-loop control of embedded systems [58].

2.1.2. Cyber-physical systems: real-time embedded systems

The term *embedded system* has different meanings in different contexts. In this work, we define an embedded system as a dedicated microprocessor-based system that controls a physical process. Recently, the term *cyber-physical systems* has been applied to describe embedded systems that control physical processes in real time [60, 61], possibly via a computer network. This term has also different scopes in different disciplines, and some authors emphasize the communication between embedded systems in the provided definition [62]. We summarize the main characteristics of cyber-physical systems that are relevant for later chapters.

Hardware

There exist a broad spectrum of microprocessors. We focus on devices at the bottom end of the spectrum in terms of computational capabilities, namely microcontrollers.

Even among microcontrollers there is a broad range of options. The CPU for example is driven by clock frequencies usually in the range of several tens to a few hundred megahertz. Another distinctive features is the word size, which is typically 8-bit, 16-bit or 32-bit. For computationally intensive applications, like the MPC algorithm, 32-bit microcontrollers are in general preferred. We briefly describe the most relevant hardware characteristics of microntrollers: the processor (or CPU) and the memory. In the case of the processor, we base our discussion on a popular family of 32-bit microprocessors: ARM Cortex-M. These state-of-the-art processors are nowadays in widespread use and are well suited for applications with complex data processing requirements like MPC [63, Ch. 1]. For a detailed description of this family of processors see [63].

We start by discussing the arithmetic capabilities of a microcontroller. Let us first consider the low-cost Cortex-M3 microprocessor. This CPU lacks a floating-point unit (FPU). This means that only integer arithmetic can be performed directly on hardware. Integer addition and multiplication can be executed in one clock cycle, whereas division needs several cycles. Furthermore, floating-point operations are emulated by software, with additions and multiplications being several times slower than their integer counterparts. Empirical evidence shows that fixed-point operations (add, multiply) are around 4 times faster than floating-point operations for the Cortex-M3 [25] as well as for the classical ARM-7TDMI microprocessor [64]. In contrast, the more expensive Cortex-M4 includes digital signal processing (DSP) capabilities and, optionally, a single-precision FPU. This allows to execute 32-bit floating-point arithmetic operations directly on hardware.

The next big component of a microcontroller is memory. We distinguish between two types of memory: program memory, and data memory. The former is a non-volatile storage typically based on flash technology. It is commonly between ten kilobytes and a few megabytes, and is sometimes colloquially called ROM (read-only memory). The latter is a volatile type of storage typically based on SRAM (static random-access memory), and has a capacity of one kilobyte to a few hundred kilobytes.

Real-time systems

We now introduce some concepts related to applications that process data in real-time. We take most of our definitions from [60]. A *real-time system* is one whose correctness is based not only on the correctness of the results of the computations but also on the

physical time at which these results are produced.

A *deadline* is the time by which a result must be produced. Deadlines can be classified in three types: firm, soft and hard. A firm deadline is characterized by the fact that a result is no longer useful after the deadline has passed. If the deadline is soft, a result that missed the deadline can still be used. A hard deadline is one that, if missed, failure or catastrophic consequences may result. It follows that any hard deadline is a firm deadline.

Similar as in the classification of deadlines, real-time systems are also separated in soft and hard real-time systems (also called safety-critical systems). The former is any real-time system with no hard deadlines, whereas the latter is a real-time system that must meet at least one hard deadline.

In this work, we consider digital control applications with firm deadlines. Depending on the application, a control system may be soft or hard real-time. In the case of soft real-time control systems, missing a deadline may result in degradation of performance.

Another important concept is that of *temporal determinism*, which means that the response time for any event is known. *Computational delay* is the time it takes the microprocessor to execute the control algorithm. The variation on the computational delay is called the *control jitter*. We consider an algorithm to be (temporally) deterministic, if the control jitter is much smaller than the computational delay.

Software

Within software we make the distinction between application, operating system and compiler. We provide loose definitions of these terms, adapted from [65, Ch. 1]. An *application* performs an user-defined task. A digital control algorithm is considered an application. It may be a state observer, a model predictive control algorithm, etc.

The *operating system* is in charge of interfacing the application with the hardware. A *real-time operating system* (RTOS) is additionally in charge, among other things, of task scheduling. The control application is one of the many tasks that run concurrently on the CPU. The RTOS scheduling algorithm guarantees, under certain conditions, that our control algorithm is executed periodically with a constant sampling period [65, 66].

In embedded applications the C programming language [39], or simply *C*, and C++ are commonly used. However, C presents certain advantages over C++, namely: for embedded hardware targets, object-oriented C++ may carry penalty on computational

performance and power consumption when compared to procedural C [67]. Furthermore, for embedded targets, C compilers are more common than C++ compilers.

We use the term *compiler* as a broad term to denote the process of translating the user source code into binary code executable by the microcontroller. Hence, in this context a compiler also includes a preprocessor, an assembler, and a linker. The source code refers to both, the application and the operating system. In embedded systems, an application is commonly written entirely in C or C++, whereas the OS is usually a mix of C/C++ and hardware specific assembler code.

Memory management

Although the topic of memory management is a complex one (see [65, Ch. 3]), here we reduce it to two simplified cases: dynamic and static memory allocation. A programming language typically offers only certain types of memory management functionality. Dynamic memory allocation is done during program execution. In C and C++ for instance, dynamic allocation is done with a call to the `malloc` function. In C++ the `new` operator can also be used to dynamically allocate memory. Furthermore, an RTOS is in charge of allocating memory for each task using the functionality provided by the base language.

The main advantage of dynamic allocation is that it offers flexibility. That is, the amount of memory required by certain types of data (e.g. arrays) does not need to be known at compile time. However, a dynamic memory allocation algorithm does not show temporal determinism [68, 69]. Therefore, its use is restricted in real-time systems. Moreover, the use of dynamic memory allocation is prohibited in critical systems according to the MISRA-C standard, [70]. Furthermore, some embedded C/C++ compilers do not implement dynamic memory allocation algorithms.

In contrast, static memory allocation is made at compile time. As its name implies, the amount of memory cannot be change during runtime. If the size of the data involved changes, the affected code needs to be recompiled. The main advantage is that there are no runtime penalties. Therefore, static memory allocation is commonly required in real-time applications.

2.2. On-line optimization

This section introduces some of the basic concepts of on-line optimization, which is fundamental for the implementation of MPC.

2.2.1. Basics of convex optimization

Before we proceed, let us state some required definitions. The following presentation is based on [31].

We consider an *optimization problem* of the form

$$\begin{aligned} \underset{y\in\mathbb{R}^{n_y}}{\text{minimize}} \;\; & f_0(y) \\ \text{subject to} \;\; & f_i(y) \leq 0, \text{ for all } i \in \mathcal{I}, \\ & f_i(y) = 0, \text{ for all } i \in \mathcal{E}, \end{aligned} \tag{2.2}$$

where the functions f_i are smooth real-valued functions in $\mathbb{R}^{n_y}$ and $\mathcal{I}$ and $\mathcal{E}$ are each a finite set of indices. The function $f_0 : \mathbb{R}^{n_y} \to \mathbb{R}$ is called the *cost function* or *objective function*. The vector $y \in \mathbb{R}^{n_y}$ is the *optimization variable*. The functions $f_i : \mathbb{R}^{n_y} \to \mathbb{R}, i \in \mathcal{I}$ and $f_i : \mathbb{R}^{n_y} \to \mathbb{R}, i \in \mathcal{E}$ are called *inequality* and *equality constraint functions*, respectively. A point y is *feasible* if it satisfies all constraints. Problem (2.2) is said to be feasible if there exist at least one feasible point. Otherwise it is called *infeasible*. The set containing all feasible points is called the *constraint set* or *feasible set*. The *optimal value* f_0^* of problem (2.2) is defined as:

$$f_0^* = \inf\{f_0(x) \,|\, f_i(x) \leq 0, \; \forall \; i \in \mathcal{I}, f_i(x) = 0, \; \forall \; i \in \mathcal{E}\}.$$

A *solution* y^* to problem (2.2), also called an *optimal point*, is a point that is feasible and for which $f_0(y^*) = f_0^*$ holds. A feasible point y is called ϵ-*suboptimal* if satisfies $f_0(y) \leq f_0^* + \epsilon$, for $\epsilon > 0$. We call ϵ the *suboptimality level*.

An optimization problem is called *convex* if the cost function is convex, the inequality constraint functions are convex, and the equality constraint functions are affine (i.e. $f_i(y) = D_i y - d_i$, for all $i \in \mathcal{E}$). This implies that the feasible set of a convex optimization problem is convex.

2.2.2. Quadratic programs

Of particular interest in this work are quadratic optimization problem (QP) of the form:

$$\begin{aligned} \underset{y\in\mathbb{R}^{n_y}}{\text{minimize}} \;& \frac{1}{2}y^\mathsf{T}Hy + g^\mathsf{T}y \\ \text{subject to} \;& Cy \leq c, \\ & Dy = d. \end{aligned} \tag{2.3}$$

With respect to (2.2) we have $f_0(y) = \frac{1}{2}y^\mathsf{T}Hy + g^\mathsf{T}y$, $f_i(y) = D_iy - d_i$, for all $i \in \mathcal{E}$, and $f_i(y) = C_iy - c_i$, $i \in \mathcal{I}$. If we replace the affine inequality constraints in (2.3) with quadratic constraints we obtain the problem:

$$\begin{aligned} \underset{y\in\mathbb{R}^{n_y}}{\text{minimize}} \;& \frac{1}{2}y^\mathsf{T}Hy + g^\mathsf{T}y \\ \text{subject to} \;& \frac{1}{2}y^\mathsf{T}P_iy + q_i^\mathsf{T}y \leq r_i, \text{ for all } i \in \mathcal{I}, \\ & Dy = d, \end{aligned}$$

with P_i symmetric positive semi-definite, the problem is called a *quadratically constrained quadratic program* (QCQP). This type of problem arises in various MPC setups, see for example [71, 72]. A related problem is the *second-order cone program* (SOCP) of the form:

$$\begin{aligned} \underset{y\in\mathbb{R}^{n_y}}{\text{minimize}} \;& \frac{1}{2}y^\mathsf{T}Hy + g^\mathsf{T}y \\ \text{subject to} \;& \|\bar{P}_iy + \bar{p}_i\|_2 \leq \bar{q}_i^\mathsf{T}y + \bar{r}_i, \text{ for all } i \in \mathcal{I}, \\ & Dy = d. \end{aligned}$$

SOCPs arise in many engineering applications [73], including stochastic MPC problems [53, 74, 75]. We are mainly interested in QPs without equality constraints, which in standard form are represented as:

$$\begin{aligned} \underset{y\in\mathbb{R}^{n_y}}{\text{minimize}} \;& \frac{1}{2}y^\mathsf{T}Hy + g^\mathsf{T}y \\ \text{subject to} \;& Cy \leq c. \end{aligned} \tag{2.4}$$

In general, a QP is convex if and only if the Hessian matrix H is symmetric positive semi-definite. Furthermore, we focus on *strictly convex* QPs. A QP is called strictly convex if and only if $H^\mathsf{T} = H > 0$. Strict convexity implies that if a solution y^* exists

then it is unique. Additionally, strict convexity is an important property exploited by many optimization algorithms, and in particular by the method discussed in Section 3.3.

2.2.3. Optimization theory for quadratic programs

One important concept is that of duality, since it commonly exploited by optimization algorithms. We start by defining the *Lagrangian function* of problem (2.2), which is called the *primal* problem, as

$$\mathcal{L}(y, \lambda) = f_0(y) + \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i f_i(y), \tag{2.5}$$

where λ denotes a Lagrange multiplier vector. Then, the *Lagrange dual function* is defined as the minimum value of the Lagrangian over y for some fixed λ,

$$g_0(\lambda) = \inf_{y \in \mathcal{D}_y} \mathcal{L}(y, \lambda) = f_0(y) + \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i f_i(y). \tag{2.6}$$

One important characteristic of the dual function is that it is concave, even if problem (2.2) is not convex, [31, Ch. 5]. The *Lagrange dual problem* for problem (2.2) is defined as

$$\begin{aligned} &\text{maximize} \;\; g_0(\lambda) \\ &\text{subject to} \;\; \lambda_i \geq 0, \; i \in \mathcal{I}, \end{aligned} \tag{2.7}$$

which is itself a convex optimization problem. We say that *strong duality* holds if the optimal value of dual problem g_0^* equals the optimal value of the primal problem f_0^*. Duality is important since, for a strictly convex quadratic program (2.8) (for which strong duality holds), the solution to the primal problem can be readily found by solving the dual problem. Many optimization algorithms solve directly the primal problem, others solve the dual problem instead to find the solution of the primal, or exploit information of both primal and dual problems to find a solution.

We are interested in quadratic programs of the form (2.4). If the gradient and constraint vector depend on a parameter $w \in \mathbb{R}^{n_w}$, we have the parametric problem $\mathbb{P}(w)$:

$$\begin{aligned} \underset{y \in \mathbb{R}^{n_y}}{\text{minimize}} \;\; & \frac{1}{2} y^\intercal H y + g(w)^\intercal y \\ \text{subject to} \;\; & Cy \leq c(w), \end{aligned} \tag{2.8}$$

with $c(w) \in \mathbb{R}^{n_c}$. In MPC applications, this parameter is commonly given by (a combination of) the current state vector x, a reference trajectory, known disturbances. For notational simplicity and where no confusion may arise, the dependency on the parameter w will be dropped.

For our further discussion, we need a few further definitions.

Definition 1 (Feasible set)**.** *For the quadratic program $\mathbb{P}(w)$ the* feasible set *is given by*

$$\mathcal{F}(w) = \{y \in \mathbb{R}^{n_y} \mid Cy \leq c(w)\}. \tag{2.9}$$

Definition 2 (Active set)**.** *Consider problem $\mathbb{P}(w)$. A constraint $C_i y \leq c_i(w)$, $1 \leq i \leq n_c$ is called* active *at $y \in \mathcal{F}(w) \neq \emptyset$ if $C_i y = c_i(w)$ holds, and* inactive *otherwise. The set of indices*

$$\mathcal{A}(y; w) = \{i \in \{1, \ldots, n_c\} \mid C_i y = c_i(w)\} \tag{2.10}$$

is called set of active constraints *or simply* active set*. If y^* is the solution to $\mathbb{P}(w)$ we call $\mathcal{A}(y^*; w)$* optimal active set*.*

Definition 3 (LICQ)**.** *Given the pair (y, w), the constraints $\hat{c}_i(y; w) = C_i y - c_i(w)$, and the active set $\mathcal{A}(y; w)$, we say that the* linear independence constraint qualification (LICQ) *holds if the set of active constraint gradients $\{\nabla \hat{c}_i(y; w), i \in \mathcal{A}(y; w)\}$, is linearly independent.*

Now we are ready to introduce the first-order optimality conditions, also known as Karush-Kuhn-Tucker (KKT) conditions (taken from [29, Theorem 12.1]), which characterize an optimal point.

Theorem 1 (First-order optimality conditions)**.** *Suppose that y^* is a local solution to (2.2), that the functions f_i in (2.2) are continuously differentiable, and that the LICQ holds at y^*. Then there is a Lagrange multiplier vector λ^*, with components λ_i^*, $i \in \mathcal{E} \cup \mathcal{I}$, such that the following conditions are satisfied at (y^*, λ^*):*

$$\begin{aligned} \nabla \mathcal{L}(y^*, \lambda^*) &= 0, \\ f_i(y^*) &\leq 0, \quad \forall\, i \in \mathcal{I}, \\ f_i(y^*) &= 0, \quad \forall\, i \in \mathcal{E}, \\ \lambda_i^* &\geq 0, \quad \forall\, i \in \mathcal{I}, \\ \lambda_i^* f_i(y^*) &= 0, \quad \forall\, i \in \mathcal{I} \cup \mathcal{E}. \end{aligned} \tag{2.11}$$

A proof can for example be found in [29, Ch. 12].

For convex problems like (2.8), the KKT conditions provide sufficient and necessary conditions for a pair (y^*, λ^*) to be optimal. The point y^* is the unique minimizer if the problem is strictly convex. Furthermore, if the LICQ holds, for any given y^* the optimal Lagrange multiplier λ^* is also unique.

For the discussion of the active set optimization algorithm and the explicit MPC method in Chapter 3, the following definitions are needed.

Definition 4 (Set of feasible parameters)**.** *The* set of feasible parameters *of a parametric quadratic program is given by*

$$\mathcal{W} = \{w \in \mathbb{R}^{n_p} \mid \mathcal{F}(w) \neq \emptyset\}. \tag{2.12}$$

Definition 5 (Critical region)**.** *Let a strictly convex quadratic program $\mathbb{P}(w)$ with $w \in \mathcal{W}$ be given. Furthermore, let $y^*(w)$ denote the solution and $\mathcal{A}(y^*(w); w)$ the corresponding optimal active set. Then, for every index set $\mathcal{A} \subseteq \{1, \ldots, n_c\}$, the* critical region *of $\mathcal{W}$ is given by*

$$CR_{\mathcal{A}} = \{w \in \mathcal{W} \mid \mathcal{A}(y^*(w); w) = \mathcal{A}\}. \tag{2.13}$$

2.3. Model predictive control for linear systems

We are ready to introduce the model predictive control algorithm. We consider the linear MPC setup of the form (1.1). Figure 2.2 depicts the behavior of a system being controlled by a discrete MPC implementation. At time $t_k = kT$, $k \in \mathbb{Z}$ and T the sampling period, we estimate the state $\hat{x}_k$ from measurements $\hat{y}_k$, typically using a state observer. The MPC algorithm uses $\hat{x}_k$ and the linear model of the system to *predict* the future trajectory of the state $\mathbf{x}_k$ for the following N steps. With this information, a minimization problem is solved to find the input sequence $\mathbf{u}_k$ that optimizes a given performance criterion for the given horizon N. In the figure, the continuous function $\phi(t, \hat{x}_k, \mathbf{u}_k)$ (shown in dashed red) represents the *nominal* state evolution starting at $\hat{x}_k$ under the influence of the input $\mathbf{u}_k$. The predictive state trajectory $\mathbf{x}_k$ coincides with $\phi(t, \hat{x}_k, \mathbf{u}_k)$ at each sampling time t_k (represented by black dots over $\phi(t_k, \cdot)$). The discrete-time MPC algorithm has no knowledge of $\phi(\cdot)$, though. We have included this continuous time plot in the figure to simplify explanation of the concept. Furthermore,

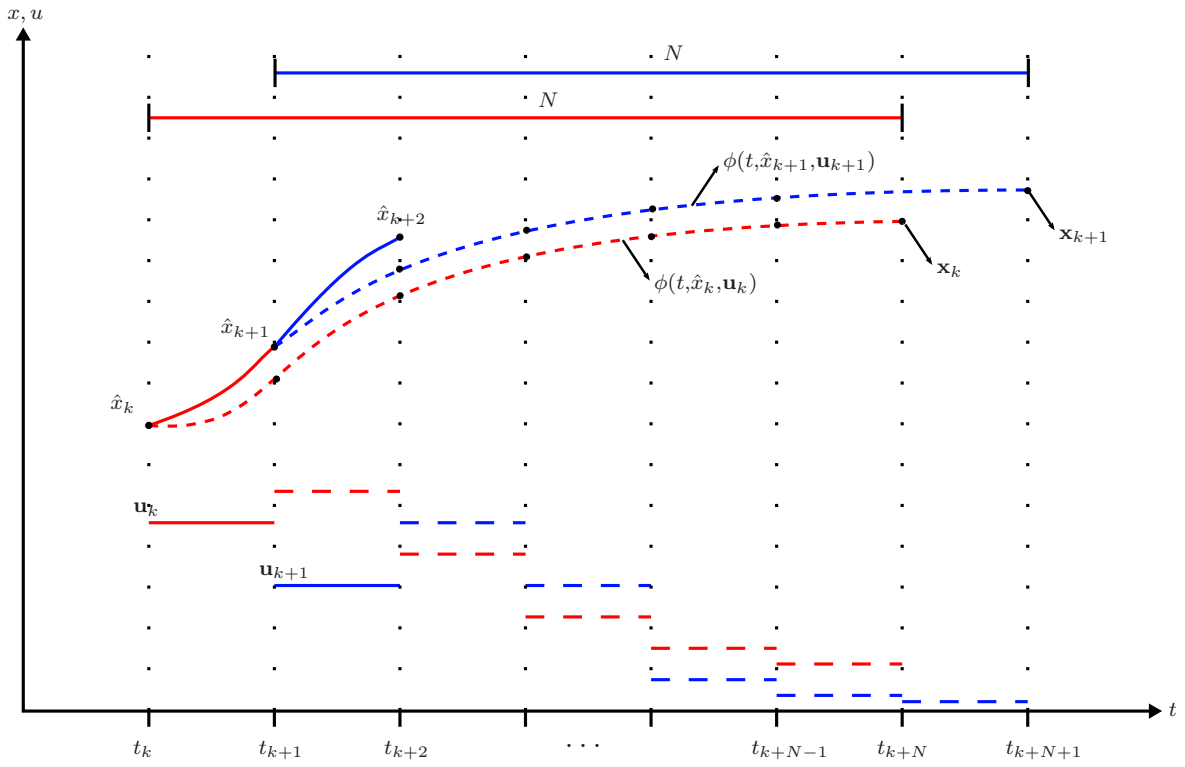

Figure 2.2.: A visual representation of the MPC algorithm.

$\mathbf{u}_k$ is held constant at each sampling interval (a zero-order hold), as is common for digital controllers [7].

Ideally, the state trajectory of the controlled system would follow the trajectory $\phi(t, \hat{x}_k, \mathbf{u}_k)$. However, due to several effects which are often not included in the MPC model (disturbances, model-plant mismatch, errors in the state estimation, quantization of the input, etc.), the real system follows a slightly different trajectory under the influence $\mathbf{u}_k$ during the interval from t_k to t_{k+1} (shown in solid red).

At the next sampling period t_{k+1}, we repeat the process. We first take a new measurement $\hat{y}_{k+1}$ and estimate $\hat{x}_{k+1}$. We then compute the input sequence $\mathbf{u}_{k+1}$ (dashed blue) for the next following N steps, and apply the first part of $\mathbf{u}_{k+1}$ (solid blue) to the real system during the current sampling interval, and so on for the subsequent sampling times t_k, $k = 2, 3 \ldots$

To simplify our discussion, we focus on one particular moving horizon setup, namely regulation to the origin of a constrained linear system. We describe this sample setup in the following.

2.3.1. A basic MPC setup

We considered discrete-time linear time-invariant systems subject to input and state constraints described by

$$x^+ = Ax + Bu, \tag{2.14}$$

where $x \in X \subseteq \mathbb{R}^n$, and $u \in U \subset \mathbb{R}^m$ are the system state and input at the current sampling time, respectively. The successor state is denoted by x^+. The considered input constraint set U is a convex and compact box set containing the origin. Moreover, the state constraint set X is a closed (not necessarily bounded) convex set with the origin in its interior. These type of constraints are commonly found in practice (see [3, Ch. 1]).

The discrete-time system and input matrices are represented by A and B, respectively. We assume the pair (A, B) to be stabilizable. The control objective is to bring the system to a desired equilibrium point while satisfying all constraints. For simplicity and without loss of generality, in this section we only consider regulation to the origin.

MPC is a natural candidate to solve the constrained regulation problem. The MPC controller needs to solve at each sampling time k an optimization problem parameterized by the current state x. This problem is, for the conditions given above, defined as follows:

$$\begin{aligned}
\underset{\mathbf{u}}{\text{minimize}} \quad & \frac{1}{2}\sum_{j=0}^{N-1}(\|x_j\|_Q^2 + \|u_j\|_R^2) + \frac{1}{2}\|x_N\|_P^2 & & \\
\text{subject to} \quad & x_{j+1} = Ax_j + Bu_j, & & j = 0, ..., N-1, \\
& \underline{u} \leq u_j \leq \overline{u}, & & j = 0, ..., N-1, \\
& \underline{e} \leq E_x x_j + E_u u_j \leq \overline{e}, & & j = 0, ..., N-1, \\
& \underline{f} \leq F x_N \leq \overline{f}, & & \\
& x_0 = x, & &
\end{aligned} \tag{2.15}$$

where $N \geq 2$ is an integer denoting the prediction horizon. We consider positive definite quadratic forms $\|w\|_S^2 = w^\intercal S w > 0$ and $\|w\|_S^2 = 0$ only if $w = 0$, with S a symmetric positive definite matrix. The matrices Q, R and P are symmetric positive definite and denote the states, input and terminal weighting matrices, respectively. If Q is symmetric positive semi-definite, an additional requirement for stability is needed, namely that the pair (A, Q) is detectable [3, Ch. 2]. The matrices Q and R, loosely speaking, are a

specification of the desired controller behavior. The term $\ell(x_j, u_j) = \frac{1}{2}(\|x_j\|_Q^2 + \|u_j\|_R^2)$ is called *stage cost*, whereas $\ell_N(x_N) = \frac{1}{2}\|x_N\|_P^2$ is called *terminal cost*. Furthermore, the box set U is defined by the lower and upper bounds $\underline{u}$ and $\overline{u}$. Similarly, the state constraints X are defined by the bounds $\underline{e}$ and $\overline{e}$, and the matrices $E_x \in \mathbb{R}^{q\times n}$ and $E_u \in \mathbb{R}^{q\times m}$. The *terminal state constraint set* $X_f \subseteq X$ is a polytope defined by the triplet $\underline{f}$, $\overline{f}$ and $F \in \mathbb{R}^{r\times n}$. This set, together with the cost function, is often used to guarantee closed-loop stability [3, Ch. 2].

We optimize over the input sequence $\mathbf{u} = \{u_0, \ldots, u_{N-1}\} \in \mathcal{U} = U^N$. The state constraints further impose the constraints $\mathbf{u} \in \mathcal{U}_N(x) = \{\mathbf{u} \in \mathcal{U} \mid \mathbf{x}(x, \mathbf{u}) \in \mathcal{X}\}$. The state sequence $\mathbf{x}(x, \mathbf{u}) \in \mathbb{R}^{(N+1)n}$ describes the trajectory followed by the states of system (2.14) starting at x and under the input sequence $\mathbf{u}$, i.e. $\mathbf{x}(x, \mathbf{u}) = \{x_0 = x, x_1 = Ax_0 + Bu_0, \ldots, x_N = Ax_{N-1} + Bu_{N-1}\}$. The set $\mathcal{X} = \{\mathbf{x} \in \mathbb{R}^{(N+1)n} \mid x_k \in X, k = 0, \ldots, N-1, x_N \in X_f\}$ denotes the state trajectories satisfying all constraints. Thus, $\mathcal{U}_N(x)$ is the set of feasible input sequences of problem (2.15) for a given x.

In an MPC scheme, the first element of the optimal input sequence $\mathbf{u}^*$ is used as feedback control action, i.e. during each sampling period $u = u_0^*$ is applied to the plant. In many applications, it is safe to assume that the input sequence found by optimization algorithms is indeed $\mathbf{u}^*$. However, in the case of embedded systems with fast sampling rate, the MPC controller may only have enough time to compute $\mathbf{u}^{\sim} \in \mathcal{U}_N(x)$, a (possibly rough) approximation to $\mathbf{u}^*$.

2.3.2. MPC as a QP

Most implementations of optimization algorithms cannot directly solve problems like (2.15). Therefore, the MPC problem needs to be brought into a form that can be directly used together with a tailored solver.

The MPC optimization problem (2.15) can be brought into the following equivalent form (refer to Appendix A)

$$\begin{aligned} \underset{\mathbf{u}\in\mathbb{R}^{Nm}}{\text{minimize}} \quad & \frac{1}{2}\mathbf{u}^\mathsf{T} H \mathbf{u} + g(x)^\mathsf{T}\mathbf{u} \\ \text{subject to} \quad & C\mathbf{u} \leq c(x). \end{aligned} \tag{2.16}$$

Compared to (2.4), we see that the *gradient vector* $g(x)$ is now given by a function $g : \mathbb{R}^n \to \mathbb{R}^{Nm}$. Similarly, the inequality constraint functions also depend on x, in

particular $c : \mathbb{R}^n \to \mathbb{R}^{2((m+q)N+r)}$. Problem (2.16) is known as *parametric QP*, and is parameterized by the variable x. Note that this problem does not include equality constraints (i.e. $\mathcal{E} = \emptyset$).

There are several ways to guarantee strict convexity of (2.16). In the case of setpoint stabilization (2.15), selecting the (symmetric) weighting matrices such that $Q \geq 0$, $P \geq 0$ and $R > 0$ guarantees that the Hessian matrix is symmetric positive definite.

A MPC problem like (2.16) can be reliably solved by many commercial (e.g. MATLAB, Gurobi) and non-commercial (e.g. CVXOPT [76], OpenOpt [77]) optimization packages [78]. Nevertheless, to efficiently apply MPC on embedded systems, a solver needs to exploit the structure of problems like (2.16) that are derived from (2.15). Next chapter addresses this topic.

2.3.3. General moving horizon control formulation

So far, we have only consider one particular type of MPC problem. However, one of the strengths of moving horizon formulation is that is general and therefore can deal with a great variety of problems. Nevertheless, we focus our attention on problems that can be equivalently expressed as a parametric quadratic program $\mathbb{P}(\mathbf{p})$ of the form:

$$\begin{aligned} \underset{\mathbf{u}}{\text{minimize}} \quad & \frac{1}{2}\mathbf{u}^\top H \mathbf{u} + g(\mathbf{p})^\top \mathbf{u} \\ \text{subject to} \quad & C\mathbf{u} \leq c(\mathbf{p}), \end{aligned} \tag{2.17}$$

with H and C constant matrices, and $g(\mathbf{p})$ and $c(\mathbf{p})$ vectors affine in each of the parameters of the parameter sequence $\mathbf{p}$.

Note that (2.16) is the standard QP representation the MPC setup (2.15). This particular problem (regulation to the origin subject to constraints) has the state vector x as its only parameter. That is, (2.16) is a particular case of (2.17) in which $\mathbf{p}$ consist of a single parameter x, i.e. $\mathbf{p} = \{x\}$. In general, $\mathbf{p}$ may consist of several parameters that depend on the MPC problem at hand. Thus, formulation (2.17) allows us to consider a broad range of problems, including, but not limited to: setpoint stabilization, trajectory tracking [79, 80], systems with dead time [81], and systems with known-ahead disturbances [11, 82]. Notable systems that cannot be expressed in form (2.17) are time-varying systems, nonlinear systems, and formulations with weighting matrices varying with time.

Require: At each sampling time $k = 0, 1, \ldots$ get the parameter sequence $\mathbf{p}_k$
1: Find the solution $\mathbf{u}^*$ to problem (2.17) for $\mathbf{p} = \mathbf{p}_k$
2: Assign the MPC control action $u_k = \mathbf{u}_0^*$
3: **return** u_k

Algorithm 1: Model predictive control algorithm

The MPC scheme that was intuitively described in Figure 2.2 is formally presented in Algorithm algorithm 1.

2.4. Summary

We presented a general description of the characteristics of the problems we are interested in. We introduced digital control of embedded systems and outlined important concepts that are relevant for MPC applications. We focused first on the limitations of embedded computers. We then introduced the model predictive control algorithm for linear systems, and showed that it is, under certain conditions, equivalent to a convex optimization problem. We presented the theoretical foundations of convex optimization which are needed in the following chapters. In particular, we discussed convex quadratic programs, a type of optimization problem commonly arising in linear MPC. In the next chapter we discuss how problems like (2.17) can be solved efficiently, while at the same time taking the limitations of embedded systems into account.

3. Tailored on-line optimization software tools for MPC

This chapter discusses how optimization algorithms can be tailored for efficient MPC implementations. We start by describing the main properties of the MPC algorithm that need to be taken into account. It follows the discussion on how some of these properties are exploited by state-of-the-art tailored optimization software tools. We then present a novel optimization algorithm that takes into account the properties of MPC and is well suited for embedded system. This allows the algorithm to achieve good controller performance with the limited computational resources available on embedded hardware platforms.

3.1. Exploiting the properties of the MPC algorithm

In this section we discuss the most relevant properties of the MPC algorithm that need to be taken into account in embedded systems. These properties can be used to develop efficient MPC implementations. We discuss some methods that work well in practice. From a theoretical perspective, however, they present some challenges. We, therefore, emphasize in the following the practical implications of these ideas, while we only briefly mention some theoretical aspects.

3.1.1. Known-ahead maximum computation time

As discussed in Section 2.1, one fundamental assumption that is commonly made in digital control systems is that the sampling period is constant. This imposes a limit on what the maximum computational time of the MPC algorithm can be. Because an MPC scheme runs alongside with several other software components on the same processor, the exact available computational time for the MPC can in most cases only determined

empirically. Nevertheless, an important requirement of each of the software components involved is that they have a deterministic computational time. It is therefore desirable to have a known maximum computation time of Algorithm 1, and in particular of the optimization algorithm used to solve QP (2.17).

Typically these problems are solved by iterative numerical algorithms that compute a ϵ-suboptimal point (refer to Section 2.2). The suboptimality level ϵ determines the number of iterations the algorithm needs to perform. The number of iterations is in turn limited by the available computation time. For embedded systems, where the available computing time is usually the most restricting criterion, it might only be possible to find rough (and perhaps even infeasible) approximations to $\mathbf{u}^*$. In many situations, however, these approximations have been observed to provide a surprisingly good close-loop performance, see [32, 64], and the examples in Chapter 6. Tailored algorithms, like the ones discussed later in this chapter, can usually find good approximations rather quickly.

Many algorithms have theoretically deterministic execution times for any given number of iterations. Nevertheless, their software implementation may render the algorithm non-deterministic. In particular, some implementation may rely on dynamic memory allocation, or on transcendental functions like logarithms or square roots. These operations are usually implemented on a compiler or math library and are typically non-deterministic. While this may not be an issue if enough computation time is available, it could make the algorithm not suited for a real-time embedded MPC implementation.

It is worth mentioning that many theoretical developments rely on the assumption that the QPs are solved exactly. These discrepancies between the solution $\mathbf{u}^*$ and an approximation $\mathbf{u}^{\sim}$ can be considered as perturbations if they are small enough. In this context, small enough implies that the observed closed-loop performance is acceptable. Expensive high-precision applications may have different performance requirements than a cost-sensitive microcontroller application. Examples of the latter case are presented in Chapter 6.

3.1.2. Partial use of the solution by the controller

In general, in any MPC scheme only part of the computed solution $\mathbf{u}^*$ is used as control input. Typically, only the first vector element $\mathbf{u}_0^*$ of the solution sequence is applied to the dynamical system. The rest of the sequence is simply discarded. If we now take

into account the limited computation time available, we are only able to to compute the approximate solution $\mathbf{u}^{\sim}$, from which we apply the first element $\mathbf{u}_0^{\sim}$. This is repeated at each sampling time. It should be clear from this that the actual performance of the closed-loop control depends only on $\mathbf{u}_0^{\sim}$, and only indirectly on how good an approximation the whole sequence $\mathbf{u}^{\sim}$ is.

Being more precise, for any given trajectory consisting of T_N points, the performance of the approximate closed-loop trajectory can be quantified by the cost

$$J_p = \sum_{k=0}^{T_N-1} \|x_k^* - x_k^{\sim}\|_Q^2 + \|u_k^* - u_k^{\sim}\|_R^2, \tag{3.1}$$

where x_k^* and u_k^* are the state and input at point k in an optimal MPC closed-loop trajectory. Similarly, $x_k^{\sim}$ and $u_k^{\sim}$ are the state and input at point k in a closed-loop trajectory based on suboptimal MPC. A lower value of (3.1) indicates a better nominal performance.

This criterion allows us to compare the nominal closed-loop performance reached by different QP solvers under tight time constraints, as is usually the case in embedded systems. In other words, given an upper limit on the computation time, the solver that provides the lowest J_p might be preferred for the MPC controller. Alternatively, given an upper limit on J_p (whose value depends on the application), the solver that requires the least computational effort would be preferred.

3.1.3. Similarities between consecutive problems

So far, we have only consider the implementation aspects of MPC from the perspective of the controller, i.e., we only use the first part of the approximate solution as a control input, while discarding the rest. From the perspective of the optimization algorithm, the complete approximate solution is key for improving the solutions in the following sampling instances, and, therefore, the controller performance. This is based on the fact that many optimization algorithms, and in particular all of the algorithms discussed in this chapter, work iteratively. They start their first iteration with an initial guess of the solution, and with consecutive iterations they converge to the solution $\mathbf{u}^*$. Therefore, it is of great importance to choose a good initial guess, i.e., one that gets the algorithm closer to $\mathbf{u}^*$ in less iterations. The use of a previous solution to compute an initial guess for the current problem is known as *warm start* ([3, Ch. 2]).

Although several methods exits to provide an optimization algorithm with a good initial guess, we focus here on the *shift method* ([4, Ch. 10]). The shift method can also be used to find initial guesses of the Lagrange multipliers for some algorithms. This method is commonly used in applications where there exist a time similarity between any two consecutive MPC problems. Let us explain with an example what is meant by time similarity. Consider the MPC setup (2.15), and its equivalent QP form (2.16) (denoted by $\mathbb{P}(\mathbf{p})$ for the general case (2.17)). For any two consecutive sampling times t_{k1} and t_{k2} we have the states x_{k1} and $x_{k2} = Ax_{k1} + Bu^*_{k1}$ as parameters. Here, u^*_{k1} is the first part of the solution to $\mathbb{P}(x_{k1})$. In this case, the MPC problems $\mathbb{P}(x_{k1})$ and $\mathbb{P}(x_{k2})$ have a time similarity.

In general, the shift method works as follows: it considers the approximate solution sequence $\mathbf{u}^{\sim} = \{u_0^{\sim}, u_1^{\sim}, \ldots, u_{N-1}^{\sim}\}$ (computed at the previous sampling interval), and from it we compute the initial guess for the current sampling interval:

$$\mathbf{u}^0 = \{u_1^{\sim}, \ldots, u_{N-1}^{\sim}, u_{N-1}^0\}, \tag{3.2}$$

with the term u_{N-1}^0 to be chosen.

The simplest choices are $u_{N-1}^0 = u_{N-1}^{\sim}$, or $u_{N-1}^0 = 0$. However, this might result in an infeasible initial guess $\mathbf{u}^0$. A more complex value that guarantees a feasible initial guess can be computed by finding the solution $u^*_{N,1}$ for the MPC problem with $N = 1$ and $x_0 = x_N$, and setting $u_{N-1}^{\sim} = u^*_{N,1}$. Finally, the control action of a linear quadratic regulator can also be used as $u_{N-1}^{\sim}$. This guarantees nominal stability of the MPC problem under particular conditions, see [3] for details.

For embedded applications, when a warm start is used, any of the simplest approaches of the shift method might be preferred, i.e. choosing $u_{N-1}^0 = u_{N-1}^{\sim}$, or $u_{N-1}^0 = 0$. In these simple cases the shift method relies only on copying data, i.e. it does not involve any arithmetic operations. This makes it particularly attractive for embedded applications, as it is easy to implement and the computational requirements are relatively small. The theoretical implications of using an infeasible initial guess are beyond the scope of this work. Nevertheless, using $u_{N-1}^0 = 0$ works well in practice for the algorithm presented in Section 3.3.

The shift method in general provides good initial guesses only for certain families of problems, e.g. gradient methods and active set methods. A notable exception are primal-dual interior point methods, which require an initial point satisfying certain char-

acteristic that a previous solution does not typically fulfill, [20]. It is worth mentioning that, in some applications, using a single predetermined sequence as initial guess at every sampling instance may deliver better performance (see [15]). This way of proceeding is known as *cold starting* the MPC algorithm.

3.1.4. Special structure of the data

The matrices that form the cost function and the constraints of QPs arising from MPC have particular structures that can be exploited by tailored algorithms.

Hessian matrix

As outlined, there are two common ways to express the general MPC problem into a standard QP form: *sparse* and *condensed* formulation.

In the sparse QP formulation the Hessian typically has a block tridiagonal structure. Tailored algorithms can exploit this structure in a way that makes the computational cost only slightly higher than $O(N)$ [20, 83], with N being the horizon length.

In a condensed QP, the resulting Hessian matrix is dense and does not show a directly exploitable structure. For optimization algorithms like active set and interior-point methods, the computational cost (in terms of memory requirements and arithmetic operations) for obtaining a solution using a dense Hessian is often proportional to N^3. For gradient-based methods, the computational complexity is proportional to N^2. Appendix A exemplifies how to form a condensed QP for the MPC problem (2.15).

In general, the condensed QP formulation is preferred when short horizons are used. For very long horizons, the quadratic (or cubic) dependence of the complexity on N makes the use of the condensed formulation impractical. What represents a short or long horizon is application dependent. In other words, the decision of whether to use a sparse or condensed formulation is to be made on a case-by-case basis. A rule of thumb states that a condensed formulation is preferred if the number of states is much larger than the number of inputs and the horizon is not too long (say less than 30 steps) [7]. In the case of stochastic MPC formulations based on polynomial chaos expansion [84], a condensed formulation allows the efficient implementation on a microcontroller of, for example, a stochastic MPC problem with 30 states, 1 input and a horizon length of 5 steps [21].

In embedded applications, the horizon length is often selected to be as short as possible, while still delivering suitable performance, thus condensed formulations can be of computational advantage due to lower memory and CPU requirements. Furthermore, the condition number of the Hessian increases if the horizon length is increased [36, Corollary 1]. A Hessian with a low condition number is in general desired. Furthermore, in many embedded applications a well conditioned Hessian may greatly improve the overall algorithm performance (cf. Subsection 3.3.4). Therefore, in the following, we focus on condensed QPs formulations.

Constraints

The constraint matrices can also be treated as condensed or sparse. Furthermore, as mentioned in Subsection 2.3.1, in MPC applications a strict separation can be made between input constraints, $u \in U$ and state constraints $x \in X$. Very frequently, the input constraint set is defined as $U = \{u \in \mathbb{R}^m \mid \underline{u} \leq u \leq \overline{u}\}$, and the state constraint set is given by $X = \{x \in \mathbb{R}^n \mid \underline{e} \leq E_x x + E_u u \leq \overline{e}\}$. Take for example MPC problem (2.15), which can be concisely expressed as the QP (2.17). The constraint $C\mathbf{u} \leq c(\mathbf{p})$ has the matrices with the following structure

$$C = \begin{bmatrix} I \\ -I \\ \hat{C} \\ -\hat{C} \end{bmatrix}, \; c(\mathbf{p}) = \begin{bmatrix} \overline{\mathbf{u}} \\ \underline{\mathbf{u}} \\ \overline{\hat{c}}(\mathbf{p}) \\ -\underline{\hat{c}}(\mathbf{p}) \end{bmatrix}, \tag{3.3}$$

which can be split into the box constraints $\underline{\mathbf{u}} \leq \mathbf{u} \leq \overline{\mathbf{u}}$ which are in this case parameter independent, and the more general $\underline{\hat{c}}(\mathbf{p}) \leq \hat{C}\mathbf{u} \leq \overline{\hat{c}}(\mathbf{p})$. Taking into account this particular structure can significantly speed up an optimization algorithm.

3.1.5. Soft constraints

In general, in model predictive control the constraints on the states cannot be strictly enforced. They might be violated due to prediction horizons that are too short, or, as is common in real applications, due to differences between the nominal and actual closed-loop behavior. In the nominal case, full and exact state information is considered. Furthermore, a perfect model of the plant is assumed without (unknown) disturbances.

These conditions are not met in real applications, and may result in violation of state constraints. This implies that the optimization problem becomes infeasible, and cannot be solved. Obviously, this is an undesired on-line behavior that should be avoided. One approach is to replace the *hard* state constraints with *soft* constraints. The constraints in the inputs are left unchanged.

One way to implement soft constraints is to introduce slack variables $\mathbf{s} \geq 0$. The hard state constraints $\mathbf{x} \leq \overline{\mathbf{x}}$ is then rewritten as $\mathbf{x} \leq \overline{\mathbf{x}} + \mathbf{s}$. If there are no violations of the original constraints, then $\mathbf{s} = 0$. Otherwise, if constraint $\mathbf{x}_i$ is violated, then $\mathbf{s}_i > 0$. Furthermore, the cost function is modified to include a metric of the cost of any constraint violation $\mathbf{s}_i > 0$. There are several ways to modify the cost function. In MPC, it is preferable that the new cost function gives an *exact penalty* method. An exact penalty soft problem means that if the original hard problem is feasible, the same solution is obtained for the soft problem. In [85], the authors present an exact penalty that is a combination of ℓ_1-norms and squared ℓ_2-norms. In [86] a method is discussed to provide a lower bound on how heavy the infeasibilities are to be penalized in the cost function to guarantee an exact penalty method under certain conditions.

As stated above, the use of soft constraints avoids infeasibility of the parametric QP. An MPC implementation could explicitly include slack variables in the formulation. Alternatively, the QP solver could implicitly relax the state constraints. The algorithm described in Section 3.3 follows the latter approach.

3.2. Tailored optimization software tools

In this section, we shortly discuss several software tools of state-of-the-art algorithms tailored for MPC. We briefly describe the algorithms on which these tools are based, and provide a concise explanation on how each implementation exploits the MPC properties.

Before we continue, we repeat here for easier reference the optimization problem (2.2):

$$\begin{aligned} \underset{y \in \mathbb{R}^{n_y}}{\text{minimize}} \quad & f_0(y) \\ \text{subject to} \quad & f_i(y) \leq 0, \text{ for all } i \in \mathcal{I}, \\ & f_i(y) = 0, \text{ for all } i \in \mathcal{E}, \end{aligned} \tag{3.4}$$

and the first-order optimality (KKT) conditions (2.11):

$$\begin{aligned} \nabla \mathcal{L}(y^*, \lambda^*) &= 0, \\ f_i(y^*) &\leq 0, \quad \forall\, i \in \mathcal{I}, \\ f_i(y^*) &= 0, \quad \forall\, i \in \mathcal{E}, \\ \lambda_i^* &\geq 0, \quad \forall\, i \in \mathcal{I}, \\ \lambda_i^* f_i(y^*) &= 0, \quad \forall\, i \in \mathcal{I} \cup \mathcal{E}. \end{aligned} \tag{3.5}$$

3.2.1. Interior point methods

Interior point methods (IPM) can be broadly classified as primal or primal-dual. Tailored primal-dual methods often outperform primal barrier methods, in particular where very accurate solutions are required, [31, Ch. 11.7]. IPMs can deal with a broad variety of convex problems, including QPs, quadratically constrained quadratic programs (QCQP), and second-order cone programs (SOCP). One of the first IPMs tailored for MPC was presented in [20] based on a primal-dual approach. Tailored primal IPMs have been discussed in [32, 87]. Here, we briefly discuss primal-dual methods, in particular with Mehrotra predictor-corrector [88], as there are several recent relevant results for code generation of MPC algorithms using these methods. Of interest here are the code generating tools based on primal-dual IPMs for QPs [44], QCQPs [22, 23], and SOCPs [24, 46, 89].

A primal-dual IPM start with an initialization phase that searches for a point (y, λ) such that the optimization variable y satisfies all inequality constraints strictly (i.e. $f_i(y) < 0, i \in \mathcal{I}$) and the dual variable satisfies $\lambda > 0$. This step usually involves solving a linear system of equations.

Once this initialization point is found, a primal-dual search direction must be computed. This requires solving a linear system based on the linearization of the KKT conditions (3.5) (sometimes called Newton system). The resulting Newton system has the structure

$$\begin{bmatrix} \underline{S} & F^\top \\ F & 0 \end{bmatrix} \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}, \tag{3.6}$$

where $\underline{S}$ is symmetric positive semidefinite, and F has full row rank. At each iteration of the IPM method, the Newton system needs to be solved for two different right hand sides, i.e. it is solved once to find a scaling direction, and a second time to compute

the centering-plus-correction directions (see [44] for details). With the solutions of these two Newton systems, an updated pair of primal and dual variables can be computed.

The main computational burden of IPM lays in solving the Newton systems, which have the structure (3.6). Therefore, using methods that exploit their structure is advantageous. Several implementations of primal-dual IPM exist. We focus here on the software tool CVXGEN [44], FORCES, [22], and ECOS/QCML [24, 46].

CVXGEN is a parser-solver that uses a high-level description of convex optimizations problems (only QPs are considered at the time of this writing) to generate tailored *embedded* solvers based on IPMs. An embedded solver, in the CVXGEN context, is one that takes into account the inherited properties of a problem family (e.g. QPs with some particular features) to generate code that is self-contained. This in strong contrast to a general solver, which attempts to solve a broader class of problems (e.g. QPs) without any previous information about the particular problem instance. An embedded solver is usually orders of magnitude faster than a general solver. CVXGEN reports solving some small problems $10,000$ times faster than the general parser/solver CVX. The high-level problems description of CVXGEN makes it really easy to generate fast embedded solvers for MPC [90]. However, CVXGEN follows by design an explicit approach to code generation. This implies that the size of the generated code quickly grows with problem size [44]. This restricts CVXGEN to problems of moderate size, in general. For low-cost embedded systems, where there is little memory available (say a few 100 kB), CVXGEN might be further restricted to problems of small size [25].

FORCES is as well a code generation tool that uses a IPM similar to the one used by CVXGEN. However, FORCES code generation is less sensitive to problem size. Furthermore, FORCES explicitly takes into account the MPC problem structure. This can make FORCES several times faster than CVXGEN for MPC applications with long horizons [22]. In addition, FORCES can deal with QCQPs problems. QCQPs arise in various MPC setups, see for example [71, 72]. FORCES is a MATLAB based toolbox that, in contrast to CVXGEN, does not use a high-level language specification. The problem formulation of FORCES is very explicit. A simple MPC formulation that in CVXGEN would be around 10 lines, in FORCES takes around 100 lines. A commercial version of FORCES provides a simpler formulation.

A third code generation tool based on primal-dual IPMs for SOCPs is the combination ECOS/QCML. ECOS is a general purpose solver for small to medium size SOCPs. QCML is a parsing and code generating tool for ECOS. SOCPs arise in many engineering

applications [73], including stochastic MPC problems [53, 75].

In the context of MPC, one of the often cited drawbacks of primal-dual methods is their difficulty to be warm started (see [91, 92] and references therein). CVXGEN, FORCES and ECOS do not implement warm starting strategies. Thus, although very efficient, these solvers can be outperformed by other optimization methods in MPC applications that can fully exploit warm start. This can make a primal-dual IPM solver less effective for embedded applications with tight computational deadlines [25]. Next we present two algorithms that can be easily warm started, namely active set methods, and gradient based methods.

3.2.2. Active set methods

The basic idea behind active set methods is to fix a set of active constraints (called the *working* set) and solve a much easier equality constrained QP. Active constraints are added to or removed from the working set at each iteration of the algorithm until the optimal active set, and hence the solution to the original problem, is found. Active set methods can be broadly classified in primal and dual methods. We focus on the tailored dual approach called *online active set strategy* [27] and the implementation qpOASES [33]. We refer to [35] for a detail presentation of the online active set strategy.

Online active set strategy at a glance

Fundamental to the online active set strategy is the introduction of a homotopy to go from one critical region to another (refer to Section 2.2 for the definitions of critical region, and optimal active set). At each iteration, the online active set strategy method computes a step direction for the primal and dual variables by solving a KKT system (with a structure like (3.6)) based on the difference of a previous point y^0 (either the solution of a previous QP or a point from the previous iteration) and the optimal point y^* (see Figure 3.1).

From this, a maximum homotopy step length $\tau_{\max} \in [0, 1]$ can be computed, with $\tau_{\max} = \min\{1,\ \tau_{\max}^{\text{dual}},\ \tau_{\max}^{\text{prim}}\}$. From this, two possible actions are possible: If $\tau_{\max} = 1$, a solution has been found and the algorithm stops iterating, otherwise if $\tau_{\max} < 1$ the current working set needs to be modified and the procedure to find a new homotopy must be repeated. More specifically, if $\tau_{\max} = \tau_{\max}^{\text{dual}}$ a constraint needs to be removed

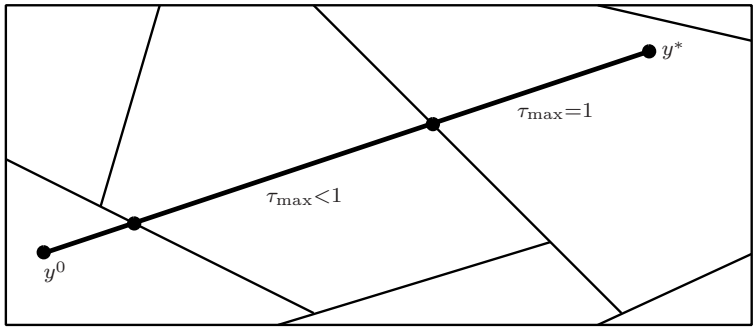

Figure 3.1.: Homotopy paths for a QP across multiple critical regions, adapted from [27].

from the working set. If $\tau_{\max} = \tau_{\max}^{\text{prim}}$ then a constraint must be added to the working set.

Active set methods in general require a feasible point as initial guess. One of the advantages of dual active set methods, with respect to primal ones, is that finding a dual feasible starting point is straightforward for strictly convex QPs.

Exploiting MPC properties in online active set methods

In general, the algorithm should stop iterating only if $\tau_{\max} = 1$, i.e. an optimal point y^* is found. In a MPC application, the algorithm may be stopped at a suboptimal point $y^{\sim}$ where $\tau_{\max} < 1$. Due to the homotopy, this points lies on the boundary of a critical region of the current QP. The online active set method can use this information for warm starting the QP at the next sampling time (see [35, Ch. 3]). Furthermore, this method uses a so-called *hot start* procedure that consist on warm starting plus reusing the matrix factorizations employed to solve the KKT system. These greatly speeds up the solutions of consecutive QPs. Furthermore, the maximum number of arithmetic operations are fixed for any given QP.

The online active set strategy has been implemented in *qpOASES* [33], an open-source software implemented using the C++ programming language. qpOASES has been successfully used in linear MPC applications directly or with slight modifications [35, Ch.5] as well as in nonlinear ones as a solver embedded in the ACADO toolkit [49].

qpOASES is a quite mature QP solver that exploits many of the properties of the MPC problem inherited from the active set strategy. It can be hot started and deals with input box constraints explicitly. qpOASES relies on dynamic memory allocation, i.e. it makes extensive use of the `new` operator. This, and the fact that is implemented in C++, may

pose several drawbacks for embedded applications. Some embedded C/C++ compilers do not implement dynamic memory allocation algorithms, which makes qpOASES unusable in such cases. Moreover, dynamic memory allocation is not deterministic. This last point implies that, although the online active set strategy itself has deterministic computation time, the use of dynamic memory allocation by qpOASES renders the implementation non-deterministic. This might not have a noticeable effect on applications with enough computational power. However, for MPC applications with tight real-time deadlines, this can have a significant impact in closed-loop performance due to missed deadlines. See the discussion in Subsection 2.1.2.

3.2.3. Gradient methods

In this section we focus on tailored algorithms based on gradient and Lagrange relaxation methods. We especially focus on the MATLAB toolbox FiOrdOs [48], an implementation that generates C-code based on this algorithm, is also discussed.

FiOrdOs directly the dual problem (2.7) for convex quadratic programs using first-order methods. To apply first-order methods the gradient of (2.7) must be computed. This in turn implies that the problem defined by the Lagrange dual function (2.6) are be solved. To achieve this, two optimization problems must be solved in a nested way. In this context, problem (2.7) is called outer problem and (2.6) is the inner problem. FiOrdOs implements classical gradient methods, and a fast gradient method (see Section 3.3 for a brief description of the fast gradient method). Furthermore, FiOrdOs allows to compute off-line a pre-conditioner of the Hessian by solving a semi-definite program. This can significantly improve the convergence rate of the fast gradient method (see Subsection 3.3.4). Furthermore, for certain problem structures, a certificate of sub-optimality can be provided given a maximum number of iterations of the fast gradient method [36, 47].

FiOrdOs takes into account the separation of input and state constraints. It considers problems where the optimization variable y belongs to a *simple set*, which is defined as the Cartesian product of *elementary simple sets*. An elementary simple set is one that is convex, closed and its projection is easy to compute. Examples of such sets are box sets, Euclidean balls, second-order cones, and simplexes. Restricting the optimization variable to such sets allows the efficient use of gradient methods.

Overall, FiOrdOs is a powerful code-generation tool that allows fine configuration for

the algorithms. However, the syntax is very technical and might be daunting for users of MPC not familiar with optimization theory.

3.2.4. Comments on explicit methods

An alternative approach to solving a parametric QP on-line is to find off-line the QP solutions for all possible parameters and recall the solution for a particular parameter on-line. This is often referred to as *explicit MPC*, c.f. [26]. The main idea is to use the critical regions of the parametric QP to form a polytopic representation of the solution (see Figure 3.1). Inside each critical region, the solution to the QP is given by an affine function of the parameter. Obtaining this representation, which is guaranteed to have a finite number of partitions, is typically done off-line. The on-line computation is then limited to identifying to which region the current parameter belongs to. This is done by means of a look-up table stored in memory. Once this region is identified, the affine function corresponding to this region is evaluated to find the optimal point.

A drawback of explicit methods is that number of regions grows quickly with problem size. Thus storing the resulting number of regions, and quickly selecting the optimal critical region, may present some challenges in embedded systems. There have been improvements with respect to these two aspects in the recent years, see [93] and references therein. Still, explicit methods are in general restricted to problems of small size. A typical MPC problem for which an explicit approach can be applied has one or two inputs, less than ten states, and up to four free control moves (i.e. a very short control horizon).

The main advantage of explicit methods is that for small problems, a solution can be found much quicker than on-line methods. Furthermore, parallelization can be easily exploited by, for example, field-programmable gate array technology [94]. In general, explicit methods are suitable for problems of small size with very fast sampling rates and where memory limitations are not a big concern.

3.3. A novel optimization algorithm for embedded MPC

We present a novel optimization algorithm specifically developed to exploit the properties of MPC (discussed in Section 2.1) and at the same time take into account the limitations of embedded sytems (presented in Section 3.1). The algorithm is based on an augmented

Lagrangian method combined with Nesterov's fast gradient method, [83, 95]. We start by describing the fast gradient method and the augmented Lagrangian method in a general setting. Afterwards we discuss how the two algorithms are combined and applied to efficiently solve MPC problems in embedded systems.

3.3.1. Fast gradient method

We briefly describe how the fast gradient method (FGM) [96] can be applied for solving a convex optimization problem of the form

$$\underset{y\in\mathcal{Y}}{\text{minimize}} \;\; f_0(y), \tag{3.7}$$

which is a particular case of (3.4). Here, the set $\mathcal{Y} \subset \mathbb{R}^{n_y}$ is convex and compact. In the following, we briefly present the main properties of the fast gradient method. This subsection is based on [97, Ch. 2], where a thorough treatment of the FGM can be found.

We first define some important concepts.

Definition 6 (Lipschitz continuity of gradient)**.** *The gradient of a continuously differentiable function $\nabla f_0(y)$ is called Lipschitz continuous on $\mathcal{Y}$ if there exists a Lipschitz constant $L \geq 0$ such that for any $y, z \in \mathcal{Y}$ we have*

$$\|\nabla f_0(z) - \nabla f_0(y)\| \leq L\|z - y\|. \tag{3.8}$$

Definition 7 (Strong convexity)**.** *A continuously differentiable function $f_0(y)$ is called strongly convex on $\mathcal{Y}$ if there exists a strong convexity parameter $\phi > 0$ such that for any $y, z \in \mathcal{Y}$ we have*

$$f_0(z) \geq f_0(y) + \nabla f_0^{\mathrm{T}}(y)(z - y) + \frac{\phi}{2}\|z - y\|^2. \tag{3.9}$$

Definition 8 (Projection onto a set)**.** *Let $\mathcal{Y}$ be a closed set and $z \in \mathbb{R}^{n_y}$. Then the* projection *of z onto $\mathcal{Y}$ is defined as*

$$\mathcal{P}_{\mathcal{Y}}(z) = \arg\min_{y\in\mathcal{Y}} \|y - z\|. \tag{3.10}$$

For a continuously differentiable function $f_0(y)$ whose gradient is Lipschitz continuous

the projection of the gradient step onto the feasible set $\mathcal{Y}$ is computed as follows:

$$\mathcal{G}_{\mathcal{Y}}(z) = \mathcal{P}_{\mathcal{Y}}(z - \frac{1}{L}\nabla f_0(z)). \tag{3.11}$$

We call $\mathcal{G}_{\mathcal{Y}}(z)$ the *projected gradient step* of f_0 onto $\mathcal{Y}$.

Fast gradient method for convex problems

The fast gradient method belongs to a family of *optimal* methods that make use of an *estimate* sequence. In contrast, classical gradient methods rely on a relaxation sequence $f(y^{k+1}) \leq f(y^k)$. The advantage of using an estimate sequence is that it takes into account the global topological properties of convex functions. In comparison, a relaxation sequence works on a narrower scope of the function. Enforcing a relaxation sequence might be inefficient for some problems.

In the following, we assume that the gradient of f_0 is Lipschitz continuous and f_0 is strongly convex, with $\phi > 0$. Furthermore, the constraint set is a *simple* set for which the projected gradient step can be explicitly computed. In Subsection 3.3.3 we show that these assumptions are satisfied by the considered MPC setup.

Let us first look at why we require these properties on problem (3.7). The Lipschitz continuity of the gradient of f_0 is required to compute the projection onto the set $\mathcal{Y}$. To build an estimate sequence the strong convexity of f_0 is additionally required. To compute a vector z for the current step k on the estimate sequence, we make use of two auxiliary scalar sequences α^k and β^k. The scalar sequence $\beta^k, k \geq 0$ is computed as

$$\beta^k = (\alpha^k)^2(1 - \alpha^k)/((\alpha^k)^2 + \alpha^{k+1}), \tag{3.12}$$

where

$$(\alpha^{k+1})^2 = (1 - \alpha^{k+1})(\alpha^k)^2 + \frac{\phi}{L}\alpha^{k+1}, \tag{3.13}$$

with $0 < \sqrt{\frac{\phi}{L}} \leq \alpha^0$. In these two relations $(\alpha^k)^2$ denotes α^k to the power of 2.

Our last assumption limit us to use a simple set. Examples of such sets are box, simplex, and Euclidean balls. We are particularly interested in box constraint sets. Specially we consider the set $\mathcal{Y} = \{y \in \mathbb{R}^{n_y} \mid \underline{y}_i \leq y_i \leq \overline{y}_i,\ i = 1, \ldots, n_y\}$, for which the projection (3.10) is simply computed as the entry-wise arithmetic saturation. So the i-th component of $\mathcal{P}_{\mathcal{Y}}(z)$, with $z \in \mathbb{R}^{n_y}$, is computed as

Require: $y^0 \in \mathcal{Y}, 0 < \sqrt{\frac{\phi}{L}} \leq \alpha^0 < 1$
1: set $z = y^0$
2: **for** $k = 0 \to k_{\max} - 1$ **do**
3: compute $y^{k+1} = \mathcal{G}_{\mathcal{Y}}(z)$
4: compute $\alpha^{k+1} \in (0, 1)$ from (3.13)
5: compute β^k from (3.12)
6: compute $z = y^{k+1} + \beta^k(y^{k+1} - y^k)$
7: **end for**
8: **return** y^{k+1}

Algorithm 2: Fast gradient method for convex problems

Require: $y^0 \in \mathcal{Y}$, ν from (3.16)
1: set $z = y^0$
2: **for** $k = 0 \to k_{\max} - 1$ **do**
3: compute $y^{k+1} = \mathcal{G}_{\mathcal{Y}}(z)$
4: compute $z = y^{k+1} + \nu(y^{k+1} - y^k)$
5: **end for**
6: **return** y^{k+1}

Algorithm 3: Fast gradient method for strictly convex problems

$$\mathcal{P}_{\mathcal{Y}}(z)_i = \begin{cases} \overline{y}_i & \text{if } z_i > \overline{y}_i \\ \underline{y}_i & \text{if } z_i < \underline{y}_i \\ z_i & \text{otherwise.} \end{cases} \tag{3.14}$$

More concisely, it can be equivalently expressed as

$$\mathcal{P}_{\mathcal{Y}}(z) = \max(\underline{y}, \min(\overline{y}, z)), \tag{3.15}$$

where max and min are applied entry-wise. It follows that the projected gradient step (3.11) is easily computed. The FGM is summarized in Algorithm 2.

For strictly convex problems we can set $\alpha_0 = \sqrt{\frac{\phi}{L}}$, and from (3.13) we get a constant $\alpha_k = \alpha_0, k = 1, \ldots$. Moreover, from (3.12) we also get a constant $\beta_k = \nu, k = 0, \ldots$, with

$$\nu = \frac{\sqrt{L} - \sqrt{\phi}}{\sqrt{L} + \sqrt{\phi}}. \tag{3.16}$$

In this case, the FGM turns out to be even simpler, as can be seen in Algorithm 3.

The fast gradient method shows a convergence of $O(1)\min(\sqrt{\kappa}\ln(1/\epsilon), \sqrt{L/\epsilon})$, with ϵ the suboptimality level, and $\kappa = L/\phi$ the condition number of the cost function's Hessian [97, Theorem 2.2.3]. In comparison, a gradient method shows a convergence of $O(1)\min(\kappa\ln(1/\epsilon), L/\epsilon)$. This implies that the fast gradient method is much faster when the Hessian has a very low condition number.

We have shown how the FGM can deal with problems like (3.7). This type of problem have constraints that only restrict the domain of the problem directly, i.e. $y \in \mathcal{Y}$. We consider next convex problems with a more general type of constraints.

3.3.2. Augmented Lagrangian method

In this subsection, we deal with the more general optimization problem (3.4). The basic idea behind the augmented Lagrangian method (ALM) is to solve a series of unconstrained optimization problems instead of solving the original constrained problem at once. The following presentation is based on [29, Ch. 12, Ch. 17]

ALM for problems with equality constraints

Consider the equality constrained problem (i.e. $\mathcal{I} = \emptyset$)

$$\begin{aligned} &\underset{y\in\mathbb{R}^{n_y}}{\text{minimize}} \;\; f_0(y) \\ &\text{subject to} \;\; f_i(y) = 0,\; i \in \mathcal{E}. \end{aligned} \tag{3.17}$$

The Lagrangian function (cf. (2.5)) for this problem is given by

$$\overline{\mathcal{L}}(y, \lambda) = f_0(y) + \sum_{i\in\mathcal{E}} \lambda_i f_i(y), \tag{3.18}$$

and the augmented Lagrangian function for this problem is defined as

$$\mathcal{L}_A(y, \lambda, \mu) = f_0(y) + \sum_{i\in\mathcal{E}} \lambda_i f_i(y) + \frac{\mu}{2}\sum_{i\in\mathcal{E}} f_i^2(y). \tag{3.19}$$

Here μ is the penalty parameter. Compared to the Lagrangian function $\overline{\mathcal{L}}(y, \lambda)$, the augmented Lagrangian $\mathcal{L}_A(y, \lambda, \mu)$ additionally penalizes infeasibilities (i.e. $f_i(y) \neq 0$) by squaring them and scaling them by $\frac{\mu}{2}$.

Let us first take an intuitive approach to justify the use of the ALM method. We formalize these ideas later. We want to solve the unconstrained problem

$$\underset{y\in\mathbb{R}^{n_y}}{\text{minimize}} \;\; \mathcal{L}_A(y,\lambda,\mu). \tag{3.20}$$

Observe from the KKT conditions (3.5) that if we have knowledge of the optimal Lagrange multiplier λ^*, then the minimizer y^* of $\mathcal{L}_A(y,\lambda^*,\mu)$ is also the solution to (3.17). To see this, note that $\nabla_y\mathcal{L}_A(y^*,\lambda^*,\mu) = \nabla_y\mathcal{L}(y^*,\lambda^*) = 0$. Recall that for convex unconstrained problems, a sufficient and necessary conditions for a point y to be optimal is $\nabla_y f_0(y) = 0$, which can also be derived from (3.5) with $\mathcal{I} = \mathcal{E} = \emptyset$. In this particular case, we can find the solution y^* simply by solving an unconstrained problem. Now consider the more realistic case in which λ^* is unknown. First recall that the term $\frac{\mu}{2}\sum_{i\in\mathcal{E}} f_i^2(y)$ penalizes infeasibilities. Intuition then tells us that the solution $y^\sim$ to $\mathcal{L}_A(y,\lambda,\mu)$ should be a better approximation to y^* for higher values of μ.

Let us now combine the influence of these two factors, namely the knowledge of λ^* and the value of the penalty parameter μ. We additionally assume that a numerical optimization algorithm solves the problem only approximately. Let then y^k denote the *approximate* minimizer of the function $\mathcal{L}_A(y,\lambda^k,\mu^k)$. For a μ^k large enough we have

$$0 \approx \nabla_y\mathcal{L}_A(y^k,\lambda^k,\mu^k) = \nabla f_0(y^k) + \sum_{i\in\mathcal{E}}(\lambda_i^k + \mu^k f_i(y^k))\nabla f_i(y^k).$$

From this and the KKT conditions we can deduce that

$$\lambda_i^* \approx \lambda_i^k + \mu^k f_i(y^k), \;\; \forall\, i\in\mathcal{E}, \tag{3.21}$$

and from it we have

$$f_i(y^k) \approx \frac{1}{\mu^k}(\lambda_i^* - \lambda_i^k), \;\; \forall\, i\in\mathcal{E}.$$

From this last relation, we can observe that the infeasibilities become smaller either by increasing μ^k or by using a λ^k that is close to the optimal multiplier vector λ^*. Furthermore, from (3.21) we can derive a relation for an improved estimate of λ^*:

$$\lambda_i^{k+1} = \lambda_i^k + \mu^k f_i(y^k), \;\; \forall\, i\in\mathcal{E}. \tag{3.22}$$

We refer to this relation as the *multiplier update*.

Require: λ^0, μ
1: **for** $k = 0, \dots, k_{\max} - 1$ **do**
2: Find $y^k = \arg\min \mathcal{L}_A(y, \lambda^k, \mu)$
3: Compute $\lambda_i^{k+1} = \lambda_i^k - \mu f_i(y^k), \quad \forall\, i \in \mathcal{E}$
4: **end for**
5: **return** y^k, λ^{k+1}

Algorithm 4: Augmented Lagrangian method (ALM)

Having now a better estimate of λ^*, it makes sense to find an improved estimate of y^* by iteratively minimizing $\mathcal{L}_A$ using better estimates of λ^* at each iteration. This is the basic idea of the augmented Lagrangian method, which is presented in Algorithm 4 (for constant μ).

ALM for quadratic programs

Let us now apply the augmented Lagrangian method to the quadratic program (2.4). In particular, we consider QPs with constraints having the structure (3.3) that arise from MPC problems. That is, QP (2.4) can be expressed in the form

$$\begin{aligned} \underset{y \in \mathbb{R}^{n_y}}{\text{minimize}} \quad & f_0(y) \\ \text{subject to} \quad & \underline{y} \leq y \leq \overline{y} \\ & \underline{c} \leq \hat{C} y \leq \overline{c}, \end{aligned} \tag{3.23}$$

which is the type of QP typically arising from the MPC setup considered in this work. Although this QP contains inequality constraints only, the equality constrained ALM can still be applied by introducing slack variables.

The following is based on [98, Ch.5]. Problem (3.23) is equivalent to

$$\begin{aligned} \underset{\underline{y} \leq y \leq \overline{y},\, s}{\text{minimize}} \quad & f_0(y) \\ \text{subject to} \quad & \underline{c}_i \leq \hat{C}_i y + s_i \leq \overline{c}_i, \\ & s_i = 0, \quad \forall\, i \in \mathcal{C}, \end{aligned} \tag{3.24}$$

where $\mathcal{C}$ denotes the set of indices for mixed constraints. Note that we are now minimizing with respect to the vector $(y^\mathsf{T}, s^\mathsf{T})^\mathsf{T}$. If we apply the ALM to the equality $s = 0$

we obtain

$$\begin{aligned} \underset{\underline{y}\le y\le \overline{y},\ s}{\text{minimize}} \ \ & f_0(y) + \sum_{i\in\mathcal{C}} (\lambda_i^k s_i + \frac{\mu}{2} s_i^2) \\ \text{subject to} \ \ & \underline{c}_i \le \hat{C}_i y + s_i \le \overline{c}_i, \ \ \forall\, i \in \mathcal{C}, \end{aligned} \tag{3.25}$$

and the multiplier update is given by

$$\lambda_i^{k+1} = \lambda_i^k + \mu s_i^k, \ \ \forall\, i \in \mathcal{C}. \tag{3.26}$$

We can minimize (3.25) first with respect to s for each fixed y, such that

$$\underset{\underline{c}\le \hat{C} y + s \le \overline{c}}{\text{minimize}} \ \ f_0(y) + \sum_{i\in\mathcal{C}} (\lambda_i^k s_i + \frac{\mu}{2} s_i^2), \tag{3.27}$$

is equivalent to

$$\sum_{i\in\mathcal{C}} \underset{\underline{c}_i\le \hat{C}_i y + s_i \le \overline{c}_i}{\text{minimize}} \ (\lambda_i^k s_i + \frac{\mu}{2} s_i^2). \tag{3.28}$$

This problem is a sum of scalar optimization problem which can be solved analytically. Consider the unconstrained case of each subproblem $i \in \mathcal{C}$, whose solution is given by $\hat{s}_i$, the scalar for which the derivative of $\tilde{f}_i(s_i) = \lambda_i^k s_i + \frac{\mu}{2} s_i^2$ with respect to s_i is zero. We then have

$$\frac{\mathrm{d}\tilde{f}_i(s_i)}{\mathrm{d}s_i} = \lambda_i^k + \mu s_i,$$

from which it follows that

$$\hat{s}_i = -\frac{\lambda_i^k}{\mu}.$$

With this in mind, the constrained problem (3.28) has the solution $\hat{s}_i$ if $\underline{c}_i \le \hat{C}_i y + \hat{s}_i \le \overline{c}_i$. Otherwise, i belongs to the active set, i.e. the solution is given by:

$$s_i^*(\lambda_i^k, y) = \begin{cases} \overline{c}_i - \hat{C}_i y & \text{if } \lambda_i^k + \mu(\overline{c} - \hat{C}_i y) < 0 \\ \underline{c}_i - \hat{C}_i y & \text{if } \lambda_i^k + \mu(\underline{c} - \hat{C}_i y) > 0 \\ -\lambda_i^k/\mu & \text{otherwise.} \end{cases} \tag{3.29}$$

Then, the minimum of (3.28) is given by

$$\check{p}_i^*(\lambda_i^k, y) = \lambda_i^k s_i^*(\lambda_i^k, y) + \frac{\mu}{2} s_i^*(\lambda_i^k, y)^2, \ \ \forall\, i \in \mathcal{C}. \tag{3.30}$$

Using these results, we can eliminate the explicit dependence on s from problem (3.25),

which is then equivalent to

$$\underset{\underline{y}\leq y\leq \overline{y}}{\text{minimize}} \;\; f_0(y) + \sum_{i\in\mathcal{C}} \check{p}_i^*(\lambda_i^k, y). \tag{3.31}$$

This problem in turn can be solved using the fast gradient method if the condition described in Subsection 3.3.1 are met.

So far, we have outlined the FGM and the ALM for a convex QP derived from an MPC setup. However, we haven't yet explicitly shown that the MPC problem (2.15) in fact meets the necessary conditions for these methods to apply. The next subsection deals with this challenge in detail.

3.3.3. The ALM+FGM algorithm for MPC

In the following we tailor the ALM combined with the FGM to the MPC problem. We discuss how the particular characteristics of MPC are used to efficiently apply the ALM+FGM algorithm.

We first take into account that the MPC problem is equivalent to the parametric QP (2.17). We assume, that for any parameter $\mathbf{p}$ we get a problem in one of the following two forms: input constrained problem (3.7), or mixed constraint problem (3.23). Next, we discuss these two cases in detail.

Input constrained case

We start by expressing (3.7) using the MPC input sequence $\mathbf{u}$ as optimization variable:

$$\underset{\mathbf{u}\in\mathcal{U}}{\text{minimize}} \;\; \frac{1}{2}\mathbf{u}^\mathsf{T} H \mathbf{u} + g^\mathsf{T}\mathbf{u}. \tag{3.32}$$

Here we have $\mathcal{U} = \{\mathbf{u} \mid \underline{\mathbf{u}} \leq \mathbf{u} \leq \overline{\mathbf{u}}\}$, and we denote the cost function as $f_0^{\text{IC}} = \frac{1}{2}\mathbf{u}^\mathsf{T} H \mathbf{u} + g^\mathsf{T}\mathbf{u}$. As discussed in Subsection 3.3.1, the FGM can efficiently solve this type of problems under certain conditions. The fast gradient method was first proposed in the MPC context in [99]. Recall that in Subsection 2.3.2 we discussed strict convexity of an MPC problem, which required having a positive definite Hessian of the QP's cost function. As mentioned in Subsection 3.3.1, the basic requirements for the applicability of the FGM are the Lipschitz continuity of the gradient of f_0^{IC} and the strong convexity of f_0^{IC}. Furthermore, the Lipschitz constant L and the strong convexity parameter ϕ

Require: initial guess $\mathbf{u}^0$, current $\mathbf{p}$
1: form (3.32) for the given $\mathbf{p}$
2: set $\mathbf{w} = \mathbf{u}^0$
3: **for** $j = 0 \to j_{in} - 1$ **do**
4: compute $\mathbf{u}^{j+1} = \mathcal{G}_{\mathcal{U}}^{\mathrm{IC}}(\mathbf{w})$
5: compute $\mathbf{w} = \mathbf{u}^{j+1} + \nu(\mathbf{u}^{j+1} - \mathbf{u}^j)$
6: **end for**
7: **return** $\mathbf{u}^{j+1}$

Algorithm 5: Fast gradient method (FGM) for MPC with input constraints

are given by the maximum and minimum eigenvalues of H, respectively. Consequently, strong convexity of f_0^{IC} is guaranteed by $H > 0$. This in turn allows us to select a constant scalar parameter ν for the FGM, given by (3.16), which greatly simplifies the FGM implementation (see Algorithm 3).

The gradient of the cost function is given by

$$\nabla f_0^{\mathrm{IC}}(\mathbf{u}) = H\mathbf{u} + g. \tag{3.33}$$

The projected gradient step of a point $\mathbf{w} \in \mathbb{R}^{Nm}$ onto the set $\mathcal{U}$ is computed by

$$\mathcal{G}_{\mathcal{U}}^{\mathrm{IC}}(\mathbf{w}) = \mathcal{P}_{\mathcal{U}}(\mathbf{w} - \frac{1}{L}\nabla f_0^{\mathrm{IC}}(\mathbf{w})), \tag{3.34}$$

where

$$\mathcal{P}_{\mathcal{U}}(\mathbf{w}) = \max(\underline{\mathbf{u}}, \min(\overline{\mathbf{u}}, \mathbf{w})). \tag{3.35}$$

The final step is to compute an extra step

$$\mathbf{w} = \mathbf{u}^{j+1} + \nu(\mathbf{u}^{j+1} - \mathbf{u}^j), \tag{3.36}$$

and then repeat the procedure as shown in Algorithm 5.

Mixed constraints case

As in the input constrained case, we start this section with problem (3.23) using the MPC input sequence as optimization variable:

$$\begin{aligned} \underset{\mathbf{u}\in\mathcal{U}}{\text{minimize}} \;\; & \frac{1}{2}\mathbf{u}^\intercal H\mathbf{u} + g^\intercal\mathbf{u} \\ \text{subject to} \;\; & \underline{\mathbf{c}} \leq \hat{C}\mathbf{u} \leq \overline{\mathbf{c}}, \end{aligned} \tag{3.37}$$

Applying the augmented Lagrangian method for double sided constraints (3.31), to problem (3.37) we have

$$\underset{\mathbf{u}\in\mathcal{U}}{\text{minimize}} \;\; f_0^{\text{MC}}(\mathbf{u};\lambda^i), \tag{3.38}$$

where

$$f_0^{\text{MC}}(\mathbf{u};\lambda^i) = \frac{1}{2}\mathbf{u}^\intercal H\mathbf{u} + g^\intercal\mathbf{u} + \check{p}^*(\mathbf{u};\lambda^i). \tag{3.39}$$

The term $\check{p}^*(\mathbf{u};\lambda^i)$ is given by (3.30) so that for any λ

$$\begin{aligned} \check{p}^*(\mathbf{u};\lambda) = \min_{\mathbf{s}\in\mathbb{R}^{Nq+r}} \;\; & \lambda^\intercal\mathbf{s} + \frac{\mu}{2}\|\mathbf{s}\|_2^2 \\ \text{s. t.} \;\; & \underline{\mathbf{c}} \leq \hat{C}\mathbf{u} + \mathbf{s} \leq \overline{\mathbf{c}}, \end{aligned} \tag{3.40}$$

where $\mathbf{s}$ is a slack variable. We are looking for the pair of vectors $\mathbf{u}^*, \lambda^*$. The former is the minimizer of (3.37), whereas the latter is the Lagrange multiplier that together with $\mathbf{u}^*$ minimizes (3.38). The finite sequence $\{\lambda^0, \ldots, \lambda^{i_{ex}}\}$ that approximates λ^* is computed using $\check{p}^*(\mathbf{u};\lambda^i)$, given that (3.40) is solved analytically. The next element in the sequence is given by the multiplier update, which is computed as

$$\lambda^{i+1} = \mathcal{S}(\mathbf{u}^{*i}, \lambda^i), \tag{3.41}$$

where $\mathbf{u}^{*i}$ denotes the minimizer of (3.38) using λ^i instead of λ^*, i.e.:

$$\mathbf{u}^{*i} = \arg\min_{\mathbf{u}\in\mathcal{U}} f_0^{\text{MC}}(\mathbf{u};\lambda^i), \tag{3.42}$$

and

$$\mathcal{S}(\mathbf{u},\lambda) = \min(\underline{\lambda}(\mathbf{u},\lambda), 0) + \max(\bar{\lambda}(\mathbf{u},\lambda), 0). \tag{3.43}$$

The jth element of $\underline{\lambda}(\mathbf{u}, \lambda)$, and $\bar{\lambda}(\mathbf{u}, \lambda)$ are given by

$$\begin{aligned} \underline{\lambda}_j(\mathbf{u}, \lambda) &= \lambda_j + \mu(\hat{C}_j \mathbf{u} - \underline{\mathbf{c}}_j), \\ \overline{\lambda}_j(\mathbf{u}, \lambda) &= \lambda_j + \mu(\hat{C}_j \mathbf{u} - \overline{\mathbf{c}}_j), \end{aligned} \tag{3.44}$$

respectively. To find an approximation $\mathbf{u}^i$ of the minimizer $\mathbf{u}^{*i}$, we use the FGM. To apply the FGM, we need to show that there exist a Lipschitz constant for the gradient of $f_0^{\mathrm{MC}}(\mathbf{u}; \lambda)$ and a convexity parameter for $f_0^{\mathrm{MC}}(\mathbf{u}; \lambda)$.

In [83, Proposition 1] it is shown that a Lipschitz constant L for ∇f_0^{MC} is given by $L = \|H + \mu \hat{C}^\intercal \hat{C}\|$ and a strong convexity parameter ϕ for f_0^{MC} is given by the smallest eigenvalue of H. For every λ^i of the sequence, we need to find an approximate minimizer $\mathbf{u}^i$. We proceed iteratively similarly to the input constrained case. The main difference is the cost function $f_0^{\mathrm{MC}}(\mathbf{u}; \lambda^i)$ and its gradient, which is now given by

$$\nabla f_0^{\mathrm{MC}}(\mathbf{u}; \lambda^i) = H\mathbf{u} + g + \nabla \check{p}^*(\mathbf{u}; \lambda^i), \tag{3.45}$$

and the gradient of (3.40) is computed as

$$\nabla \check{p}^*(\mathbf{u}; \lambda^i) = \hat{C}^\intercal \mathcal{S}(\mathbf{u}, \lambda^i). \tag{3.46}$$

The projected gradient step is in this case given by

$$\mathcal{G}_{\mathcal{U}}^{\mathrm{MC}}(\mathbf{w}; \lambda^i) = \mathcal{P}_{\mathcal{U}}(\mathbf{w} - \frac{1}{L} \nabla f_0^{\mathrm{MC}}(\mathbf{w}; \lambda^i)), \tag{3.47}$$

where $\mathcal{P}_{\mathcal{U}}$ is computed as in (3.35). The final extra step of the FGM is computed as in (3.36). The FGM is given in this scenario by Algorithm 6.

The augmented Lagrangian method (ALM) is outlined in Algorithm 7. Note that the complete algorithm ALM+FGM requires two iteration loops, the internal loop corresponds to the FGM, and the external loop to the multiplier update of the ALM. The algorithm has therefore only three tuning parameters: the first two are the maximum number of iterations of each loop, and are denoted as i_{ex} (external loop) and j_{in} (internal loop), and the last is the penalty parameter μ of the ALM. We discuss next why the ALM+FGM algorithm is well suited for embedded MPC applications.

Require: initial guess $\mathbf{u}^0$, multiplier λ^i
1: set $\mathbf{w} = \mathbf{u}^0$
2: **for** $j = 0 \to j_{in} - 1$ **do**
3: compute $\mathbf{u}^{j+1} = \mathcal{G}_{\mathcal{U}}^{\mathrm{MC}}(\mathbf{w}; \lambda^i)$
4: compute $\mathbf{w} = \mathbf{u}^{j+1} + \nu(\mathbf{u}^{j+1} - \mathbf{u}^j)$
5: **end for**
6: **return** $\mathbf{u}^{j+1}$

Algorithm 6: Fast gradient method for MPC with mixed constraints.

Require: Initial guesses λ^0, $\mathbf{u}^0$, current $\mathbf{p}$.
1: form (3.37) for the given $\mathbf{p}$
2: **for** $i = 0, \ldots, i_{ex} - 1$ **do**
3: Find $\mathbf{u}^{i+1}$ using Algorithm 6 with initial guess $\mathbf{u}^i$, and parameters λ^i
4: Compute $\lambda^{i+1} = \mathcal{S}(\mathbf{u}^{i+1}, \lambda^i)$
5: **end for**
6: **return** λ^{i+1}, $\mathbf{u}^{i+1}$

Algorithm 7: Augmented Lagrangian method for MPC with mixed constraints.

3.3.4. ALM+FGM for embedded MPC

Perhaps one of the main characteristics that differentiate the ALM+FGM algorithm from other approaches is that it not only exploits the MPC properties discussed in Section 3.1, but at the same time, it also takes into account the limitations of embedded systems discussed in Section 2.1.

Exploiting the properties of MPC

In Section 3.1, we showed that the parametric QP arising from MPC has several properties that can be exploited to get efficient MPC implementations. For a given number of external and internal iterations (i_{ex}, j_{in}) of Algorithm 7, the number of arithmetic operations of the complete algorithm is fixed. Furthermore, the involved operations are only additions and multiplications. On a modern embedded computational platform, these operations are temporally deterministic. This in turn implies that the algorithm itself will show temporal determinism for a fixed number of iterations. In Chapter 5, we discuss the implementation of the algorithm, and we show with examples that it is indeed deterministic.

Cold start or warm start strategies can easily be implemented. Here, not only an initial guess for the input vectors can be supplied, but also for the Lagrange multipliers. In the case of warm start using the shift method, a feasible initial guess for the inputs is straightforward to find for box constraints. Furthermore, the last element of (3.2) can be determined off-line. In the case of the Lagrange multipliers, the only restriction on a multiplier for a problem like (2.8) is that $\lambda_i \geq 0$. However, if the QP is formulated with explicit use of double sided constraints as in (3.37), the Lagrange multipliers can also take negative values. Therefore, setting an unknown multiplier to zero is always a safe strategy when using the shift method for warm starting.

The special structure of the constraints can also be exploited. The FGM takes care of the input constraints. These are treated as hard constraints, as is usual in MPC. The state constraints are dealt with by the ALM, and are implicitly treated as soft constrained. In Chapter 5 we show that for certain values of the penalty parameter $\mu \geq \overline{\mu}$, the ALM+FGM behaves as an exact penalty method.

Numerical conditioning

The condition number of the internal problem is given by

$$\kappa = L/\phi = \frac{\|H + \mu \hat{C}^{\mathsf{T}} \hat{C}\|}{\lambda_{min}(H)}. \tag{3.48}$$

Recall that the convergence rate of the fast gradient method depends on the value of κ, as in (3.48). This in turn is bounded below by the condition number of the Hessian κ_H. Therefore, a reduction of κ_H implies a reduction of κ itself, and thus faster convergence of the FGM. One way to do reduce the condition number of κ_H is to compute a pre-conditioner for H, [36]. Another way is to compute the weighting matrices of the MPC problem such that κ_H is reduced [100]. In Chapter 5 we discuss the latter strategy.

A further benefit of having a low condition number is more reliable numerical results on embedded platforms. A rule of thumb states that the conditioning of the Hessian is proportional to the accumulation of round-off errors in the solution of a QP [101]. This is of particular importance in embedded systems using single precision (32-bit) floating-point computations, and even more so when using fixed-point computations. It follows that a low value of κ_H, and thus of κ, has two benefits: a faster convergence rate of the ALM+FGM algorithm and reduced numerical errors on embedded platforms.

3.4. Summary

In this chapter, we discussed the main properties of embedded MPC that are exploited by tailored optimization algorithms. We briefly reviewed some existing approaches, highlighting their strengths and weaknesses. Afterward, we presented a novel optimization algorithm based on an augmented Lagrangian method combined with Nesterov's fast gradient method (the ALM+FGM algorithm). This algorithm can be easily warm started, which, for certain MPC problems, offers an speed advantage in finding a solution. Furthermore, the ALM+FGM inherently deals with soft constraints. This avoids running into infeasible problems on-line. Due to the convergence rate of the FGM, this algorithm works best for MPC problems that are well conditioned, i.e. where κ_H, the condition number of the Hessian of the QP, is low. The ALM+FGM relies only on multiplications and additions. This implies that, for a fixed number of iterations of the ALM+FGM, the algorithm shows temporal determinism. Furthermore, this makes the ALM+FGM simple to implement using fixed-point arithmetic. The use of fixed-point operations is limited to cases where the round-off errors remain small. The magnitude of this errors is related to the condition number of the Hessian of the QP. Furthermore, we saw that the horizon length is directly proportional to κ_H. In addition, due to limitations of embedded applications, it is common to select horizon lengths that are just long enough for good closed-loop performance. Therefore, we focused our discussion to problems where the horizon length is not too long. In such cases, a condensed QP has in general lower memory and CPU requirements and is thus preferred to a sparse QP. The next chapter presents a systematic approach to turn a rather general MPC formulation into a condensed QP, which can then be solve efficiently using the presented approach.

4. General description of multistage problems

This chapters describes in detail the family of *multistage* problems we consider in this thesis. We use the term multistage problem to describe a broad class of problems that include model predictive control and moving horizon estimation.

We start this chapter giving a formal mathematical description of the general type of multistage problems we consider. In previous chapters we showed that condensed formulations are preferred to sparse formulations in many embedded applications. Therefore, we show later in the chapter how the presented multistage setup can be transformed into a condensed parametric optimization problem. Afterwards we introduce a high-level language that greatly simplifies formulating the considered multistage problems.

4.1. General formulation of multistage problems

In this section we use the following notation: we denote matrices with capital letters, whereas vectors and scalars are denoted with lowercase letters. Bold face letters denote arrays (or sequences) of vectors or matrices. The dimension of the array will be stated explicitly. We will write for example that $\mathbf{P}$ is a 3-dimensional array. This implies that $\mathbf{P}_{kji}$ is a matrix. Similarly, the term $\mathbf{u}_i$ is a vector from the 1-dimensional array (i.e. the sequence) $\mathbf{u}$.

4.1.1. Motivational example

Consider the following optimization problem:

$$\begin{aligned} \underset{\mathbf{u}}{\text{minimize}} \quad & \sum_{i=0}^{N-1} \|\mathbf{u}_i\|_R^2 + \sum_{i=1}^{N} \|\mathbf{x}_i\|_Q^2 \\ \text{subject to} \quad & \mathbf{x}_{i+1} = A\mathbf{x}_i + B\mathbf{u}_i, \qquad i = 0, ..., N-1, \\ & \mathbf{x}_0 = x^k. \end{aligned}$$

We aim at expressing this type of problems using a single general formulation. To that end, we introduce the following relation:

$$\vartheta_{ji} = \mathbf{D}_{ji}\mathbf{u} + \mathbf{P}_{ji}\mathbf{p} + \mathbf{E}_{ji}\mathbf{e}. \tag{4.1}$$

Let us express the optimization problem above in terms of the vector sequences ϑ_j, with $j = \{0, 1, 2\}$. We start by defining parameter x^k as the vector $\mathbf{p}$ and $\mathbf{x}$ as the vector sequence $\mathbf{e}$. We define $\mathbf{e}$ to be equal to ϑ_2, and each of the vectors ϑ_{2i} is to be expressed in the form (4.1). Thus, we can express $\mathbf{x}_0 = x^k$ as $\mathbf{e}_0 = \mathbf{p}$, which in turn is equivalent to:

$$\vartheta_{20} = \mathbf{P}_{20}\mathbf{p}$$

with $\mathbf{P}_{20} = I$ and $\mathbf{D}_{20}$, $\mathbf{E}_{20}$ equal to zero matrices of appropriate size. The next vector of the sequence $\mathbf{x}$ is given by $\mathbf{x}_1 = A\mathbf{x}_0 + B\mathbf{u}_0$, which is equivalent to $\mathbf{e}_1 = A\mathbf{e}_0 + B\mathbf{u}_0$, and can be expressed as:

$$\vartheta_{21} = \mathbf{E}_{21}\mathbf{e} + \mathbf{D}_{21}\mathbf{u}$$

with $\mathbf{E}_{21} = \begin{bmatrix} A & 0 & \cdots & 0 \end{bmatrix}$, and $\mathbf{D}_{21} = \begin{bmatrix} B & 0 & \cdots & 0 \end{bmatrix}$.

More generally the expression $\mathbf{x}_{i+1} = A\mathbf{x}_i + B\mathbf{u}_i$, $i = 0, \ldots, N-1$, can be brought into form (4.1) in the following way:

$$\begin{aligned} \mathbf{x}_{i+1} &= A\mathbf{x}_i + B\mathbf{u}_i, & i &= 0, \ldots, N-1 \\ \mathbf{x}_i &= A\mathbf{x}_{i-1} + B\mathbf{u}_{i-1}, & i &= 1, \ldots, N \\ \mathbf{e}_i &= A\mathbf{e}_{i-1} + B\mathbf{u}_{i-1}, & i &= 1, \ldots, N \\ \vartheta_{2i} &= \mathbf{E}_{2i}\mathbf{e} + \mathbf{D}_{2i}\mathbf{u}, & i &= 1, \ldots, N. \end{aligned}$$

The matrices $\mathbf{E}_{2i}$, and $\mathbf{D}_{2i}$, for $i = 1, \ldots, N$ are respectively given by

$$\mathbf{E}_{21} = \begin{bmatrix} A & 0 & \cdots & 0 \end{bmatrix}, \mathbf{E}_{22} = \begin{bmatrix} 0 & A & \cdots & 0 \end{bmatrix}, \cdots, \mathbf{E}_{2N} = \begin{bmatrix} 0 & \cdots & 0 & A \end{bmatrix},$$

and

$$\mathbf{D}_{21} = \begin{bmatrix} B & 0 & \cdots & 0 \end{bmatrix}, \mathbf{D}_{22} = \begin{bmatrix} 0 & B & \cdots & 0 \end{bmatrix}, \cdots, \mathbf{D}_{2N} = \begin{bmatrix} 0 & \cdots & 0 & B \end{bmatrix}.$$

The cost function can be represented in terms of (4.1) in a similar fashion. The term $\sum_{i=1}^{N} \|\mathbf{x}_i\|_Q^2$ is equivalent to $\sum_{i=1}^{N} \|\vartheta_{0i}\|_{\mathbf{M}_{0i}}^2$, which can be expressed as:

$$\vartheta_{0i} = \mathbf{E}_{0i}\mathbf{e}, \quad i = 1, \ldots, N$$

with $\mathbf{M}_{0i} = Q$ for $i = 1, \ldots, N$ and

$$\mathbf{E}_{01} = \begin{bmatrix} 0 & I & 0 & \cdots & 0 \end{bmatrix}, \mathbf{E}_{02} = \begin{bmatrix} 0 & 0 & I & \cdots & 0 \end{bmatrix}, \cdots, \mathbf{E}_{0N} = \begin{bmatrix} 0 & 0 & \cdots & 0 & I \end{bmatrix}.$$

Similarly, the term $\sum_{i=0}^{N-1} \|\mathbf{u}_i\|_R^2$ is equal to $\sum_{i=0}^{N-1} \|\vartheta_{1i}\|_{\mathbf{M}_{1i}}^2$, with $\vartheta_{1i} = \mathbf{D}_{1i}\mathbf{u}, \quad i = 0, \ldots, N-1$, and $\mathbf{M}_{1i} = R$ for $i = 0, \ldots, N-1$ and

$$\mathbf{D}_{00} = \begin{bmatrix} I & 0 & \cdots & 0 \end{bmatrix}, \mathbf{D}_{01} = \begin{bmatrix} 0 & I & \cdots & 0 \end{bmatrix}, \cdots, \mathbf{D}_{0(N-1)} = \begin{bmatrix} 0 & \cdots & 0 & I \end{bmatrix}.$$

This simple example introduces the basic idea behind the multistage formulation we present in this chapter. With this in mind, we formally introduce the formulation. Later in this section we present a complete example.

4.1.2. Abstract formulation of multistage problems

We consider multistage problems described by the following optimization problem:

$$\begin{aligned} \underset{\mathbf{u}}{\text{minimize}} \quad & \sum_{j \in \mathcal{Q}} \sum_{i \in \mathcal{V}_j} (\|\vartheta_{ji}\|_{\mathbf{M}_{ji}}^2 + \mathbf{n}_{ji}^{\mathsf{T}} \vartheta_{ji}^{-}) \\ \text{subject to} \quad & \vartheta_{ji} \leq 0, && \text{for all } i \in \mathcal{V}_j,\ j \in \mathcal{I}_a, \\ & \|\vartheta_{ji}\|_{\mathbf{M}_{ji}}^2 + \mathbf{n}_{ji}^{\mathsf{T}} \vartheta_{ji}^{-} \leq 0, && \text{for all } i \in \mathcal{V}_j,\ j \in \mathcal{I}_q, \\ & \mathbf{e}_{ji} = \vartheta_{ji}, && \text{for all } i \in \mathcal{V}_j,\ j \in \mathcal{E}, \end{aligned} \tag{4.2}$$

where $\vartheta_{ji}^- \in \mathbb{R}^{r_{n_{ji}}}$, and $\vartheta_{ji} \in \mathbb{R}^{r_{\vartheta_{ji}}}$ need to satisfy

$$\vartheta_{ji} = \mathbf{c}_{ji} + \mathbf{D}_{ji}\mathbf{u} + \sum_{k\in\mathcal{P}} \mathbf{P}_{kji}\mathbf{p}_k + \sum_{k\in\mathcal{E}} \mathbf{E}_{kji}\mathbf{e}_k. \tag{4.3}$$

Here $\mathbf{u} \in \mathbb{R}^{r_u}$ denotes the optimization variable, and $\mathbf{p}$ is a 2-dimensional array of $n_\mathcal{P}$ parameters. In this context, $\mathbf{p}_k \in \mathbb{R}^{r_{p_k}}$ represents the parameter $k \in \mathcal{P} = \{0, \ldots, n_\mathcal{P}-1\}$. The parameter $\mathbf{p}_k$ can be a single vector or a sequence of vectors. Similarly, $\mathbf{e}$ is a 2-dimensional array of $n_\mathcal{E}$ vectors that denotes auxiliary vector terms. Thus, for all $j \in \mathcal{E}$, $\mathbf{e}_j$ is defined through an equality constraint. The term $\mathbf{c}$ is a 2-dimensional array of vectors that are constant. The array $\mathbf{D}$ is a 2-dimensional array of matrices, $\mathbf{M}$ is a 2-dimensional array of symmetric matrices, and $\mathbf{n}$ is a 2-dimensional array of vectors. The arrays $\mathbf{P}$ and $\mathbf{E}$ are 3-dimensional arrays of matrices.

The indexes for the cost function are given by the set $\mathcal{Q} = \{0, \ldots, n_\mathcal{Q} - 1\}$, with $n_\mathcal{Q}$ the number of quadratic forms being added, and the set $\mathcal{V}_j = \{0, \ldots, n_j - 1\}$, with n_j the number of vectors in the sequence ϑ_j. The index for the inequality constraints is given by $\mathcal{I}_a = \{n_\mathcal{Q}, \ldots, n_\mathcal{Q} + n_{\mathcal{I}_a} - 1\}$, where $n_{\mathcal{I}_a}$ denotes the number of affine inequality constraints, and by $\mathcal{I}_q = \{n_\mathcal{Q} + n_{\mathcal{I}_a}, \ldots, n_\mathcal{Q} + n_{\mathcal{I}_a} + n_{\mathcal{I}_q} - 1\}$, where $n_{\mathcal{I}_q}$ denotes the number of quadratic inequality constraints. We define the index set $\mathcal{I} = \mathcal{I}_a \cup \mathcal{I}_q = \{n_\mathcal{Q}, \ldots, n_\mathcal{Q} + n_\mathcal{I} - 1\}$, where $n_\mathcal{I} = n_{\mathcal{I}_a} + n_{\mathcal{I}_q}$ denotes the total number of inequality constraints. We use the symbol $\leq$ to denote vector inequalities ($j \in \mathcal{I}_a$) as well as scalar inequalities ($j \in \mathcal{I}_q$) with a slight abuse of notation. The index for the equality constraints is given by the set $\mathcal{E} = \{n_\mathcal{Q} + n_\mathcal{I}, \ldots, n_\mathcal{Q} + n_\mathcal{I} + n_\mathcal{E} - 1\}$, where $n_\mathcal{E}$ denotes the number of equality constraints. Note that ϑ_{ji} is affine with respect to the optimization variable $\mathbf{u}$ and to each parameter $\mathbf{p}_k, k \in \mathcal{P}$.

To avoid recursive definitions of $\mathbf{e}_{ji}$, we impose the following restriction on the equality constraints (4.2):

$$\mathbf{e}_{j0} = \vartheta_{j0}^0, \qquad \text{for all } j \in \mathcal{E}, \tag{4.4a}$$

$$\mathbf{e}_{ji} = \vartheta_{ji}^+, \qquad \text{for all } i \in \mathcal{V}_j - \{0\},\ j \in \mathcal{E}, \tag{4.4b}$$

$$\tag{4.4c}$$

with

$$\vartheta_{j0}^{0} = \mathbf{c}_{j0} + \mathbf{D}_{j0}\mathbf{u} + \sum_{k\in\mathcal{P}} \mathbf{P}_{kj0}\mathbf{p}_k + \sum_{k\in\mathcal{E}_j} \mathbf{E}_{kj0}\mathbf{e}_k, \tag{4.4d}$$

$$\vartheta_{ji}^{+} = \mathbf{c}_{ji} + \mathbf{D}_{ji}\mathbf{u} + \sum_{k\in\mathcal{P}} \mathbf{P}_{kji}\mathbf{p}_k + \sum_{k\in\mathcal{E}_j} \mathbf{E}_{kji}\mathbf{e}_k + \mathbf{E}_{jji}^{+}\mathbf{e}_j, \tag{4.4e}$$

and $\mathbf{E}_{jji}^{+}$ has the columns that multiply the terms $\mathbf{e}_{j\underline{i}}, \underline{i} \geq i$ equal to zero. Furthermore, the set $\mathcal{E}_j = \{\underline{j} \in \mathcal{E} \mid \underline{j} < j\}$ denotes the indexes of equality constraints that have an index inferior than the current index j. This avoids recursive definitions of equality constraints, i.e. each $\mathbf{e}_{ji}$ only depends on symbols that have been previously defined. In other words, $\mathbf{E}_{jj}^{+}$ is a block lower triangular matrix.

The restrictions (4.4) characterize the predictive nature of the multistage problems we consider in this work. To see this, note first that ϑ_{j0}^{0} simply states that the first element $\mathbf{e}_{j0}$ of any auxiliary vector $\mathbf{e}_j$ does not depend on $\mathbf{e}_j$ itself. Additionally, note that ϑ_{ji}^{+} limits $\mathbf{e}_{ji}$ to only depend on parts of the full vector $\mathbf{e}_j$ that have been previously defined. Because we start with the index 0, $\mathbf{e}_{ji}$ depends only on $\mathbf{e}_{j\underline{i}}$, with $0 \leq \underline{i} < i$, and $i \in \mathcal{V}_j - \{0\}$. In other words, if i denotes the discrete-time index, $\mathbf{e}_{ji}$ does not depend on current or future values, it only depends on past $\mathbf{e}_{j\underline{i}}$, with $0 \leq \underline{i} < i$. Furthermore, if the dynamics are of the form $\mathbf{e}_{j(i+1)} = f(\mathbf{e}_{ji})$, the matrix $\mathbf{E}_{jj}^{+}$ is block diagonal.

Note that both ϑ_{j0}^{0} and ϑ_{ji}^{+} may depend on other auxiliary vectors $\mathbf{e}_k,\ k \in \mathcal{E}_j$, i.e. any other vector that has been already defined. For example, assume $\mathcal{E} = \{0, 1, 2\}$. First we treat the vectors in $\mathbf{e}_0$, which do not depend on other $\mathbf{e}_j$, $j \in \mathcal{E}$, except only partly on itself. Afterwards we treat the vectors in $\mathbf{e}_1$, that may only depend on $\mathbf{e}_0$, and partly on $\mathbf{e}_1$. Finally, we construct $\mathbf{e}_2$, which may depend on $\mathbf{e}_0$, $\mathbf{e}_1$ and only partly on itself.

At first glance, the general description given by (4.2) may not look like the typical MPC problem described in Subsection 2.3.1. However, note that if $\mathbf{n}_j^{\intercal}\vartheta_j^{-} = 0,\ \forall\, j \in \mathcal{Q}$, the cost function is basically a sum of quadratic forms, i.e. $\sum_{j=0}^{n_{\mathcal{Q}}-1} \sum_{i=0}^{n_j-1} \|\vartheta_{ji}\|_{\mathbf{M}_{ji}}^{2}$. The scalar n_j resembles a horizon length and $n_{\mathcal{Q}}$ is simply the number of quadratic forms being added. Note that for each $j = 0, \ldots, n_{\mathcal{Q}} - 1$ we can have a different value for n_j. This allows to specify explicitly a prediction horizon and a control horizon, for instance. Furthermore, note that the equality constraints assign ϑ_{ji} to an auxiliary vector $\mathbf{e}_{ji}$. From this, the full term $\mathbf{e}_j$, as defined in (4.4), can then be replaced in the cost and constraint functions if needed. An example might best illustrate the resemblance.

MPC as a multistage problem

Consider the case where we need to track a trajectory in the states and inputs for a linear system with a known disturbance, i.e. $x_{k+1} = Ax_k + Bu_k + w_k$. Our parameter sequence is $\mathbf{p} = \{x, \mathbf{w}, \check{\mathbf{x}}, \check{\mathbf{u}}\}$, with $n_{\mathcal{P}} = 4$. Here, x denotes the current state of the system, $\mathbf{w}$ the sequence of known disturbances for a given horizon N, and $\check{\mathbf{x}}$ and $\check{\mathbf{u}}$ are the state and input trajectories to be tracked for the same horizon. We assume that The MPC problem has box constraints in the inputs, and is subject to state constraints. We can represent this MPC problem as:

$$
\begin{aligned}
\underset{\mathbf{u}}{\text{minimize}} \quad & \sum_{i=0}^{N-1} (\|\mathbf{u}_i - \check{\mathbf{u}}_i\|_R^2 + \|\mathbf{x}_i - \check{\mathbf{x}}_i\|_Q^2) + \|\mathbf{x}_N - \check{\mathbf{x}}_N\|_P^2 \\
\text{subject to} \quad & \underline{u} \leq \mathbf{u}_i \leq \overline{u}, && i = 0, ..., N-1, \\
& \underline{c} \leq C_x \mathbf{x}_i + C_u \mathbf{u}_i \leq \overline{c}, && i = 0, ..., N-1, \\
& \underline{f} \leq C_f \mathbf{x}_N \leq \overline{f}, \\
& \mathbf{x}_{i+1} = A\mathbf{x}_i + B\mathbf{u}_i + \mathbf{w}_i, && i = 0, ..., N-1, \\
& \mathbf{x}_0 = x.
\end{aligned}
$$

Let us focus first on the cost function, which in the general formulation (4.2) looks like:

$$
\begin{aligned}
\sum_{j \in \mathcal{Q}} \sum_{i \in \mathcal{V}_j} \|\vartheta_{ji}\|_{\mathbf{M}_{ji}}^2 + \mathbf{n}_{ji}^\intercal \vartheta_{ji}^- &= \sum_{j=0}^{1} \sum_{i=0}^{n_j - 1} \|\vartheta_{ji}\|_{\mathbf{M}_{ji}}^2, \\
&= \sum_{i=0}^{N-1} \|\vartheta_{0i}\|_{\mathbf{M}_{0i}}^2 + \sum_{i=0}^{N} \|\vartheta_{1i}\|_{\mathbf{M}_{1i}}^2, \\
&= \sum_{i=0}^{N-1} \|\mathbf{D}_{0i}\mathbf{u} + \mathbf{P}_{31i}\mathbf{p}_3\|_{\mathbf{M}_{0i}}^2 + \sum_{i=0}^{N} \|\mathbf{E}_{61i}\mathbf{e}_6 + \mathbf{P}_{21i}\mathbf{p}_2\|_{\mathbf{M}_{1i}}^2, \\
&= \sum_{i=1}^{N-1} \|\mathbf{u}_i - \check{\mathbf{u}}_i\|_R^2 + \sum_{i=0}^{N-1} \|\mathbf{x}_i - \check{\mathbf{x}}_i\|_Q^2 + \|\mathbf{x}_N - \check{\mathbf{x}}_N\|_P^2.
\end{aligned}
$$

Thus, for this case $n_{\mathcal{Q}} = 2$. Additionally, we have $\mathbf{M}_{0i} = R$ and $\mathbf{M}_{1i} = Q$ for $i = 0, \ldots, N-1$, and $\mathbf{M}_{1N} = P$. The terms $\mathbf{n}_{ji}^\intercal \vartheta_{ji}^-$ are all zero. The integers n_j take the values $n_0 = N$ and $n_1 = N + 1$ for $j = 0, 1$, respectively. Important to note is that the newly introduced sequence $\mathbf{x}$ represents an auxiliary sequence $\mathbf{e}_k$, $k = 6 \in \mathcal{E}$. The equality constraints are discuss below. The matrices $\mathbf{D}_{0i}$, $\mathbf{P}_{30i}$, $\mathbf{E}_{61i}$ and $\mathbf{P}_{21i}$ in this case

basically take out the vector i from the corresponding sequences. For example, in the case of the sequence $\mathbf{u}$ the matrices $\mathbf{D}_{0i}$ are:

$$\mathbf{D}_{00} = \begin{bmatrix} I & 0 & \cdots & 0 \end{bmatrix}, \mathbf{D}_{01} = \begin{bmatrix} 0 & I & \cdots & 0 \end{bmatrix}, \cdots, \mathbf{D}_{0(N-1)} = \begin{bmatrix} 0 & \cdots & 0 & I \end{bmatrix},$$

with I and 0 denoting identity and zero matrices of appropriate size, respectively. The matrices $\mathbf{P}_{30i}$, $\mathbf{E}_{61i}$ and $\mathbf{P}_{21i}$ are defined similarly. In the case of the last two we use $-I$ instead of I.

The constraints on the inputs:

$$\underline{u} \leq \mathbf{u}_i \leq \overline{u}, \quad i = 0, ..., N-1,$$

are equivalent to two one-sided *vector* inequalities, i.e. $\mathbf{u}_i \leq \overline{u}$ and $-\mathbf{u}_i \leq -\underline{u}$, $i = 0, \ldots, N-1$. The inequality indexes are $j = 2, 3$ with $n_2 = n_3 = N$. The first inequality will be translated to $(\mathbf{D}_{2i}\mathbf{u} + \mathbf{c}_{2i}) \leq 0$, with $\mathbf{D}_{2i}$ defined identically as $\mathbf{D}_{0i}$ and $\mathbf{c}_{2i} = -\overline{u}$, for $i = 0, \ldots, N-1$. The other inequality is constructed in a similar fashion.

The more complex mixed inequalities constraints can be expressed by (4.2) in a similar way:

$$\begin{aligned} \underline{c} \leq C_x\mathbf{x}_i + C_u\mathbf{u}_i \leq \overline{c}, \quad i = 0, ..., N-1, \\ \underline{f} \leq C_f\mathbf{x}_N \leq \overline{f}, \end{aligned}$$

which, just as in the input constraint case, can be expressed as two one-sided inequality constraints. We have $j = 4, 5$ and $n_4 = n_5 = N$. The matrices in (4.2) ($\mathbf{D}_4$, etc.) take a similar form as for the input constraint case. The exact values are not discussed in detail here. Note that both, the input and mixed constraint, define our set $\mathcal{I} = \{2, 3, 4, 5\}$.

Let us now define our equality constraints. We only need to define the state vector sequence $\mathbf{x}$ as an auxiliary vector. This relation can be expressed in the following form:

$$\begin{aligned} \mathbf{x}_{i+1} = A\mathbf{x}_i + B\mathbf{u}_i + \mathbf{w}_i, \quad i = 0, \ldots, N-1, \\ \mathbf{x}_0 = x. \end{aligned}$$

Note that the above two expressions define the single vector sequence $\mathbf{x}$. Moreover, it complies with the restrictions given by (4.4). The equality index set is defined as $\mathcal{E} = \{6\}$, and $n_6 = N + 1$. Thus, the sequence $\mathbf{e}_6$ is defined by $\mathbf{e}_{6i} = \mathbf{P}_{06i}\mathbf{p}_0$, for $i = 0$, together with $\mathbf{e}_{6i} = \mathbf{E}_{66i}\mathbf{e}_6 + \mathbf{D}_{6i}\mathbf{u} + \mathbf{P}_{16i}\mathbf{p}_1$, for $i = 1, \ldots, N$. In the former expression we

have $\mathbf{P}_{060} = I$ because $\mathbf{p}_0 = x$ is simply a vector, not a sequence. In the latter expression the matrices $\mathbf{D}_{61}$ to $\mathbf{D}_{6N}$ have the same structure as $\mathbf{D}_{00}$ to $\mathbf{D}_{0(N-1)}$, respectively. The main difference is that we use B instead of I. For example

$$\mathbf{D}_{61} = \begin{bmatrix} B & 0 & \cdots & 0 \end{bmatrix}, \mathbf{D}_{62} = \begin{bmatrix} 0 & B & \cdots & 0 \end{bmatrix}, \cdots, \mathbf{D}_{6N} = \begin{bmatrix} 0 & \cdots & 0 & B \end{bmatrix}.$$

If we stack these matrices vertically (note that $\mathbf{D}_{60}$ is a zero matrix), we get the complete matrix:

$$\mathbf{D}_6 = \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ B & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & \cdots & B & 0 \\ 0 & 0 & \cdots & 0 & B \end{bmatrix}.$$

The same applies for $\mathbf{E}_{66i}$ and $\mathbf{P}_{16i}$, $i = 1, \ldots, N$, but instead using A and I, respectively.

Formulation (4.2) allows to consider a broad range of multistage problems, besides setpoint stabilization and trajectory tracking. Further examples are MPC for systems with dead time [81], and for systems with known-ahead disturbances [11, 82]. A different class of multistage problems that can also be considered are moving horizon estimation (MHE) problems [3, Ch. 4]. Next we show that the considered multistage formulation can be expressed in a particular type of parametric optimization problem.

4.2. Reformulation as a condensed optimization problem

We describe how the multistage setup (4.2) can be brought into a form suitable for direct use for optimization.

4.2.1. Reformulation as a general QCQP

The multistage setup (4.2) is in its most general form a quadratically constrained quadratic program (QCQP). With some slight modifications, formulation (4.2) can represent second-order cone programs (SOCP), a class of problems related to QCQPs. Quadratic programs (QPs) and linear programs (LPs) are special cases of QCQPs [31, Ch. 4]. How the problem data needs to be organized to compute a solution depends on the solver. Many general purpose solvers (e.g. CVXOPT and ECOS) require the data

in a form similar to

$$\begin{aligned} \underset{\mathbf{u}}{\text{minimize}} \;& \frac{1}{2}\mathbf{u}^\mathsf{T} H \mathbf{u} + g(\mathbf{p})^\mathsf{T}\mathbf{u} \\ \text{subject to} \;& \frac{1}{2}\mathbf{u}^\mathsf{T}\overline{H}_i\mathbf{u} + \overline{g}_i(\mathbf{p})^\mathsf{T}\mathbf{u} \leq d_i(\mathbf{p}), \quad i = 0, \ldots, n_{\text{QCQP}} - 1. \end{aligned} \tag{4.5}$$

Some solvers may also accept the data in this form with some minor changes. For example, the tailored QP solvers qpOASES and the ALM+FGM algorithm presented in Section 3.3 require input box constraints to be separated from the rests of inequality constraints and accept double-sided constraints. However, recent tailored approaches require the data in non-standard forms that allows them to better exploit the sparse structure of MPC, in particular for problems with long horizons, see [22, 23, 32].

We are aiming at having a condensed QCQP which is well suited for embedded applications (see Subsection 3.1.4). This requires the elimination of the equality constraints. The following Lemma provides a way to achieve this.

Lemma 1. *Problem* (4.2) *with its equality constraints subject to the restrictions* (4.4) *can be expressed as an optimization problem without equality constraints of the form:*

$$\begin{aligned} \underset{\mathbf{u}}{\textit{minimize}} \;& \sum_{j\in\mathcal{Q}} (\|\hat{\vartheta}_j\|^2_{\hat{\mathbf{M}}_j} + \mathbf{n}_j^\mathsf{T}\hat{\vartheta}_j^-) \\ \textit{subject to} \;& \hat{\vartheta}_j \leq 0, && \textit{for all } j \in \mathcal{I}_a, \\ & \|\hat{\vartheta}_{ji}\|^2_{\mathbf{M}_{ji}} + \mathbf{n}_{ji}^\mathsf{T}\hat{\vartheta}_{ji}^- \leq 0, && \textit{for all } i \in \mathcal{V}_j,\ j \in \mathcal{I}_q, \end{aligned} \tag{4.6}$$

with the vectors $\hat{\vartheta}_j$, $\hat{\vartheta}_j^-$ *for all* $j \in \mathcal{Q} \cup \mathcal{I}$ *given by:*

$$\hat{\vartheta}_j = \hat{\mathbf{c}}_j + \hat{\mathbf{D}}_j\mathbf{u} + \sum_{k\in\mathcal{P}} \hat{\mathbf{P}}_{kj}\mathbf{p}_k, \tag{4.7}$$

and $\hat{\mathbf{M}}_j$ *being a block diagonal matrix.*

Proof. The main task is to replace each vector sequence $\mathbf{e}_j, j \in \mathcal{E}$ in all $\vartheta_j, j \in \mathcal{Q} \cup \mathcal{I}$, i.e. to replace each equality where it correspond in the cost or inequality constraints. The first step is to form the vectors $\mathbf{e}_j$ from each $\mathbf{e}_{ji}$. The core of the formulation is the representation of vector (4.3), for which we define the vector sequence:

$$\vartheta_j = \mathbf{c}_j + \mathbf{D}_j\mathbf{u} + \sum_{k\in\mathcal{P}} \mathbf{P}_{kj}\mathbf{p}_k + \sum_{k\in\mathcal{E}} \mathbf{E}_{kj}\mathbf{e}_k, \tag{4.8}$$

with $\vartheta_j \in \mathbb{R}^{r_{\vartheta_j}}$, and $r_{\vartheta_j} = \sum_{i \in \mathcal{V}_j} r_{\vartheta_{ji}}$. Note that, because $\mathbf{e}_{ji}$ is equal to ϑ_{ji}, $\mathbf{e}_{ji}$ also depends on the rest of auxiliary vectors $\mathbf{e}_j$ (including itself). This means that for any ϑ_{ji} we actually have the nested form:

$$\begin{aligned}\vartheta_{ji} &= \mathbf{c}_{ji} + \mathbf{D}_{ji}\mathbf{u} + \sum_{k \in \mathcal{P}} \mathbf{P}_{kji}\mathbf{p}_k + \sum_{c \in \mathcal{E}} \mathbf{E}_{cji}\mathbf{e}_c, \\ &= \mathbf{c}_{ji} + \mathbf{D}_{ji}\mathbf{u} + \sum_{k \in \mathcal{P}} \mathbf{P}_{kji}\mathbf{p}_k + \sum_{c \in \mathcal{E}} \mathbf{E}_{cji}(\mathbf{c}_c + \mathbf{D}_c\mathbf{u} + \sum_{k \in \mathcal{P}} \mathbf{P}_{kc}\mathbf{p}_k + \sum_{k \in \mathcal{E}} \mathbf{E}_{kc}\mathbf{e}_k),\end{aligned}$$

which in turn can be simply written as

$$\vartheta_{ji} = \hat{\mathbf{c}}_{ji} + \hat{\mathbf{D}}_{ji}\mathbf{u} + \sum_{k \in \mathcal{P}} \hat{\mathbf{P}}_{kji}\mathbf{p}_k + \sum_{k \in \mathcal{E}} \hat{\mathbf{E}}_{kji}\mathbf{e}_k, \tag{4.9}$$

with

$$\begin{aligned}\hat{\mathbf{c}}_{ji} &= \mathbf{c}_{ji} + \sum_{c \in \mathcal{E}} \mathbf{E}_{cji}\mathbf{c}_c, \\ \hat{\mathbf{D}}_{ji} &= \mathbf{D}_{ji} + \sum_{c \in \mathcal{E}} \mathbf{E}_{cji}\mathbf{D}_c, \\ \hat{\mathbf{P}}_{kji} &= \mathbf{P}_{kji} + \sum_{c \in \mathcal{E}} \mathbf{E}_{cji}\mathbf{P}_{kc}, \\ \hat{\mathbf{E}}_{kji} &= \sum_{c \in \mathcal{E}} \mathbf{E}_{cji}\mathbf{E}_{kc}.\end{aligned} \tag{4.10}$$

Note that the nested form (4.9) has the same structure as the general form (4.3). This implies that several levels of nesting are possible. Eventually, when all equality constraints have been replaced (we have reached the deepest level of nesting), $\mathbf{e}_{ji} = \vartheta_{ji}$ does not depend on $\mathbf{e}$. This is guaranteed by the restrictions (4.4) imposed on $\mathbf{e}$. Therefore, we obtain the following vector:

$$\begin{aligned}\hat{\vartheta}_{ji} &= \mathbf{c}_{ji} + \mathbf{D}_{ji}\mathbf{u} + \sum_{k \in \mathcal{P}} \mathbf{P}_{kji}\mathbf{p}_k + \sum_{c \in \mathcal{E}} \mathbf{E}_{cji}\mathbf{e}_c, \\ &= \mathbf{c}_{ji} + \mathbf{D}_{ji}\mathbf{u} + \sum_{k \in \mathcal{P}} \mathbf{P}_{kji}\mathbf{p}_k + \sum_{c \in \mathcal{E}} \mathbf{E}_{cji}(\mathbf{c}_c + \mathbf{D}_c\mathbf{u} + \sum_{k \in \mathcal{P}} \mathbf{P}_{kc}\mathbf{p}_k),\end{aligned}$$

which can be written as:

$$\hat{\vartheta}_{ji} = \hat{\mathbf{c}}_{ji} + \hat{\mathbf{D}}_{ji}\mathbf{u} + \sum_{k \in \mathcal{P}} \hat{\mathbf{P}}_{kji}\mathbf{p}_k.$$

Here $\hat{\mathbf{c}}_{ji}$, $\hat{\mathbf{D}}_{ji}$ and $\hat{\mathbf{P}}_{kji}$ are given by (4.10). Thus, the full vector $\hat{\vartheta}_j$ correspond to (4.7).

It follows that problem (4.2) can be equivalently represented by (4.6) with the vectors $\hat{\vartheta}_j$, $\hat{\vartheta}_j^-$ given by (4.7) for all $j \in \mathcal{Q} \cup \mathcal{I}$, and $\hat{\mathbf{M}}_j$ being a block diagonal matrix given by:

$$\hat{\mathbf{M}}_j = \begin{bmatrix} \mathbf{M}_{j0} & 0 & \cdots & 0 \\ 0 & \mathbf{M}_{j1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{M}_{j(n_j-1)} \end{bmatrix}.$$

□

Although Lemma 1 provides a way to eliminate the equality constraints, the resulting problem is in a form unsuitable for direct use in standard numerical solvers. We next discuss a further reformulation leading to a standard formulation.

4.2.2. Reformulation as a QCQP in standard form

Problem (4.6) is clearly a quadratically constrained quadratic program. However, we want to have it in standard form (4.5). The following Lemma guarantees that this transformation can always be done.

Lemma 2. *The optimization problem* (4.6) *is equivalent to:*

$$\begin{aligned} &\underset{\mathbf{u}}{\text{minimize}} \ \frac{1}{2}\mathbf{u}^\intercal H \mathbf{u} + g(\mathbf{p})^\intercal \mathbf{u} \\ &\text{subject to} \ \frac{1}{2}\mathbf{u}^\intercal \overline{H}_i \mathbf{u} + \overline{g}_i(\mathbf{p})^\intercal \mathbf{u} \leq d_i(\mathbf{p}), \quad i = 0, \ldots, n_{QCQP} - 1, \end{aligned} \tag{4.11}$$

where H and $\overline{H}$ are constant matrices, and the vectors $g(\mathbf{p})$, $\overline{g}_i(\mathbf{p})$, $d_i(\mathbf{p})$ are of the form

$$g(\mathbf{p}) = \hat{g} + \sum_{k \in \mathcal{P}} \hat{\mathbf{G}}_k \mathbf{p}_k.$$

Proof. Let us first consider how to bring the cost $\sum_{j \in \mathcal{Q}} (\|\hat{\vartheta}_j\|^2_{\hat{\mathbf{M}}_j} + \mathbf{n}_j^\intercal \hat{\vartheta}_j^-)$ in (4.6) into the

form $\frac{1}{2}\mathbf{u}^\intercal H\mathbf{u} + g(\mathbf{p})^\intercal\mathbf{u}$ in (4.11). For any quadratic form we have

$$\begin{aligned}
\|\hat{\vartheta}_j\|^2_{\hat{\mathbf{M}}_j} &= \hat{\vartheta}_j^\intercal\hat{\mathbf{M}}_j\hat{\vartheta}_j,\\
&= (\hat{\mathbf{c}}_j + \hat{\mathbf{D}}_j\mathbf{u} + \sum_{k\in\mathcal{P}}\hat{\mathbf{P}}_{kj}\mathbf{p}_k)^\intercal\hat{\mathbf{M}}_j(\hat{\mathbf{c}}_j + \hat{\mathbf{D}}_j\mathbf{u} + \sum_{k\in\mathcal{P}}\hat{\mathbf{P}}_{kj}\mathbf{p}_k),\\
&= (\hat{\mathbf{D}}_j\mathbf{u})^\intercal\hat{\mathbf{M}}_j\hat{\mathbf{D}}_j\mathbf{u} + 2(\hat{\mathbf{c}}_j + \sum_{k\in\mathcal{P}}\hat{\mathbf{P}}_{kj}\mathbf{p}_k)^\intercal\hat{\mathbf{M}}_j\hat{\mathbf{D}}_j\mathbf{u} + \hat{d}(\mathbf{p}),\\
&= \mathbf{u}^\intercal\hat{\mathbf{D}}_j^\intercal\hat{\mathbf{M}}_j\hat{\mathbf{D}}_j\mathbf{u} + 2(\hat{\mathbf{D}}_j^\intercal\hat{\mathbf{M}}_j(\hat{\mathbf{c}}_j + \sum_{k\in\mathcal{P}}\hat{\mathbf{P}}_{kj}\mathbf{p}_k))^\intercal\mathbf{u} + \hat{d}(\mathbf{p}),\\
&= \mathbf{u}^\intercal\hat{\mathbf{D}}_j^\intercal\hat{\mathbf{M}}_j\hat{\mathbf{D}}_j\mathbf{u} + 2(\hat{\mathbf{D}}_j^\intercal\hat{\mathbf{M}}_j\hat{\mathbf{c}}_j + \sum_{k\in\mathcal{P}}\hat{\mathbf{D}}_j^\intercal\hat{\mathbf{M}}_j\hat{\mathbf{P}}_{kj}\mathbf{p}_k)^\intercal\mathbf{u} + \hat{d}(\mathbf{p}),
\end{aligned} \tag{4.12}$$

where $\hat{d}(\mathbf{p}) = (\hat{\mathbf{c}}_j + \sum_{k\in\mathcal{P}}\hat{\mathbf{P}}_{kj}\mathbf{p}_k)^\intercal\hat{\mathbf{M}}_j(\hat{\mathbf{c}}_j + \sum_{k\in\mathcal{P}}\hat{\mathbf{P}}_{kj}\mathbf{p}_k)$. The cost for this term is then

$$\sum_{j\in\mathcal{Q}}\|\hat{\vartheta}_j\|^2_{\hat{\mathbf{M}}_j} = \mathbf{u}^\intercal(\sum_{j\in\mathcal{Q}}\hat{\mathbf{D}}_j^\intercal\hat{\mathbf{M}}_j\hat{\mathbf{D}}_j)\mathbf{u} + 2\sum_{j\in\mathcal{Q}}(\hat{\mathbf{D}}_j^\intercal\hat{\mathbf{M}}_j\hat{\mathbf{c}}_j + \sum_{k\in\mathcal{P}}\hat{\mathbf{D}}_j^\intercal\hat{\mathbf{M}}_j\hat{\mathbf{P}}_{kj}\mathbf{p}_k)^\intercal\mathbf{u}, \tag{4.13}$$

where we omitted the terms that do not depend on the minimization variable (i.e. $\hat{d}(\mathbf{p})$). Similarly, for any affine term

$$\begin{aligned}
\hat{\vartheta}_j^- &= \hat{\mathbf{D}}_j^-\mathbf{u} + \hat{\mathbf{c}}_j^- + \sum_{k\in\mathcal{P}}\hat{\mathbf{P}}_{kj}^-\mathbf{p}_k,\\
&= \hat{\mathbf{D}}_j^-\mathbf{u} + \hat{\mathbf{d}}_j^-(\mathbf{p}),
\end{aligned} \tag{4.14}$$

with $\hat{\mathbf{d}}_j^-(\mathbf{p}) = \hat{\mathbf{c}}_j^- + \sum_{k\in\mathcal{P}}\hat{\mathbf{P}}_{kj}^-\mathbf{p}_k$, the cost is simply given by

$$\sum_{j\in\mathcal{Q}}\mathbf{n}_j^\intercal\hat{\vartheta}_j^- = (\sum_{j\in\mathcal{Q}}\mathbf{n}_j^\intercal\hat{\mathbf{D}}_j^-)\mathbf{u}, \tag{4.15}$$

where we omitted the term $\hat{\mathbf{d}}_j^-(\mathbf{p})$. The cost $\sum_{j\in\mathcal{Q}}(\|\hat{\vartheta}_j\|^2_{\hat{\mathbf{M}}_j} + \mathbf{n}_j^\intercal\hat{\vartheta}_j^-)$ is equivalent to $\sum_{j\in\mathcal{Q}}\|\hat{\vartheta}_j\|^2_{\hat{\mathbf{M}}_j} + \sum_{j\in\mathcal{Q}}\mathbf{n}_j^\intercal\hat{\vartheta}_j^-$, i.e. the addition of (4.13) and (4.15). From this and from (4.13), the Hessian matrix in (4.11) is given by

$$H = 2\sum_{j\in\mathcal{Q}}\hat{\mathbf{D}}_j^\intercal\hat{\mathbf{M}}_j\hat{\mathbf{D}}_j. \tag{4.16}$$

Similarly, the (transposed) gradient vector in (4.11) consist of the factors in the terms

that are affine with respect to $\mathbf{u}$ in (4.13) and (4.15), that is:

$$\begin{aligned} g(\mathbf{p}) &= 2\sum_{j\in\mathcal{Q}}(\hat{\mathbf{D}}_j^{\mathsf{T}}\hat{\mathbf{M}}_j\hat{\mathbf{c}}_j + \sum_{k\in\mathcal{P}}\hat{\mathbf{D}}_j^{\mathsf{T}}\hat{\mathbf{M}}_j\hat{\mathbf{P}}_{kj}\mathbf{p}_k) + \sum_{j\in\mathcal{Q}}\hat{\mathbf{D}}_j^{-\mathsf{T}}\mathbf{n}_j, \\ &= \sum_{j\in\mathcal{Q}}(2\hat{\mathbf{D}}_j^{\mathsf{T}}\hat{\mathbf{M}}_j\hat{\mathbf{c}}_j + \hat{\mathbf{D}}_j^{-\mathsf{T}}\mathbf{n}_j) + \sum_{k\in\mathcal{P}}\sum_{j\in\mathcal{Q}}2\hat{\mathbf{D}}_j^{\mathsf{T}}\hat{\mathbf{M}}_j\hat{\mathbf{P}}_{kj}\mathbf{p}_k. \end{aligned}$$

This can be concisely written as

$$g(\mathbf{p}) = \hat{g} + \sum_{k\in\mathcal{P}}\hat{\mathbf{G}}_k\mathbf{p}_k, \tag{4.17}$$

with

$$\hat{g} = \sum_{j\in\mathcal{Q}}(2\hat{\mathbf{D}}_j^{\mathsf{T}}\hat{\mathbf{M}}_j\hat{\mathbf{c}}_j + \hat{\mathbf{D}}_j^{-\mathsf{T}}\mathbf{n}_j),$$

and

$$\hat{\mathbf{G}}_k = \sum_{j\in\mathcal{Q}}2\hat{\mathbf{D}}_j^{\mathsf{T}}\hat{\mathbf{M}}_j\hat{\mathbf{P}}_{kj}.$$

It is important to observe that in (4.17) the gradient vector is given by a constant vector plus a linear combination of the parameters.

Let us now consider the inequality constraints in (4.11). There are two important differences between the cost and the constraint functions. The first difference is that in the cost all terms independent of the optimization variable are discarded, whereas in the constraint these terms are gathered in $d(\mathbf{p})$. The second difference is that the cost is a scalar, whereas in the description (4.2) we have a combination of vector and scalar inequalities, identified by $j \in \mathcal{I}_a$ and $j \in \mathcal{I}_q$, respectively.

Let us first focus on the scalar constraints, which in this case are quadratic inequality constraints. The Hessian matrix $\overline{H}_i$ and gradient vector $\overline{g}_i(\mathbf{p})$, $i = 0, \ldots, n_{\text{QCQP}} - 1$ in (4.11) are computed similarly as H and $g(\mathbf{p})$ in the cost, as described above using (4.12), (4.14), (4.16) and (4.17). The main difference is that $\overline{H}_i$ and $\overline{g}_i(\mathbf{p})$ are not summation of several terms. We have therefore that the Hessian for each inequality is given by $\overline{H}_{ji} = 2\hat{\mathbf{D}}_{ji}^{\mathsf{T}}\hat{\mathbf{M}}_{ji}\hat{\mathbf{D}}_{ji}$ for all $i \in \mathcal{V}_j,\ j \in \mathcal{I}_q$, cf. (4.16). The gradient vectors $\overline{g}_{ji}(\mathbf{p})$ can be constructed similarly from (4.17). The remaining term $d_i(\mathbf{p})$ in the inequality in (4.11) is computed by adding $\hat{d}_{ji}(\mathbf{p})$ in (4.12) and $\mathbf{n}_{ji}^{\mathsf{T}}\hat{\mathbf{d}}_{ji}^{-}(\mathbf{p})$ from (4.14) for $i \in \mathcal{V}_j,\ j \in \mathcal{I}_q$. □

4.2.3. Special case: condensed QP

In the following we will restrict our attention to QPs. A great variety of multistage formulations can be represented as QPs, such as nominal MPC for setpoint stabilization of linear systems (see [3, 7])

A QP is a particular case of QCQP (4.11) in which all constraints are affine. Affine inequalities are a special case of the quadratic inequalities with $\overline{H}_i = 0$ and $\overline{g}_i(\mathbf{p}) = \overline{g}_i$. Unlike the quadratic constraints, all affine constraints can be stacked together in one single vector inequality representation $C\mathbf{u} \leq c(\mathbf{p})$. For the inequality constraint we have from (4.14):

$$C = \begin{bmatrix} \hat{\mathbf{D}}_{n_{\mathcal{Q}}} \\ \hat{\mathbf{D}}_{n_{\mathcal{Q}}+1} \\ \vdots \\ \hat{\mathbf{D}}_{n_{\mathcal{Q}}+n_{\mathcal{I}_a}-1} \end{bmatrix}, \tag{4.18}$$

and

$$c(\mathbf{p}) = -\left(\begin{bmatrix} \hat{\mathbf{c}}_{n_{\mathcal{Q}}} \\ \hat{\mathbf{c}}_{n_{\mathcal{Q}}+1} \\ \vdots \\ \hat{\mathbf{c}}_{n_{\mathcal{Q}}+n_{\mathcal{I}_a}-1} \end{bmatrix} + \sum_{k \in \mathcal{P}} \begin{bmatrix} \hat{\mathbf{P}}_{kn_{\mathcal{Q}}} \\ \hat{\mathbf{P}}_{k(n_{\mathcal{Q}}+1)} \\ \vdots \\ \hat{\mathbf{P}}_{k(n_{\mathcal{Q}}+n_{\mathcal{I}_a}-1)} \end{bmatrix} \mathbf{p}_k \right). \tag{4.19}$$

Recall the definition of the set of indexes for the affine inequality constraints: $\mathcal{I}_a = \{n_{\mathcal{Q}}, \ldots, n_{\mathcal{Q}} + n_{\mathcal{I}_a} - 1\}$, where $n_{\mathcal{I}_a}$ denotes the number of affine inequality constraints.

The MPC setup (4.2) then has no quadratic constraints ($\mathcal{I}_q = \emptyset$) and can be brought into this form:

$$\begin{aligned} \underset{\mathbf{u}}{\text{minimize}} \;\; & \mathbf{u}^\intercal H \mathbf{u} + g(\mathbf{p})^\intercal \mathbf{u} \\ \text{subject to} \;\; & C\mathbf{u} \leq c(\mathbf{p}). \end{aligned} \tag{4.20}$$

Here H, $g(\mathbf{p})$, C and $c(\mathbf{p})$ are given by (4.16) to (4.19), respectively.

QP (4.20) has a particular data structure, which allows to further exploit some properties of optimization algorithms, like the ones discussed in Chapter 3, and in particular the ALM+FGM algorithm from Section 3.3.

Characteristics of the data

One of the main characteristic of the QP (4.20) is that the Hessian matrix H is constant. This allows to compute off-line the eigenvalues of this matrix, which is generally a com-

plex computation. The eigenvalues play a key role in the solution of many optimization algorithm, as discussed in Chapter 3. Similarly, the constraint matrix C is also constant. This again is exploited by some of the algorithms presented in Chapter 3.

A second characteristic of problem (4.20) is that the gradient vector $g(\mathbf{p})$ and inequality constraint vector $c(\mathbf{p})$ are simply the addition of a constant vector plus a linear combination of the parameters. Again, this allows to compute off-line all the involved matrices that are constant. Then, the on-line computation of g and c is limited to only matrix-vector operations, as discussed next.

On-line computational requirements

Of interest here is the requirements of computing the parametric terms $g(\mathbf{p})$ and $c(\mathbf{p})$, given by (4.17) and (4.19), respectively. These vectors need to be computed on-line by the MPC controller once per sampling period.

Let us first consider $g(\mathbf{p}) \in \mathbb{R}^{r_u}$. Each parameter $\mathbf{p}_k \in \mathbb{R}^{r_{p_k}}$ is multiplied by a matrix $\hat{\mathbf{G}}_k \in \mathbb{R}^{r_u \times r_{p_k}}$, $k \in \mathcal{P}$. The result of each product is then added to the vector $\hat{g} \in \mathbb{R}^{r_u}$. This leads to a total number of operations of

$$n_{\mathcal{P}} r_u + r_u \sum_{k \in \mathcal{P}} r_{p_k}.$$

Similarly for $c(\mathbf{p}) \in \mathbb{R}^{r_\vartheta^{\mathcal{I}_a}}$, where $r_\vartheta^{\mathcal{I}_a} = \sum_{j \in \mathcal{I}_a} r_{\vartheta_j}$ represents the number of inequality constraints, each parameter $\mathbf{p}_k \in \mathbb{R}^{r_{p_k}}$ is multiplied by a matrix $\hat{\mathbf{P}}_k \in \mathbb{R}^{r_\vartheta^{\mathcal{I}_a} \times r_{p_k}}$, $k \in \mathcal{P}$. The result of each product is then added to the vector $\hat{\mathbf{c}} \in \mathbb{R}^{r_\vartheta^{\mathcal{I}_a}}$. It follows that the total number of operations becomes

$$n_{\mathcal{P}} r_\vartheta^{\mathcal{I}_a} + r_\vartheta^{\mathcal{I}_a} \sum_{k \in \mathcal{P}} r_{p_k}.$$

In many cases, the vectors g and c may depend respectively on a subset $\mathcal{P}_g$ and $\mathcal{P}_c$ of the parameters set $\mathcal{P}$. This implies that the matrices $\hat{\mathbf{G}}_k, k \in \mathcal{P} - \mathcal{P}_g$, and $\hat{\mathbf{P}}_k, k \in \mathcal{P} - \mathcal{P}_c$ are zero. Exploiting this fact allows to save unnecessary computations.

The main advantage of expressing a problem in the form (4.20) is that, because most terms have been computed off-line, the computation of the QP is very fast (compared to computing all terms on-line). This also has the advantage that the on-line code is very simple and is limited to simple matrix vector additions and multiplications.

The main disadvantage of the presented approach is that it may require some effort to go from the general multistage formulation (4.2) to the condensed QP (4.20). Nevertheless, the transformation is systematic and can be realized via software. We present a tool that achieves this in Chapter 5.

4.3. High-level multistage specification language

Note that formulation (4.2) is a non-intuitive abstract representation of a multistage problem. This section introduces a domain specific language (DSL) that is based on (4.2) but brings back the intuitiveness of MPC.

This DSL allows to specify a broad class of linear multistage problems in a natural way. The considered DSL consist of the following main parts: the declaration of variables, a cost function to be minimized, and a set of equality and inequality constraints.

We explain the main components of the DSL using a rather simple example. Consider the general regulation to the origin problem with input and state constraints. The MPC formulation is given by (2.15), which we repeat here for ease of reference:

$$\begin{aligned}
\underset{\mathbf{u}}{\text{minimize}} \quad & \frac{1}{2}\sum_{j=0}^{N-1}(\|x_j\|_Q^2 + \|u_j\|_R^2) + \frac{1}{2}\|x_N\|_P^2 \\
\text{subject to} \quad & x_{j+1} = Ax_j + Bu_j, && j = 0, ..., N-1, \\
& \underline{u} \leq u_j \leq \overline{u}, && j = 0, ..., N-1, \\
& \underline{e} \leq E_x x_j + E_u u_j \leq \overline{e}, && j = 0, ..., N-1, \\
& \underline{f} \leq F x_N \leq \overline{f}, \\
& x_0 = x^k.
\end{aligned} \tag{4.21}$$

This problem is specified by the plain text presented in Listing 4.1. The DSL closely resembles the mathematical representation of the MPC problem. We describe in the following how each component is specified in the DSL.

Declaration of variables

The `parameters` keyword identifies all the vectors, or vector sequences, that are to be specified online. The `variable` keyword identifies the optimization variable. The `auxs` keyword denotes the equality constraint variable that needs to be eliminated to get a

condensed formulation. It is assumed that the restrictions (4.4) hold. For all these identifiers, the number of elements in the sequence, as well as the length of each vector element, need to be specified. This is done using the format `v[a:b](m)`, where `v` is the name of the sequence, `m` is the length of each vector in the sequence, and `a` and `b` denote the index of the first and last element of the sequence. To refer to the element `i` of this sequence, we use the notation `v[i]`. In the case that a parameter is not a sequence (i.e. it is a single vector), only the vector length need to be specified, i.e. `v(m)`. To refer to this vector no indexing is required in later parts of the problem specification, i.e. it suffices to write `v`.

All other non-numeric symbols that are not keywords are automatically taken as having a fixed, although yet unknown, shape and value. These fixed symbols need to be specified off-line at a later stage. For the example in Listing 4.1, symbols with fixed values are the prediction horizon `N`, the system matrices `A` and `B`, the vector lengths `m` and `n`, etc.

The cost function

The keyword `minimize` identifies the text following it as the cost function to be minimized. Several special keywords are accepted, for example the `sum(h[i], i=a:b)` denotes the summation of the real valued functions h_i for $i = a, \ldots, b$. The keyword `quad(v,M)` denotes the quadratic form $\|v\|_M^2$. As we only deal with problems that can be transformed into a QCQP (or a related form such as SOCP, QP or LP), the cost function is restricted to be the summation of linear and quadratic terms.

Listing 4.1: Domain specific language example: regulation to the origin.

```
1  parameters xk(n)
2  auxs x[0:N](n)
3  minimize sum(quad(x[j],Q) + quad(u[j],R), j=0:N-1) + quad(x[N],P)
4  variable u(m,N)
5  subject to
6    x[j+1] = A*x[j] + B*u[j], j=0:N-1
7    u_lb <= u[j] <= u_ub, j=0:N-1
8    e_lb <= Ex*x[j] + Eu*u[j] <= e_ub, j=0:N-1
9    f_lb <= F*x[N] <= f_ub
10   x[0] = xk
```

The constraints

The DSL allows to specify equality constraints, and single- and double-sided inequality constraints. The constraints follow the format in (4.2). As seen in the example, the equality and inequality constraints optionally accept an index variable and its range. The constraints can be quadratic or affine. For example, the prediction inherent in the MPC problem is described by the mathematical expression

$$x_{j+1} = Ax_j + Bu_j, \;\; j = 0, \ldots, N-1,$$

which in the DSL can be written as `x[j+1] = A*x[j] + B*u[j], j=0:N-1`.

It is also possible to denote terminal state constraints like $f_{lb} \leq Fx_N \leq f_{ub}$ using the expression `f_lb <= F*x[N] <= f_ub`.

Comparison to other tools

The proposed DSL is very similar to the language used in CVXGEN, and QCML. One key difference is the explicit declaration of an `auxs` variable used to eliminate equality constraints. Another difference is that we define the `parameters` exclusively as the vectors that are not constant (i.e. $\mathbf{p}_k$ in (4.2)). Finally, in our formulation the dimensions are inferred from the given variables.

4.3.1. Code generation of condensed formulation

We shortly describe how all presented ideas are applied to generate C code for a condensed formulation. A problem description $\mathbb{P}$ written in the DSL needs first to be created. The problem needs to fit formulation (4.2) and needs to comply with (4.4). An example of a valid $\mathbb{P}$ is shown in Listing 4.1. To create code for a particular instance of $\mathbb{P}$ we require the data $\mathbb{D}$. In our example $\mathbb{D}$ consist of the numerical values of A, B, N, n, etc. With this information, the sequences $\mathbf{c}$, $\mathbf{D}$ $\mathbf{P}$ and $\mathbf{E}$ that define all ϑ_{ji} in $\mathbb{P}$ can be computed, as presented in the MPC example found in Section 4.1. By applying Lemmas 1 and 2 we obtain the dense matrices H and C in the standard QP formulation (4.20), as well as $\hat{g}$, $\hat{\mathbf{G}}_k$ in (4.17) and $\hat{c}$, $\hat{\mathbf{P}}_k$ in (4.19). Finally, based on this data, C code is generated for the functions $g(\mathbf{p})$ and $c(\mathbf{p})$. This procedure is summarized in Algorithm 8.

Require: problem description $\mathbb{P}$ and instance data $\mathbb{D}$
1: Parse $\mathbb{P}$ and bring into form (4.2)
2: Create vectors (4.3) using $\mathbb{D}$
3: Apply Lemma 1 to eliminate equality constraints
4: Apply Lemma 2 to get H, C, $g(\mathbf{p})$, and $c(\mathbf{p})$ as in (4.20).
5: Generate C code to compute $g(\mathbf{p})$, and $c(\mathbf{p})$ using (4.17) and (4.19).

Algorithm 8: Code-generation of condensed formulation

4.4. Summary

In this chapter we presented a general formulation of multistage problems (e.g. MPC and MHE) that can be expressed as a condensed parametric optimization program, namely as a QCQP. We showed that for multistage problems like (4.2) for which the equality constraints satisfy the conditions (4.4), it is always possible to get a general QCQP without equality constraints (4.6). Furthermore, we showed that for any given set of parameters, a condensed QCQP in standard form can be computed following some rules.

Because expressing an multistage problem in the form (4.2) is in general complicated, we introduced a domain specific language that closely resembles the way how MPC and MHE problems are expressed mathematically. The presented language specification is not limited to represent problems like (4.2), which only accept parameters as vectors. For instance, problems where some of the matrices are specified as parameters are also considered, allowing to specify, e.g. problems with time-varying dynamics. We will nevertheless limit our discussion in the following chapters to problems like (4.2) mainly for the following reasons. First, it is a problem formulation that can be used in a broad range of embedded applications. Second, problems like (4.2) can be represented in the form (4.5), which can be efficiently computed on-line. Furthermore, several popular general purpose and tailored solvers require the problem formulation in a form similar to (4.5).

The next chapter describes a software tool that takes a problem in the DSL representation of (4.2) and generates C-code to solve QPs like (4.20). The proposed DSL captures well the main characteristic of various multistage formulations.

5. muAO-MPC: a free code generation tool for embedded MPC

The automatic code generation presented in this chapter is based on a software tool developed in the framework of this thesis called *µAO-MPC* as first presented in [25]. *µAO-MPC* is a Python-based free software whose main function is to automatically generate ready-to-use MPC controller code.

In this chapter, we do not intend to explain in detail how to *use* the tool (for information on the use of the software we refer to [102]). We rather elaborate on the principles on which *µAO-MPC* is based and the goals it pursues. We first discuss the core features of *µAO-MPC* followed by the generation of C-code tailored for the MPC problem described in Chapter 4. This tailored code exploits many of the properties of the ALM+FGM algorithm described in Section 3.3. We expand our discussion with methods to enhance the performance of the on-line MPC controller. We close this chapter with an example.

5.1. Core features

µAO-MPC explicitly takes into account the limitations of embedded computers discussed in Section 2.1. The main limitations of embedded hardware can be roughly split into two parts: low amount of memory and low numerical throughput. The former includes ROM and RAM. The latter not only means low CPU clock frequencies, but also limitations on the arithmetic operations and precision. An additional limitation of embedded computers is related to software. Usually, the compilers used for embedded targets have a different set of features than their regular (non-embedded) counterparts. Those limiting factors have been considered in the development of *µAO-MPC*.

The MPC setup is described in a intuitive manner, using the language described in Section 4.3. From this description, we generate code for solving a condensed QP. The

code generated by *µAO-MPC* is portable C code that does not depend on external libraries. Moreover, it only relies on additions and multiplications. This last fact simplifies the implementation of an algorithm using fixed-point arithmetic. A few examples of fixed-point arithmetic using the proposed algorithm can be found in [25, 64, 83]. *µAO-MPC* supports by default the generation of code based on single and double precision floating-point and fixed-point arithmetics. Additionally, *µAO-MPC* provides Python, MATLAB and SIMULINK interfaces to the generated C code. Further extensions are also possible, such as adapting *µAO-MPC* to generate C-code that can be used as input for field-programmable gate array synthesis software [103].

µAO-MPC takes a strict separation approach for *forming* and *solving* the problem. In this context, we refer to forming a problem (i.e. a QCQP or a related form) to the process of computing the numerical values of the Hessian matrix, gradient vector, etc. from a parametric problem description and a given set of parameters, like (4.5). Forming the problem is based on the setup presented in Section 4.1. The process of solving the problem involves using an optimization algorithm, like the ones presented in Chapter 3, to find the solution to a problem that has been already formed. *µAO-MPC* by default generates code for both processes.

Although *µAO-MPC* can automatically generate C-code to form QCQPs, SOCPs, and QPs, at the time of this writing the automatically generated solver can only deal with QPs. Nevertheless, the strict separation of former and solver enables the use of third-party solvers to deal with the generated QCQPs and SOCPs formers (see [74]). We recently developed a proof-of-concept implementation of an automatically generated solver for SOCPs based on a primal-barrier interior point method [74]. The discussion of this implementation is beyond the scope of this work. In the following we focus on automatic code generation of MPC problems that can be expressed as QPs.

The default solver used by *µAO-MPC* is the ALM+FGM algorithm described in Section 3.3. Although forming and solving the QP are independent of each other, they both explicitly take into account the embedded systems limitations. Moreover, this separation allows the user of *µAO-MPC* to use any QP solver, while still using the intuitive high-level formulation presented in Section 4.3. algorithms.

By default, solving the QP is based on the algorithm presented in Section 3.3. As the fast gradient method performs particularly well for problems with Hessians with low condition numbers, *µAO-MPC* includes an off-line tool that helps reduce the condition number of the QP Hessian. This tool is based on the method presented in [100].

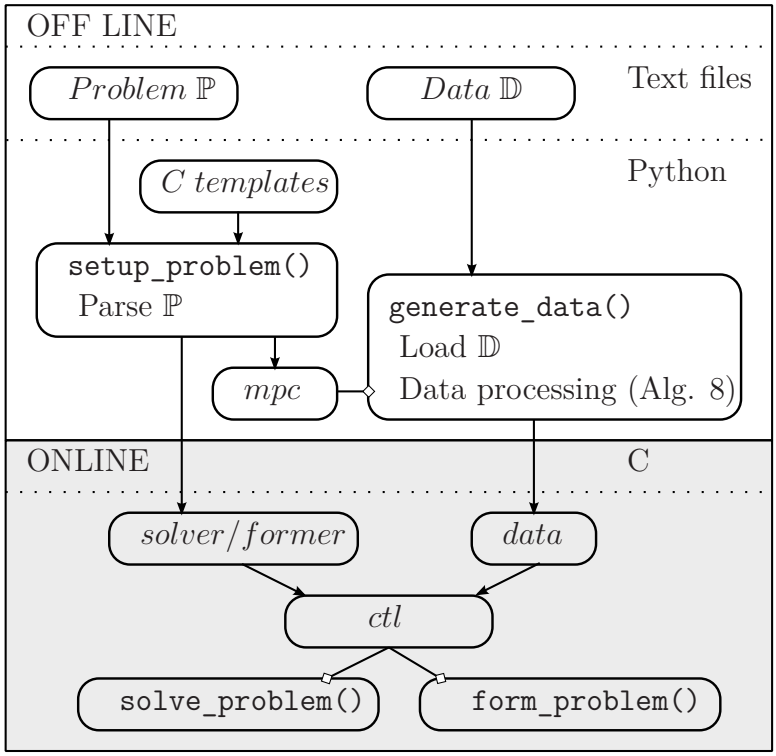

Figure 5.1.: *μAO-MPC*'s code generation data flow. Arrows denote the flow of data. Diamonds denote an object's functions.

As a last remark, it is important to note that *μAO-MPC* is not intended to be used as a general purpose QP solver. Given that the QPs used in a MPC context have many particularities, we consider *μAO-MPC* as a tool to easily implement efficient MPC controllers on embedded hardware. More specifically, *μAO-MPC* usually quickly finds approximate solutions to the QPs that deliver good controller performance. We further discuss this topic in Subsection 5.2.3.

5.2. Automatic generation of C code

The automatic code generation can be broadly split in parsing, preprocessing, and generation of code for functions and data (see Figure 5.1). Parsing refers to interpreting a string of symbols specified by a language grammar. In Section 4.3 we specified the language grammar of *μAO-MPC*. In the preprocessing step, a user-provided problem description $\mathbb{P}$ is parsed and analyzed to identify certain patterns that fit into specified

structures (the function `setup_mpc_problem` in Figure 5.1). These structures identify if a problem is a SOCP, a QCQP, or a QP, for example. Furthermore, we might look for the presence (or absence) of input box constraints, and mixed constraints. Finally, the code generation of functions and data is based on the use of template files. A template file is a text file that is almost like C code, except that it contains certain keywords that are to be replaced with appropriate values after preprocessing. Once all keywords have been replaced, the result is C code tailored for the specified optimization problem. To any problem $\mathbb{P}$ at least one data $\mathbb{D}$ must be specified. *μAO-MPC* separates the generation of code and data. This allows certain flexibility during early testing and simulation phases, where the data changes frequently, but the problem structure remains the same. This is illustrated in the middle part of Figure 5.1, where the function `setup_mpc_problem` creates the object *mpc*, which provides the method `generate_data` that transform the data $\mathbb{D}$ for problem $\mathbb{P}$ using the methods presented in Chapter 4, in particular Algorithm 8. As illustrated in the bottom part of Figure 5.1, the generated code consist of C-code for the solver and former functionality, which is separated of the C-code for data. However they both are use together in the *ctl* object to solve a problem using *μAO-MPC*'s default solver (the function `solve_problem`), or to just form a problem (the function `form_problem`) to be solved by a third-party solver.

5.2.1. Forming and solving the condensed QP

In Subsection 2.1.2 we discussed the advantages and disadvantages of dynamic and static memory allocation. We exploit the inherent flexibility of automatic code generation to produce code with either dynamic or static memory allocation. The former can be used for simulations, where the size of some arrays changes frequently (for example, by changing the horizon length). The latter is appropriate for real-time deployment once the data size has been fixed. Except for how memory is allocated, the generated code in both cases is functionally identical.

As previously mentioned, the functionality of the code is split into forming the QP and solving the QP. Although the code for forming and solving are independent of each other, they are by default used in as a bundle. Forming the QP is one of the most computationally expensive parts of the (off-line) code generation. However, this results in the (on-line) code for forming the QP being extremely simple, as only a few matrix vectors operations need to be performed on-line.

As with any optimization algorithm, the ALM+FGM will excel in some applications but in some others it will be outperformed by a different algorithm. *μAO-MPC* by design allows the use of a different QP solvers, instead of the ALM+FGM. The only restriction on the solver is that it needs to accept the data in the form provided by *μAO-MPC*. Notable examples that can be used without modifications are MATLAB's quadprog, and qpOASES. With some minor rearrangement of constraints matrices, solvers like CVXOPT can also be used.

5.2.2. Solving the QP with the ALM+FGM algorithm

The default algorithm used by *μAO-MPC* is described in Section 3.3. Of particular interest here is its application to the MPC problem, discussed in Subsection 3.3.3.

Computing off-line data

One of the main design objectives of *μAO-MPC* is to perform off-line as many operations as possible. These operations need to be performed only once, avoiding the need to be computed by an embedded processor on-line at each sampling time. The FGM requires a Lipschitz constant L and a strong convexity parameter ϕ. How the values of these parameters are computed depends on the particular type of problem. In Subsection 3.3.3 we identified two cases: input constraints and mixed constraints. In the former case, L and ϕ can be computed from the eigenvalues of H. In the latter case, L also depends on the constraint matrix $\hat{C}$ and the penalty parameter μ. In both cases, the scalar sequence ν can be computed off-line based on the values of L and ϕ.

We have assumed so far that we have enough information to compute L, ϕ and therefore ν. From the MPC problem (4.20), we have H and C constant (therefore $\hat{C}$ is also constant).

Note from (3.34) and (3.47) that we need to multiply the gradient of the cost function by the factor L^{-1}. These operation can be *partly* performed off-line. We say partly because although we have a constant H, the gradient vector depends on the parameters $\mathbf{p}$. Nevertheless, from (4.17) we can multiply $\hat{g}$ and $\hat{\mathbf{G}}$ by L^{-1} off-line. Note however that in the case of mixed constraints, the projected gradient step (3.47) depends on the term (3.46). In this case, we also need to compute $L^{-1}\hat{C}$.

Selection of on-line parameters

All the required constants are automatically computed by the code generation procedure. This leaves only two parameters of the algorithm to be selected by the user: the number of *inner* and *outer* iterations (refer to Algorithm algorithm 7, Section 3.3). These two parameters effectively set the computation time of the algorithm. Unlike other tailored algorithms, *µAO-MPC* by design does *not* check how good the computed solution is. The emphasis is put on a deterministic computation time independently of the reached suboptimality. This is motivated by the observation that, in practical applications, even rough solutions often deliver acceptable performance (see [32] and Chapter 6 for some examples).

5.2.3. Further controller performance improvements

In Section 2.1 we presented the main limitations of embedded applications. Furthermore, in Section 3.1 we discussed the main characteristics of the MPC optimization problems that can be exploited. In Subsection 3.3.4 we saw how the ALM+FGM takes into account the particularities of MPC for embedded applications. Here we briefly discuss how *µAO-MPC* implements these ideas.

In Subsection 3.3.4 we discussed how a lower condition number of the Hessian κ_H implies a faster convergence of the ALM+FGM algorithm. Furthermore, we have shown how L, ϕ and μ are related to κ_H. *µAO-MPC* implements an off-line procedure aimed at reducing the value of κ_H [100]. This procedure exploits the available degrees of freedom in the MPC problem to find suitable weighting matrices that deliver a similar controller performance and, at the same time, reduce the condition number of the Hessian. More specifically, for a given trajectory considered typical for the application, a nonlinear optimization problem finds new weighting matrices of an MPC controller that minimize the distance between the desired and the new trajectory. One constraint of the optimization problem is that the condition number of the Hessian must be lower than a certain threshold, typically a value below 100.

In Subsection 3.3.4 we mentioned the effects of quantization and round-off errors in different numeric representations. *µAO-MPC* allows to automatically generate code that uses either floating-point arithmetic (either single or double precision) or fixed-point arithmetic. The latter case is, however, only done as a proof of concept. The correct execution of arithmetic operations is not guaranteed (e.g. there are no checks for

overflow). In Chapter 6 we demonstrate with an example that fixed-point MPC control is possible with *µAO-MPC* under certain conditions.

5.3. Examples: code generation for a microcontroller

This section exemplifies the automatic code generation for a microcontroller. We will leave the discussion of the implementation on real systems for the next chapter.

5.3.1. Setup description

We consider two examples that are representative of some of the problem classes handled by *µAO-MPC*. MPC is run on a real-time embedded operating system (RTOS) [104]. We take the gcc ARM embedded toolchain (gcc-arm) [105] to create the binary executables from the generated code.

We compare *µAO-MPC* against CVXGEN. Although CVXGEN deals with general convex optimization problems and it is not aimed at microcontroller applications (the memory requirements are relatively high), it allows us to highlight some of the features and limitations of *µAO-MPC*. The code generated by CVXGEN is able to handle the considered problems and run in real-time on a µC without modifications, except for using single precision instead of the default double precision floating-point. The systems being controlled are a simple robotic arm, and a more complex aircraft. The detailed specifications of both systems are provided in Appendix B.1.

Robotic arm

We consider the example found in [83]. It is a model of a real 2-link robotic arm with four states (two of which are constrained) and two constrained inputs. The discretization time is 4 ms, and we use a prediction horizon of 5 steps.

Aircraft

As a second example we consider a more complex system, a model of a Cessna 500 aircraft linearized at constant speed [7]. The problem has 4 states, and one input. There are inputs, outputs, and slew-rate constraints. The slew-rate constraint is considered by

adding an additional state to the discrete-time formulation. The horizon length is 10 steps. The system is sampled at 0.5 s intervals.

5.3.2. Considered Embedded Hardware

We considered two different embedded test platforms based on the 32-bit ARM Cortex-M family of microcontrollers. As a low-cost platform we use a *STM32VLDISCOVERY* board based on a ARM Cortex-M3 with a clock rate of 24 MHz, 64 kB of flash and 8 kB of RAM. As a high-performance platform we use a *STM32F4DISCOVERY* board based on a ARM Cortex-M4 with a clock rate of 168 MHz, 1 MB of flash, and 192 kB of RAM. It incorporates a single precision floating-point unit, as well as DSP capabilites.

5.3.3. Results

The executable binaries of the code generated by CVXGEN were too large to fit into the flash memory of the low-cost board. Therefore, we only used that board to explore the possibility of using the fixed-point option of *μAO-MPC*. Compared to floating-point arithmetic, fixed-point arithmetic decreases computation time by nearly four times. Note that the use of fixed-point was only possible on the robotic arm example. In general, fixed-point arithmetic is limited to problems with good numerical properties (e.g. well scaled and well conditioned).

Table 5.1 summarizes the memory demands of both algorithms in the high-performance μC. The flash requirements of the CVXGEN binaries increase rapidly with problem complexity and are several times larger than the binaries of *μAO-MPC*. The amount of stack required by the CVXGEN controller thread was in the kilobytes range for both examples, two orders of magnitude more than that of our implementation. As a comparison, a recommended value for very simple tasks is 256 bytes per thread [104]. In an RTOS each thread or task requires its own stack space, which is statically allocated in RAM and usually determined empirically. Here we use a common heuristic procedure. We start by assign a typical value of 256 bytes to the RTOS thread in which the MPC algorithm is running. If the thread runs (does not run), we half (double) the amount of MPC's thread RAM. We repeat until the algorithm stops (starts) executing correctly.

We observed that both algorithms have deterministic execution times, i.e. each requires (nearly) constant time to perform a fixed number of iterations. We measure the

Table 5.1.: Memory demands on the high-performance μC

	Robotic arm		Aircraft	
Resource	*μAO-MPC*	CVXGEN	*μAO-MPC*	CVXGEN
Flash memory	11 kB	80 kB	13 kB	220 kB
Stack memory	16 B	2048 B	32 B	4096 B

time required to reach an acceptable controller performance. Conventionally, to compare the speed of two optimization algorithms, we measured the time they require to reach a certain suboptimality level. For control purposes, the approach discussed in Subsection 3.1.2 is more suitable to determine the nominal controller performance. For comparison, we consider a trajectory that starts at an initial state and ends at the origin. We first use CVXOPT to find the optimal input and state trajectories. We then compare them with the approximate input and state trajectories obtained by the embedded platform. Each trajectory consists of $T_N + 1$ points. We start with a very low number of maximum iterations that the algorithm can perform, thus limiting the performance and the maximum computation time. To measure how good the performance of the controller in each case is, we compute the cost (cf. (3.1)):

$$J_p = \sum_{k=0}^{T_N-1} \|x_k^* - x_k^{\sim}\|_Q^2 + \|u_k^* - u_k^{\sim}\|_R^2,$$

where x_k^*, u_k^* are the state x and input u at point k in the optimal trajectory computed by CVXOPT. Similarly, $x_k^{\sim}, u_k^{\sim}$ are the approximated state and input computed by one of the algorithms. We then gradually increase the maximum number of iterations and measure the time it takes the algorithm to perform that many iterations (denoted as t_{cpu}). For CVXGEN, we have increased the maximum number of iterations by 1 each time. For *μAO-MPC* we try to use values that will help us in the discussion (many are selected to approximately double the previous execution time). For the aircraft example we start at $x_0 = [0, 0, 0, -400, 0]$ with $T_N = 40$. For the robotic arm example we use $x_0 = [-1, 0, 1, 0]$ and $T_N = 400$. The results are shown in Fig. 5.2.

5.3.4. Discussion

Fig. 5.2 shows that overall for the considered examples, *μAO-MPC* reaches good controller performance with less computational effort than CVXGEN. Exemplarily, the

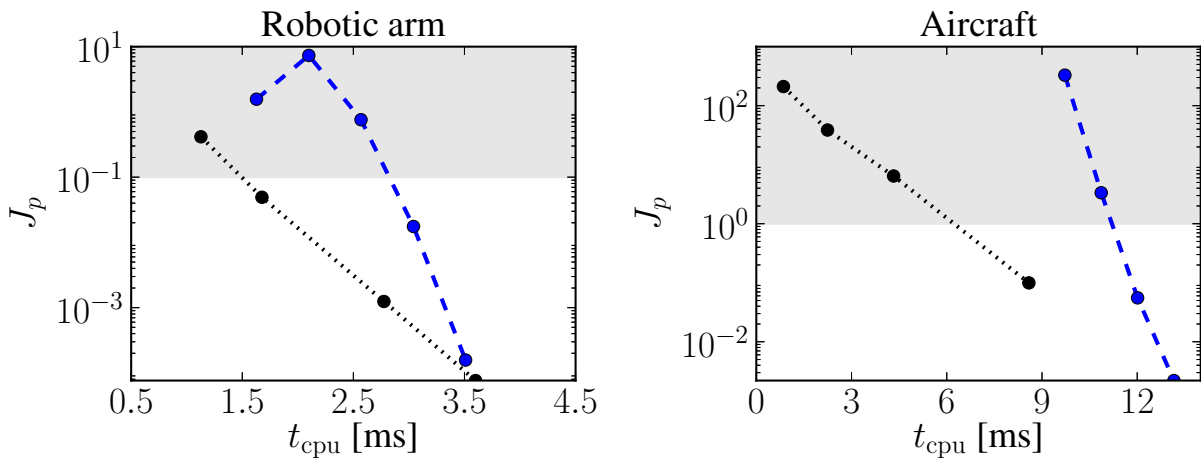

Figure 5.2.: Comparison of the controller performance with limited computation time for the considered systems. *μAO-MPC* is shown in dotted lines, CVXGEN in dashed lines. The shaded area indicates where the performance is considered poor in each case. Note the logarithmic scale with respect to J_p.

robotic arm example shows that to obtain $J_p \approx 10^{-1}$ (which we arbitrarily define as acceptable in this case using a visual criterion) our algorithm requires about half the time of CVXGEN (approximately 1.5 vs. 2.8 ms). However, if solutions of much higher precision are needed ($J_p < 10^{-4}$), CVXGEN will require less time than *μAO-MPC*. The same trend holds for the more complex aircraft example.

From a different perspective, if we limit the computation time for the robotic arm to be below 1.7 ms, from Fig. 5.2 we can expect the performance of *μAO-MPC* to be better than that of CVXGEN. In Fig. 5.3, the trajectories for this particular case are compared to the exact trajectory, which confirms what we expected. The plots correspond to CVXGEN limited to perform 3 iterations, and *μAO-MPC* with $i_{ex} = 6$, $j_{in} = 3$ and $\mu = 2000$. Similarly, fixing the time to approximately 9 ms for the aircraft example, *μAO-MPC* yields a better controller performance, as can be seen in Fig. 5.4. In this case, CVXGEN was limited to 9 iterations, and *μAO-MPC* used $i_{ex} = 2$, $j_{in} = 24$ and $\mu = 100$.

There are several reasons that may explain the different controller performances. One reason is that CVXGEN is a general purpose solver that can manage a wider range of convex optimization problems, whereas our algorithm has been specifically tailored for problems like (2.15) (e.g. box constraints are handled very efficiently). Additionally, CVXGEN is based on a primal-dual interior point method and does not implement any

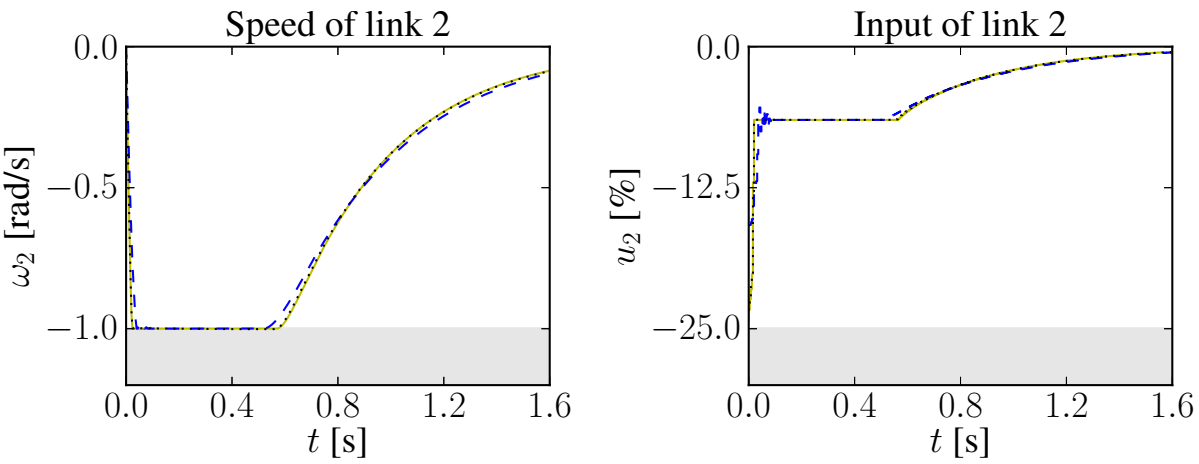

Figure 5.3.: Trajectories for the robotic arm example, for computation time $t_{\text{cpu}} \approx 1.7$ ms. Three trajectories are shown for link 2 of the arm: exact from CVXOPT (solid lines), and the approximate solutions from CVXGEN (dashed), and *μAO-MPC* (dotted). The solid and dotted lines are almost indistinguishable. The shaded area denotes constraints.

warm start strategy, whereas our algorithm can take full advantage of this strategy. Another reason is that our code only relies on additions and multiplications, which are cheap one-cycle operations in the considered μC (Cortex-M4). CVXGEN in contrast, must perform a large amount of divisions every iteration (each division requires several cycles).

As an insight, our MPC optimization algorithm converges faster if the Hessian of the QP is well conditioned. In the aircraft case, the Hessian has a condition number of around 25. Such a low number is, however, not a coincidence. The original aircraft problem, as presented in [7], uses identity matrices as weighting matrices. This results in a Hessian with a condition number in the order of 10^5. We use *μAO-MPC*'s off-line help function discussed in Subsection 5.2.3 to reduce the condition number. Similarly, applying this method to the robotic arm problem we get a condition number of around 2. This allows the use of fixed-point arithmetics, which for the considered low-cost μC increases the numerical throughput four times compared to floating-point arithmetics.

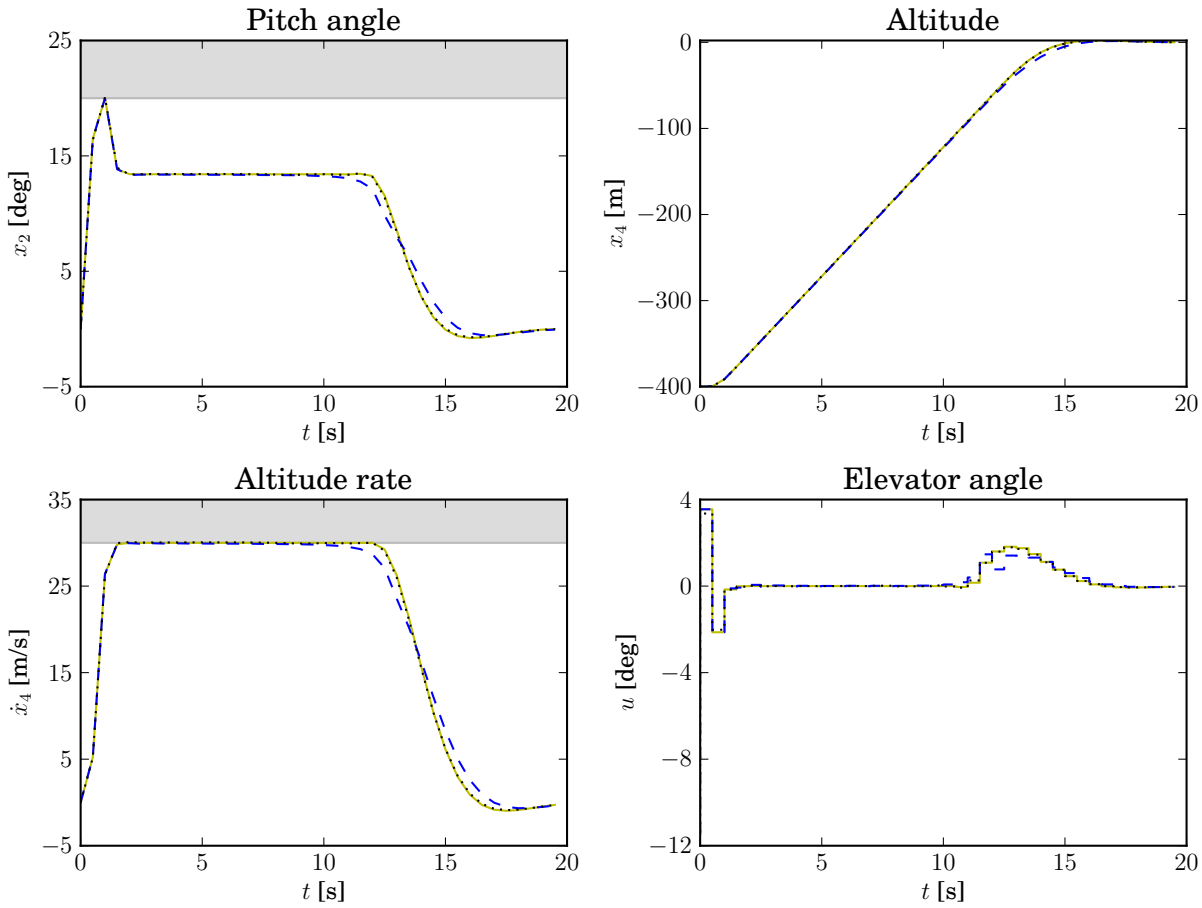

Figure 5.4.: Trajectories for the aircraft example, for computation time $t_{\text{cpu}} \approx 9$ ms. Three trajectories are shown: exact from CVXOPT (solid lines), and the approximate solutions from CVXGEN (dashed), and *µAO-MPC* (dotted). The solid and dotted lines are almost indistinguishable. The shaded area denotes constraints.

5.4. Summary

We presented *µAO-MPC*, a software tool for the automatic code generation of MPC controllers. *µAO-MPC* uses a generic yet intuitive MPC problem specification, based on the ideas presented in Chapter 4. From it, portable library-free C-code is generated for forming and solving a QP. Forming a QP refers to computing the numerical values of the QP matrices given a set of parameters. To solve the formed QP, *µAO-MPC* by default implements a tailored ALM+FGM algorithm as presented in Chapter 3. An optional intermediate step is the off-line numerical conditioning of the problem, which

has a dramatic influence on the algorithm performance and the required number of iterations. The generated code has deterministic maximal computation time, has low requirements on ROM and RAM and, for well-conditioned problems, usually reaches good controller performance with relatively low computational demands.

We presented some examples considering real embedded platforms that demonstrate the scenarios were the ALM+FGM offers a real advantage. More specifically, the algorithm works extremely well for applications that take advantage of warm start, where rough approximate solutions deliver good closed-loop performance, and that are well conditioned. As shown in the simulation examples, the ALM+FGM algorithm may not be of advantage for applications where very precise solutions are required.

Because we follow a strict separation of forming and solving a QP, the C-code generated by *µAO-MPC* can easily be used with a different QP. We have successfully combined *µAO-MPC* with qpOASES, CVXGEN, MATLAB's quadprog and CVXOPT.

The next chapter presents some real applications that show that the ALM+FGM delivers good closed-loop performance for fast mechatronic systems running in hardware with very tight computational constraints.

6. Application examples

In this chapter we demonstrate the suitability of the proposed approach using two application examples. We first discuss the implementation on a low-cost microcontroller. Afterwards, an autonomous-driving vehicle example using a high-performance microcontroller is discussed. *µAO-MPC* has proven effective in several embedded applications, such as gust load alleviation on an aircraft [106], and active vibration attenuation [107, 108].

6.1. Low-end example: A direct current motor

We present an implementation of the proposed ALM+FGM algorithm using a low-cost microcontroller unit (MCU), underpinning that the approach is well suited for low cost platforms. We briefly explain the tuning of the algorithm and discuss the results. The example presented in this section is based on a popular educational platform, the LEGO® MINDSTORM® NXT.

6.1.1. System description

We consider the motion control of a direct current (DC) motor using the NXT platform (Fig. 6.1). Our goal is to use MPC to drive the system to the origin with limited angular speed starting from an arbitrary position.

The DC motor is modeled in continuous time as

$$\ddot{y} = -\frac{1}{T}\dot{y} + \frac{K}{T}w \tag{6.1}$$

where T is the motor time constant, K the input amplification factor, y the rotor's angular position and w the energy input to the system. The continuous-time system in

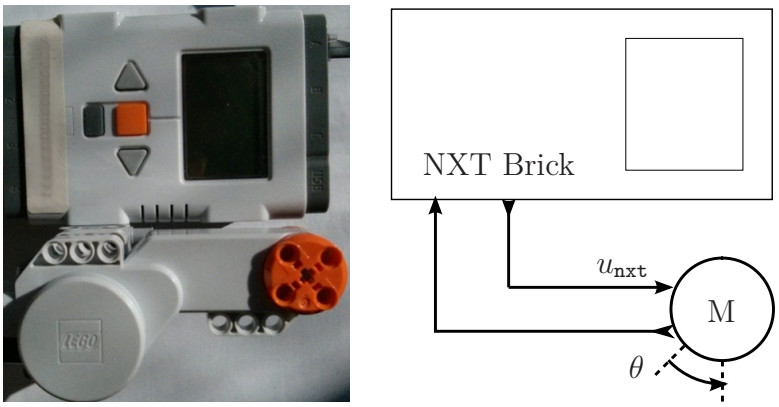

Figure 6.1.: The NXT setup. Left, the NXT brick and motor. Right, a schematic view of the setup.

state-space form is given by $\dot{x}_{\mathtt{nxt}} = A_{\mathtt{nxt}} x_{\mathtt{nxt}} + B_{\mathtt{nxt}} u_{\mathtt{nxt}}$ with:

$$A_{\mathtt{nxt}} = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{1}{T} \end{bmatrix}, \quad B_{\mathtt{nxt}} = \begin{bmatrix} 0 \\ \frac{K}{T} \end{bmatrix}, \tag{6.2}$$

where $T = 0.062\,\mathrm{s}$ and $K = 0.154\,\mathrm{s}^{-1}$. The system state is described by the vector $x_{\mathtt{nxt}} = [\theta \quad \dot{\theta}]^\intercal$ and $u_{\mathtt{nxt}}$ is the energy input (see Fig. 6.1). The state θ denotes the rotor's angular position in radians, and $\dot{\theta}$ represents the rotor's angular speed in radians per second. For demonstration purposes, we want the angular speed to be constrained between $-10 \leq \dot{\theta} \leq 10$. The input $u_{\mathtt{nxt}}$ is a pulse-width-modulated (PWM) voltage in percentage, i.e. it is limited in hardware to be between -100% and 100%. The input is therefore constrained between $-100 \leq u_{\mathtt{nxt}} \leq 100$.

The NXT Brick uses an Atmel AT91SAM7S256 microcontroller which includes an ARM7TDMI processor core, 64 kB RAM, and 256 kB flash memory. The clock frequency is 48 MHz. ARM7 processors are based on legacy architecture which is similar in features to the Cortex-M3 architecture. It has a 32-bit integer arithmetic logic unit, but lacks a floating-point unit (FPU). Floating point operations are therefore emulated by software. ARM7TDMI provides two instruction sets: ARM and Thumb. We only use the former (the standard 32-bit instruction set), as the latter did not deliver any advantage for the considered application. We refer to [109] for further details.

The NXT Brick inputs $u_{\mathtt{nxt}}$ are limited to integer values. Thus, the applied inputs to the motor are quantized. In this case, it would be similar to a 8-bit DAC, with

only $2^8 = 256$ states. This in turn implies a relatively high quantization error (see Section 2.1). In particular, from (2.1), we have that the maximum quantization error in the inputs is $\beta = 0.5\%$. This error is independent of the MPC controller, and limits the overall performance that MPC scheme can achieve even if exact solutions are used. We argue that due to this quantization, we should not aim to obtain exact solutions, as they are computationally expensive. Instead we should compute approximate solutions that are close to β. This criterion offers in many applications a good trade-off between computational time and closed-loop performance.

To deploy the ALM+FGM algorithm in real time, we use the open-source nxtOSEK real-time operating system (RTOS) [110]. nxtOSEK allows us to develop real-time applications written in C for the NXT, using different task scheduling policies with timer resolutions down to 1 ms. The whole RTOS and the application code are first cross-compiled in a standard PC, then downloaded into the flash memory. During run-time, they both (the RTOS and the user application) are copied from flash into RAM. This means, that although we have 256 kB of flash available, we are limited by RAM to 64 kB of memory.

6.1.2. Generating a fast embedded MPC controller

The design of an MPC controller allows some freedom in the choice of application dependent parameters, like the sampling period and horizon length. To simplify the discussion, we take them as fixed to the following values: sampling period of 4 ms and horizon length of 10 steps. These values offer a good trade-off between control performance and available computation time for the considered application.

In an initial test, we determined the computational capabilities of our hardware for the given setup. In real time, we could only perform 1 iteration of Algorithm 6 (the main computational burden of the ALM+FGM method) using single-precision floating-point arithmetic. The use of $Q15.16$ fixed-point arithmetic allows us to perform up to 6 iterations with, however, an inferior numeric precision.

6.1.3. Results

Fig. 6.2 shows the simulated step response of our system for an initial condition $x(t_0)^\intercal = [3.5\ 0]$. The exact trajectory (i.e. computed by exact minimization using CVXOPT)

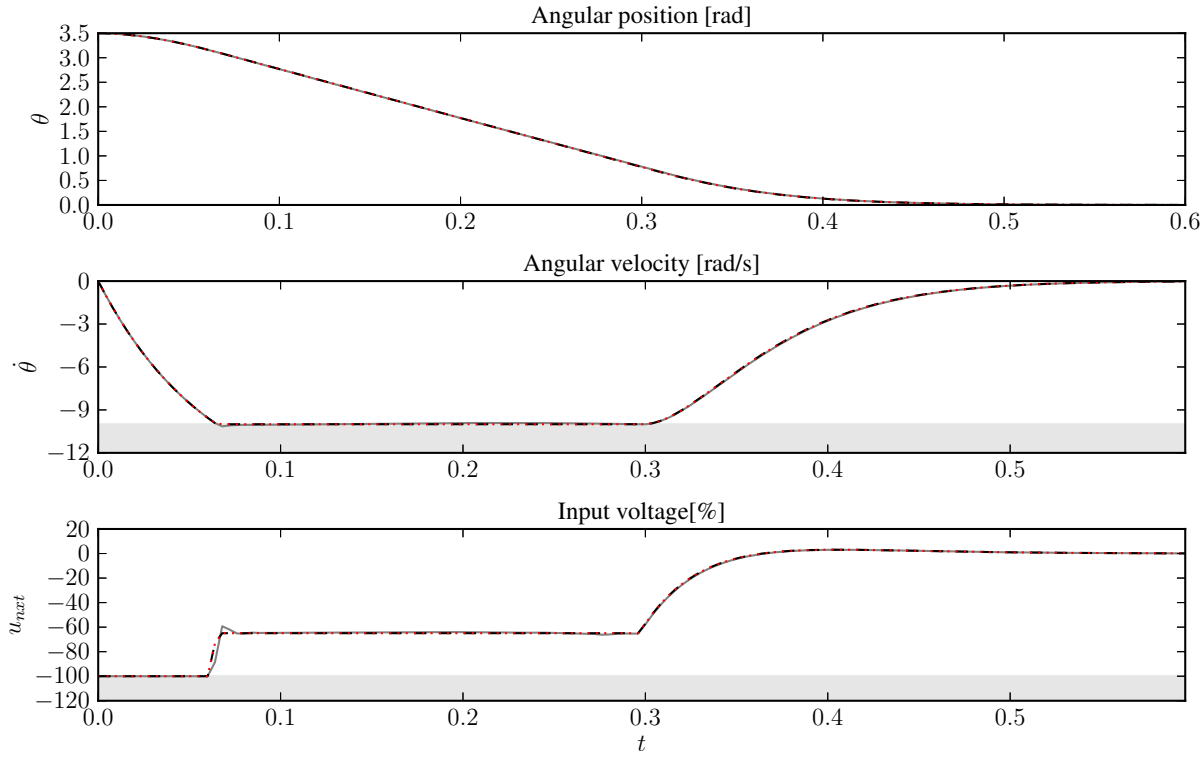

Figure 6.2.: Comparison in simulation of trajectories using exact minimization (black dotted) and approximations by *µAO-MPC* (solid grey).

and the approximated trajectory (computed by *µAO-MPC*) are shown. We perform 3 external iterations and 2 internal iterations of the ALM+FGM algorithm, with warm starting and $Q15.16$. The computation time on the MCU is 3.8 ms and the compiled code size is around 15 kB. We see that the input constraints are always satisfied.

Fig. 6.3 shows the experimental step response for the current setup. Both plots, the simulation and the real experiment use the same *µAO-MPC* code with the same ALM+FGM parameters. The state $\dot{\theta}$ is estimated by a discrete steady-state Kalman filter (see [111]) relying on measurements of the angular position θ. The constraint on $\dot{\theta}$ is violated by less than 10% during the trajectory. This is mainly due to model plant mismatch and state estimation errors. Note the active soft constraints. Even if the constraints are violated and thus rendering the original hard-constrained problem infeasible, the ALM+FGM finds a solution that steers the system back to the original feasible set. Furthermore, the ALM+FGM behaves in this case as an exact penalty method. We can visually confirm this from the simulations, cf. Figure 6.2, where the ALM+FGM delivers identical plots to the original hard-constrained problem when the

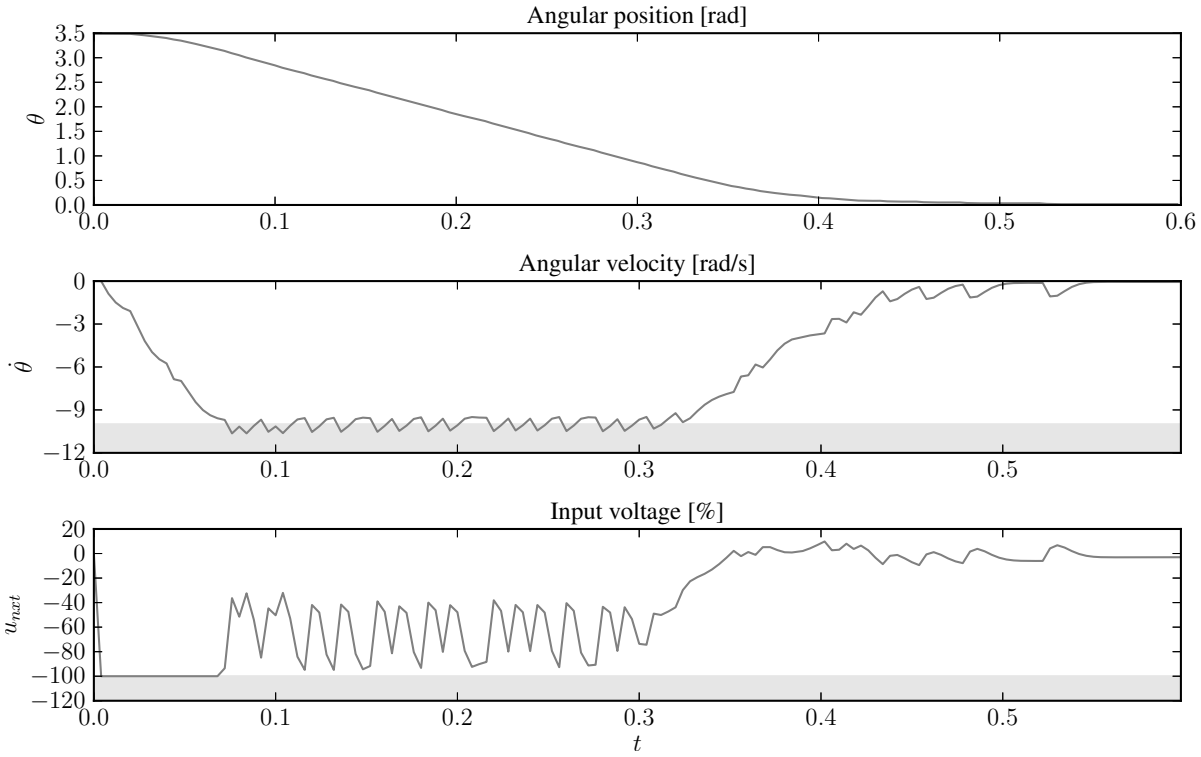

Figure 6.3.: Experimental step response of the NXT DC motor setup using *µAO-MPC*.

constraints are not violated.

As visible, *µAO-MPC* allows us to solve similarly sized problems on low-cost embedded microcontrollers, with limited computational power, low numeric precision, and little memory, using the software tool presented in this document.

6.2. High-performance example: An autonomous vehicle

We considered an autonomous four-wheeled vehicle driving on a continuous belt (Fig. 6.4). The objective is to stabilize the car at different points inside the drive surface while satisfying constraints on the inputs and states. A thorough description of the experimental setup and the validation of the MPC algorithm follows.

6.2.1. System description

The autonomous car is equipped with a high-performance Infineon TriCore TC1796 microcontroller which includes 256 kB RAM, 2 MB flash ROM, a 32-bit FPU and runs

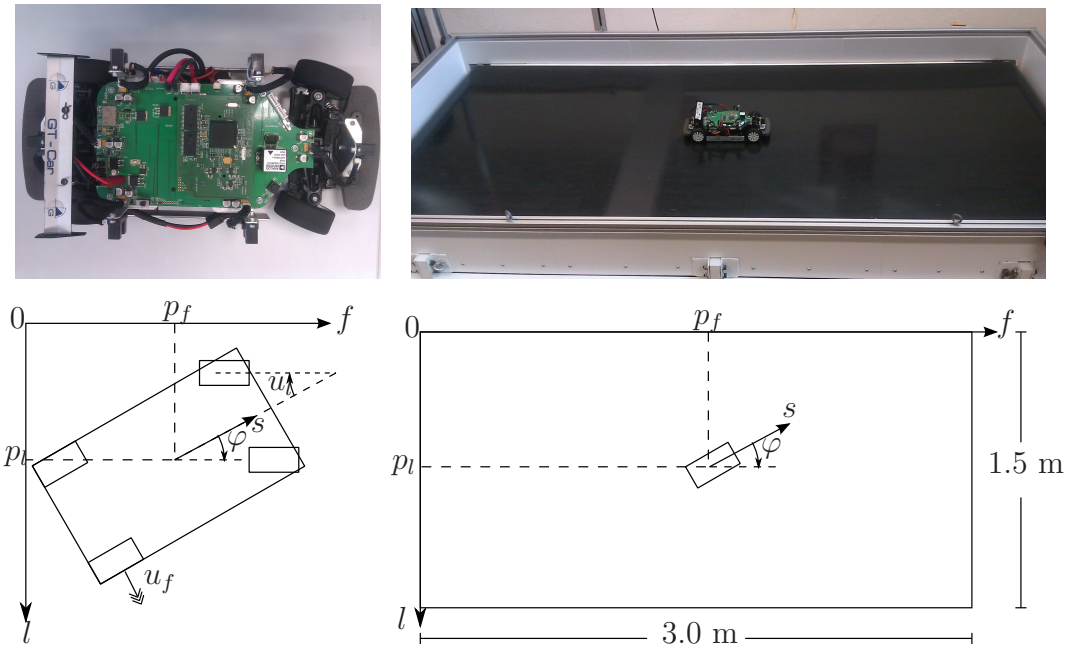

Figure 6.4.: Experimental setup: left, the real car (top) and its schematic showing the system states. On the right, the real belt and car (top) and a schematic view.

with a clock frequency of 150 MHz. The car is equipped with a gyroscope, several distance sensors, and Hall sensors on the rear wheels. Using these sensors, full-state information is provided on-line to the controller by a Kalman filter. The car is on a 1:10 scale: approximately 20 cm long by 10 cm wide. It drives on a continuous belt that has a length of 3.0 m and a width of 1.5 m. The car can position itself within that surface with a clearance to the borders of about 0.3 m, i.e. the effective drive area is $2.4 \times 0.9\,\mathrm{m}^2$. The position of the car is with respect to a coordinate frame fixed to the ground with the origin on the lower left corner of the belt frame (see Fig. 6.4).

The forward and lateral dynamics of the car are nonlinear and coupled. Nevertheless, for a constant speed of the belt, one can linearize the car dynamics. This results in two decoupled linear systems, one for the forward dynamics and another for the lateral dynamics of the car. With the belt speed fixed at 1.3 m/s (about 45 km/h at 1:10 scale), the forward dynamics are described by the continuous time system $\dot{x}_f = A_f^c x_f + B_f^c u_f$, with:

$$A_f^c = \left[\begin{smallmatrix} 0 & 1 \\ 0 & -10 \end{smallmatrix}\right], B_f^c = \left[\begin{smallmatrix} 0 \\ 35 \end{smallmatrix}\right], x_f = \left[\begin{smallmatrix} p_f \\ s \end{smallmatrix}\right].$$

Here p_f is the car forward position with respect to the origin, s the car speed in car's coordinate system, and u_f the applied torque of the motor (refer to Fig. 6.4). Similarly, the lateral dynamics are given by $\dot{x}_l = A_l^c x_l + B_l^c u_l$, with

$$A_l^c = \begin{bmatrix} 0 & 1.8 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -7.7 \end{bmatrix}, B_l^c = \begin{bmatrix} 0 \\ 0 \\ 11.5 \end{bmatrix}, x_l = \begin{bmatrix} p_l \\ \varphi \\ \dot{\varphi} \end{bmatrix},$$

with p_l the lateral position of the car with respect to the origin, φ the angle of the car with respect to the forward axis, $\dot{\varphi}$ the angular velocity of the car, and the u_l the angle of the steering wheels with respect to the car forward axis (refer to Fig. 6.4).

Clearly, two single-input controllers can be designed to stabilize the car. For the sake of discussion, however, we will consider the two-input system given by:

$$A^c = \begin{bmatrix} A_f^c & 0 \\ 0 & A_l^c \end{bmatrix}, B^c = \begin{bmatrix} B_f^c & 0 \\ 0 & B_l^c \end{bmatrix}, x = \begin{bmatrix} x_f \\ x_l \end{bmatrix}, u = \begin{bmatrix} u_f \\ u_l \end{bmatrix}. \tag{6.3}$$

We discretize (6.3) using a zero-order hold on the inputs with a sampling period of 5 ms. We drop the superscript $(\cdot)^c$ in the system matrices to denote the corresponding discrete time matrices, such that $x^+ = Ax + Bu$.

Both input are physically constrained to be in the interval $[-1, 1]$. In the case of u_f, -1 represents full motor brake, and 1 full motor power, with values in between. For u_l, -1 means steering angle completely to the left, while 1 completely to the right.

6.2.2. MPC Implementation

As already stated, the main purpose of this example is to demonstrate the suitability of the presented ALM+FGM algorithm and its implementation using *µAO-MPC* as an embedded MPC controller. We require that the states constraints $-0.1 \leq \varphi \leq 0.1$ are satisfied. Additionally, we introduce inputs constraints such that $-0.15 \leq u_l \leq 0.15$, and $-0.35 \leq u_f \leq 0.05$. These constraints keep the car in the range where the linear model is still sufficiently good. We use the matrices and parameters shown in Appendix B.2 for our MPC controller. In particular, we use a horizon length of 10 steps.

Fig. 6.5 shows a comparison of the exact and the approximate solution trajectories for the considered MPC setup. The exact trajectory is computed using CVXOPT, whereas the approximate trajectory are obtained using *µAO-MPC*. We set the ALM+FGM algorithm to perform just 1 FGM iteration, and 4 ALM iterations plus warm starting. The condition number of the internal problem $\kappa \approx 3.3$ allows to use a very low number

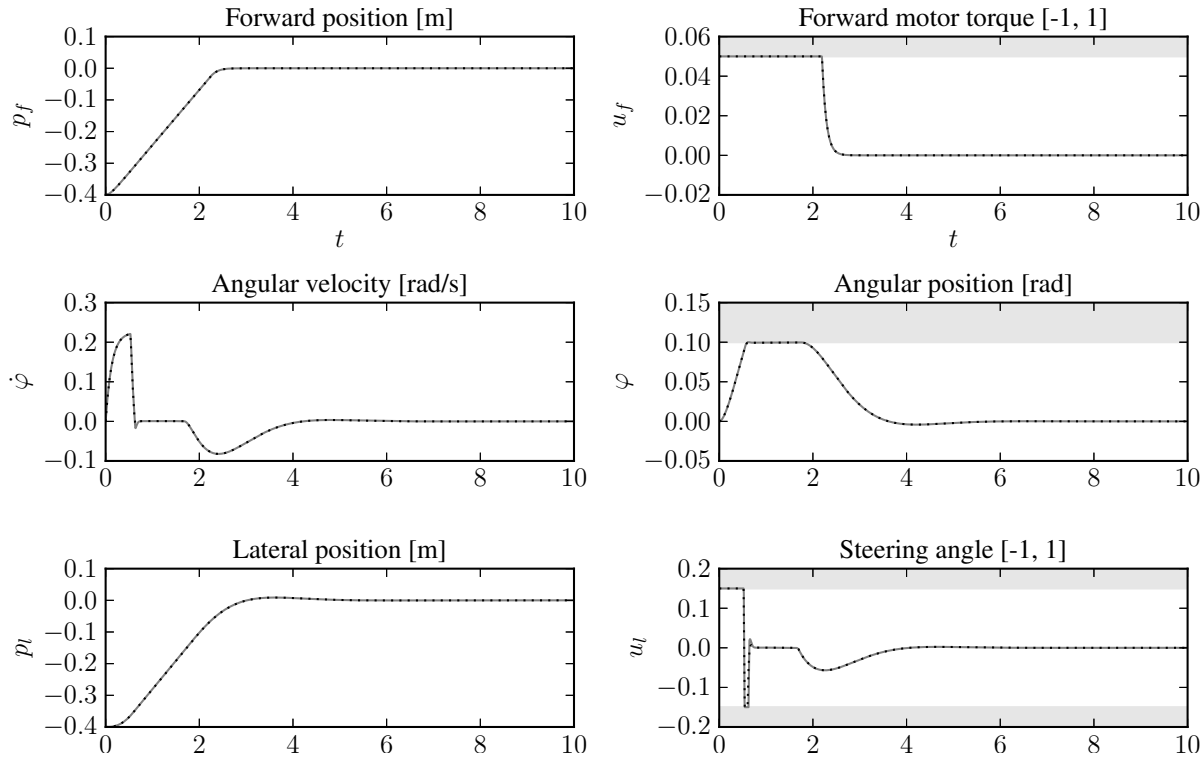

Figure 6.5.: Comparison in simulation of trajectories using exact minimization (black dotted) and approximations by our algorithm (solid grey). The trajectories are almost indistinguishable from each another.

of iterations, and in particular only 1 iteration of the internal loop. From Fig. 6.5 we observe that the approximation seems to be good enough for control.

We proceed to implement the MPC controller on the embedded platform using SIMULINK as development environment. The implementation of MPC is conceptually simple and rather straightforward to do using *µAO-MPC*. For the given setup, the MPC needs about 4 ms to be executed. The rest of the software (state estimation, real-time operating system, etc.) take less than 1 ms to execute. Thus, we are using the maximum number of iterations we could achieve for the given sampling period of 5 ms. The size of the binary file downloaded to the car's ROM is about 30 kB. In Fig. 6.6 we show the behavior of the MPC controller, and we additionally compare it to unconstrained control.

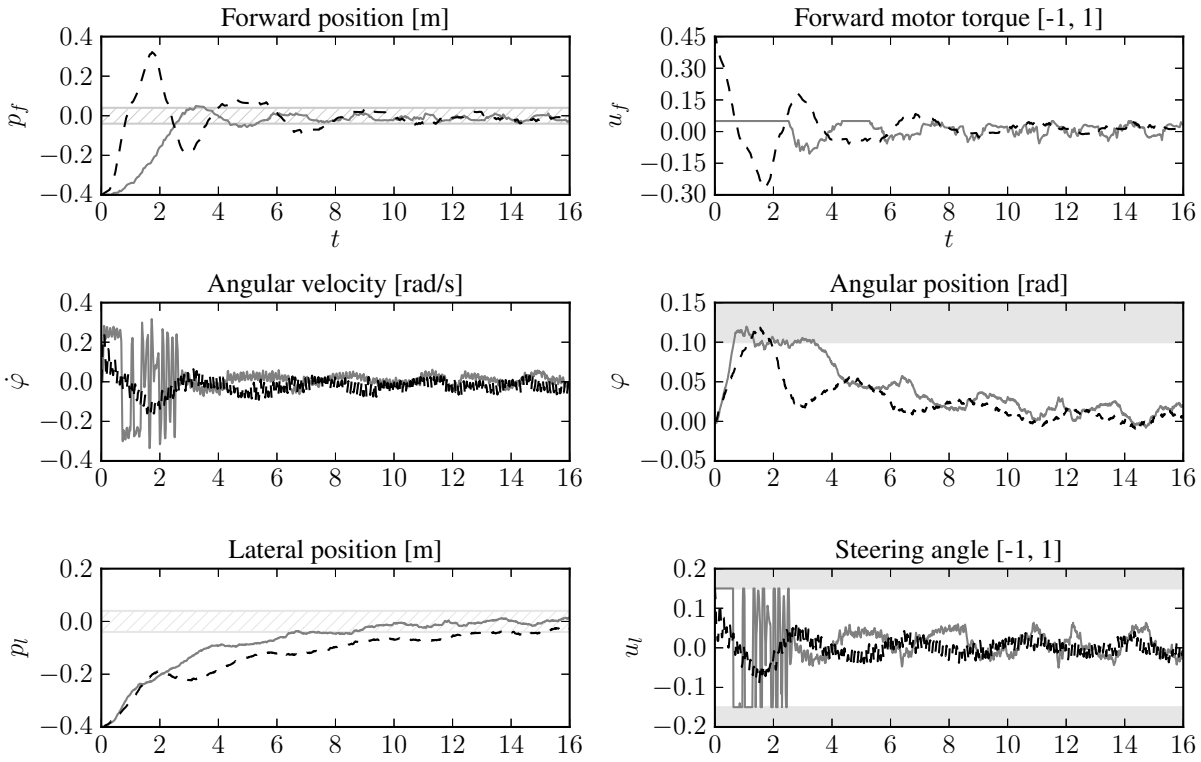

Figure 6.6.: Comparison of experiments using linear unconstrained control (dashed black) and our MPC implementation (solid gray). Shaded areas denote constraints. Dashed areas denote the 10 % setpoint error band.

6.2.3. Discussion

To estimate the low computational requirements of the proposed algorithm, we compare *µAO-MPC* again against CVXGEN. To perform our computational comparison, we use high-performance the STM32F4-DISCOVERY board described in Section 5.3. This platform together with embedded RTOS ChibiOS gives us more flexibility in the simulations and is easily accessible. Thus, it is better suited for comparison purposes. We additionally use a desktop computer equipped with a Intel Core i7-2600 CPU (Ci7-PC) clocked at 3.4 GHz running Ubuntu Linux 12.04. This may give the reader an intuitive feeling of the speed of the algorithm. Note that we compare the algorithms using the controller performance J_p, as defined in (3.1), Subsection 3.1.2, and not the exactness/sub-optimality of the solution $\mathbf{u}^{\sim}$. A thorough comparison of both algorithms is beyond the scope of this work.

In this example, CVXGEN requires a maximum of 4 interior point iterations to get

a value of J_p slightly higher than that of $\mu AO\text{-}MPC$ (i.e. CVXGEN controller delivers slightly worse performance). Under these conditions and on the CM4-μC, we measure a computational time of 0.7 ms for $\mu AO\text{-}MPC$'s ALM+FGM whereas CVXGEN requires 4.3 ms on the CM4-μC. Recall that in the car's μC we determined that $\mu AO\text{-}MPC$ needs 4 ms. Although we have not attempted to run CVXGEN on the car, by cross multiplication we estimate that CVXGEN would require around 24.6 ms in the car's μC.

In the (non real-time) Ci7-PC and for the same conditions, the average computation time of 20000 runs is about 0.007 ms for $\mu AO\text{-}MPC$ and 0.07 ms for CVXGEN. Nevertheless, from the presented examples we observe that our algorithm performs extremely well under conditions frequently found in embedded applications, namely: well conditioned problems that make use of warm starting and require only rough approximations.

With regard to the memory requirements, the code generated by $\mu AO\text{-}MPC$ compiled into a binary of 22 kB in size for the CM4-μC, with a significant part of it being problem data, e.g. the Hessian matrix H takes about 1.6 kB. In this case $\mu AO\text{-}MPC$ needed at most 128 bytes of RAM, an acceptable amount for embedded applications.

Let us now make a few remarks about the comparison of controllers shown in Figure 6.6. We will focus for the moment only on the MPC performance. The first thing to note is that the input constraints on u_l and u_f are always satisfied. The state constraint on φ, on the contrary, is shortly violated at the beginning of the motion (nearly 20% at about 0.5 s), but $\mu AO\text{-}MPC$ quickly brings φ back to the constrained region. This violation of constraints is due to imperfect data (measurements, modeling, etc.) and is to be expected in applications.

An alternative way to satisfy constraints on states is to use an LQR with high penalties on such states. The price to pay is degradation of performance when working near the constraints due to the linear nature of the LQR. In this case, we use two independent (unconstrained) LQR controllers: one for the forward dynamics, and another for the lateral dynamics. Both controllers are compared in Figure 6.6: constrained (MPC) and unconstrained (two LQRs). If we focus first on the forward dynamics, we observe that the LQR presents a much higher overshoot. We have explicitly tuned the controller in this way to make it reach the 10% region of the p_f set-point approximately at the same time as the MPC controller. MPC is still slightly faster, entering the region at about 6 s, whereas the LQR reaches it at 8 s.

In the case of the lateral dynamics, we tune the LQR to explicitly satisfy the constraints on the states, achieving maximum values in φ and $\dot{\varphi}$ similar to that of the

MPC controller. This in turns makes the LQR controller much slower than the MPC in reaching the 10% region of the p_l setpoint. MPC reaches this region at about 9 s, whereas the LQR needs 15 s. The weighting matrices for all controllers can be found in Appendix B.2. Summarizing, in this example when using unconstrained control we faced a trade-off between speed (time to reach the 10% region) and satisfaction of constraints (maximum values of states/inputs). MPC on the contrary, has an additional degree of freedom (explicit constraints) and the speed was mainly determined by the whole setup (dynamics, constraints, controller tuning).

To this end, we introduced artificial constraints to demonstrate the suitability of the presented algorithm and generated code for embedded applications. A more realistic controller implementation may use a saturating LQR for the forward dynamics, given that under certain conditions saturating LQR and MPC provide the same control action [2, Ch. 7]. For the lateral dynamics MPC is still preferred. The QP would be smaller and much faster to solve, which may allow to use longer horizons or higher sampling rates.

6.3. Summary

We demonstrated with two application examples that *µAO-MPC* is well suited for the deployment of MPC controllers for fast embedded applications.

First a rather simple example on a low-cost embedded platform was discussed. We made use of fixed-point arithmetic to speed up computations four fold. Although the optimization problem can be considered in general small, it may not be possible to use explicit MPC due to memory constraints. We showed in simulation that very rough approximations computed by *µAO-MPC* seem to deliver good closed-loop performance in the nominal case. Furthermore, the input and state constraints were satisfied. Good closed-loop performance was also observed in the real case, although the state constraints were violated by less than 10%. The main reason is the uncertainty in the model and state estimation, as well as quantization effects, etc.

As a second example, we considered a more complex setup: a autonomous-driving vehicle. In this case, we relied on a more computationally capable microcontroller. Similarly as in the low-cost case, the simulation results delivered by *µAO-MPC* good. Furthermore, we compare the results to the computational performance of CVXGEN. As

shown, *µAO-MPC* required almost two orders of magnitude less memory than CVXGEN, while at the same time the computational time of *µAO-MPC* was eight times lower to reach the same level nominal close-loop performance. Again, the experiments confirmed that *µAO-MPC* indeed delivers acceptable closed-loop performance.

We additionally compared *µAO-MPC* against an LQR. The experiment showed that for systems with constraints, the approximate solutions of *µAO-MPC* outperformed the unconstrained controller.

7. Conclusions

The implementation of model predictive control (MPC) on low-cost embedded platforms to control systems with fast dynamics has been considered difficult in the past. An MPC algorithm has high computational requirements relative to the computational capabilities of embedded computers. Although there has been several important advancements in optimization algorithms, and the cost of embedded computers is constantly decreasing, the application of MPC on embedded systems, in particular on low-cost hardware, is still challenging.

In this work we presented a software tool called *μAO-MPC* that eases the implementation of MPC schemes in embedded systems by means of automatic code generation. Embedded computers have a series of characteristics and limitations that need to be taken into account to make an efficient MPC implementation. Similarly, the MPC algorithm has its own characteristics that can be exploited. We have focused on MPC setups for linear systems that can be equivalently posed as a convex quadratically constrained quadratic programs (QCQP).

At the core of *μAO-MPC* is a novel optimization algorithm for solving QPs. It is based on an augmented Lagrangian method combined with Nesterov's fast gradient method (ALM+FGM). This algorithm can exploit two MPC specific properties, namely the ALM+FGM can be easily warm started and it naturally deals with soft constraints. Furthermore, the ALM+FGM takes into account many of the characteristics of embedded control systems: deterministic in a temporal sense, has a low memory footprint, and only requires multiplications and additions.

One core feature of *μAO-MPC* is the automatic code generation of an ALM+FGM solver based on a general description of the MPC problem. We proposed a domain specific language that closely resembles the way how MPC problems are expressed mathematically. While this language can represent a broad range of MPC problems, we restricted our discussion to QCQPs problems that are affine with respect to each on-line parameter. We showed how this general formulation of MPC can be expressed as a

parametric condensed QCQP in a form that can be efficiently computed on-line. The fact that a condensed formulation is automatically generated is a unique feature of *μAO-MPC*. This type of formulation is preferred in MPC when the horizon length is short, and in many embedded applications short horizon lengths are required. Furthermore, several popular general purpose and tailored solvers work with QCQP or QP formulations in condensed form, i.e. they cannot exploit sparsity on the problem matrices.

μAO-MPC takes a code generation approach that separates the steps of forming the QP for a given set of parameters, and solving the formed QP. To solve the formed QP, *μAO-MPC* by default implements a tailored ALM+FGM algorithm. In both steps (forming and solving), portable library-free C-code is generated that shows deterministic computation time, and has low requirements on ROM and RAM. Due to this strict separation of forming and solving a QP, the C-code generated by *μAO-MPC* can easily be used with other QP solvers, either for simulation purposes or for the final implementation. We have successfully combined *μAO-MPC* with qpOASES, CVXGEN, MATLAB's quadprog and CVXOPT.

As shown, the ALM+FGM algorithm works extremely well for applications that take advantage of warm start, where the Hessian is well conditioned, and where rough approximate solutions deliver good closed-loop performance. We presented several simulation and experimental applications that confirm this behavior. In particular, we used a low-cost microcontroller for a simple single-input system with two states. One of the key points of this example was the use of fixed-point arithmetics. On a more complex example, with two inputs and five states, we also exemplified how rough approximations deliver acceptable performance.

7.1. Outlook

The main contribution of this work was the implementation and experimental validation of *μAO-MPC*. In this regard, there are several theoretical properties that should be explored to complement and enhance this work. For example, although the rate of convergence and sub-optimality certifications of the fast gradient are well known, we have not developed them for the combined algorithm ALM+FGM.

Our current code generation approach is limited to Hessian matrices that are constant (i.e. do not depend on on-line parameters). In the case of linear time-varying systems,

or when the weighting matrices change with time, the Hessian is no longer constant. Considering theses case will expand the capabilities of *μAO-MPC*, however, this will also present new computational challenges.

Another extension to *μAO-MPC* would be the consideration of problems that are equivalent to second-order cone programs. This will require the implementation of a more complex optimization algorithm while still aiming at low-cost embedded systems. A primal interior point method seems like a good candidate. Although high accurate solutions may not be easily achieved with this type of algorithm, this is not a major concern for many applications as we have shown. Furthermore, a primal method can be easily warm started and the MPC structure can be as well exploited.

Bibliography

[1] I. Craig, C. Aldrich, R. Braatz, F. Cuzzola, E. Domlan, S. Engell, J. Hahn, V. Havlena, A. Horch, B. Huang *et al.*, "Control in the process industries," *The Impact of Control Technology. IEEE Control Systems Society, New York*, 2011.

[2] G. C. Goodwin and J. A. De Doná, *Constrained Control and Estimation: An Optimization Approach*. Springer, 2005.

[3] J. Rawlings and D. Mayne, *Model Predictive Control: Theory and Design*. Nob Hill Pub., 2009.

[4] L. Grüne and J. Pannek, *Nonlinear Model Predictive Control*. Springer, 2011.

[5] L. Wang, *Model Predictive Control System Design and Implementation Using MATLAB®*. Springer, 2009.

[6] R. Findeisen, F. Allgöwer, and L. T. Biegler, *Assessment and Future Directions of Nonlinear Model Predictive Control*. Springer, 2007, vol. 358.

[7] J. Maciejowski, *Predictive Control: With Constraints*. Pearson education, 2002.

[8] E. F. Camacho and C. B. Alba, *Model Predictive Control*. Springer, 2013.

[9] S. Qin and T. Badgwell, "A survey of industrial model predictive control technology," *Control Engineering Practice*, vol. 11, no. 7, pp. 733–764, 2003.

[10] T. Faulwasser, J. Matschek, P. Zometa, and R. Findeisen, "Predictive path-following control: Concept and implementation for an industrial robot," in *Int. Conf. Control Applications*. IEEE, 2013, pp. 128–133.

[11] F. Oldewurtel, A. Parisio, C. N. Jones, D. Gyalistras, M. Gwerder, V. Stauch, B. Lehmann, and M. Morari, "Use of model predictive control and weather forecasts for energy efficient building climate control," *Energy and Buildings*, vol. 45, pp. 15–27, 2012.

[12] R. R. Negenborn, P.-J. van Overloop, T. Keviczky, B. De Schutter *et al.*, "Distributed model predictive control of irrigation canals," *NHM*, vol. 4, no. 2, pp. 359–380, 2009.

[13] B. Grosman, E. Dassau, H. C. Zisser, L. Jovanovič, and F. J. Doyle, "Zone model predictive control: a strategy to minimize hyper-and hypoglycemic events," *Journal of Diabetes Science and Technology*, vol. 4, no. 4, pp. 961–975, 2010.

[14] G. Valencia-Palomo and J. Rossiter, "Efficient suboptimal parametric solutions to predictive control for plc applications," *Control Engineering Practice*, vol. 19, no. 7, pp. 732–743, 2011.

[15] S. Richter, S. Mariethoz, and M. Morari, "High-speed online MPC based on a fast gradient method applied to power converter control," in *American Control Conference*, 2010, pp. 4737–4743.

[16] P. D. Vouzis, M. V. Kothare, L. G. Bleris, and M. G. Arnold, "A system-on-a-chip implementation for embedded real-time model predictive control," *Control Systems Technology, IEEE Transactions on*, vol. 17, no. 5, pp. 1006–1017, 2009.

[17] K.-V. Ling, B. F. Wu, and J. Maciejowski, "Embedded model predictive control (MPC) using a FPGA," in *Proc. 17th IFAC World Congress*, 2008, pp. 15 250–15 255.

[18] D. Bao-Cang, *Modern Predictive Control*. CRC press, 2010.

[19] D. Mayne, J. Rawlings, C. Rao, and P. Scokaert, "Constrained model predictive control: Stability and optimality," *Automatica*, vol. 36, pp. 789–814, 2000.

[20] C. Rao, S. Wright, and J. Rawlings, "Application of interior-methods to model predictive control," *Journal of Optimization Theory and Applications*, vol. 99, pp. 723–757, 1998.

[21] S. Lucia, P. Zometa, M. Kögel, and R. Findeisen, "Efficient stochastic model predictive control based on polynomial chaos expansions for embedded applications," in *Proc. Conf. Decision and Control*, 2015, pp. 322–331.

[22] A. Domahidi, A. Zgraggen, M. Zeilinger, M. Morari, and C. Jones, "Efficient interior point methods for multistage problems arising in receding horizon control," in *Proc. Conf. Decision and Control*, 2012, pp. 668 – 674.

[23] M. Kögel and R. Findeisen, "On efficient predictive control of linear systems subject to quadratic constraints using condensed, structure-exploiting interior point methods," in *Proc. European Control Conf.*, 2013, pp. 27–34.

[24] E. Chu, N. Parikh, A. Domahidi, and S. Boyd, "Code generation for embedded second-order cone programming," in *Proc. European Control Conf.* IEEE, 2013, pp. 1547–1552.

[25] P. Zometa, M. Kögel, and R. Findeisen, "muAO-MPC: A free code generation tool for embedded real-time linear model predictive control," in *Proc. American Control Conf.*, 2013, pp. 5340–5345.

[26] A. Bemporad, M. Morari, V. Dua, and E. Pistikopoulos, "The explicit linear quadratic regulator for constrained systems," *Automatica*, vol. 38, no. 1, pp. 3–20, 2002.

[27] H. J. Ferreau, H. G. Bock, and M. Diehl, "An online active set strategy to overcome the limitations of explicit MPC," *International Journal of Robust and Nonlinear Control*, vol. 18, pp. 816–830, 2008.

[28] D. Bertsekas, *Nonlinear Programming*. Athena Scientific Belmont, MA, 1999, pp. 149–151.

[29] S. Wright and J. Nocedal, *Numerical Optimization*. Springer New York, 1999, vol. 2.

[30] S. J. Wright, *Primal-dual Interior-point Methods*. Siam, 1997.

[31] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[32] 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, 2010.

[33] H. Ferreau, "qpOASES–an open-source implementation of the online active set strategy for fast model predictive control," in *Proc. of the Workshop on Nonlinear Model Based Control–Software and Applications, Loughborough*, 2007, pp. 29–30.

[34] L. K. McGovern, "Computational Analysis of Real-time Convex Optimization for Control Systems," Ph.D. dissertation, Massachusetts Institute of Technology, 2000.

[35] H. J. Ferreau, "Model Predictive Control Algorithms for Applications with Millisecond Timescales," Ph.D. dissertation, PhD thesis, KU Leuven, 2011.

[36] S. Richter, C. N. Jones, and M. Morari, "Computational complexity certification for real-time MPC with input constraints based on the fast gradient method," *Automatic Control, IEEE Transactions on*, vol. 57, no. 6, pp. 1391–1403, 2012.

[37] V. Nedelcu and I. Necoara, "Iteration complexity of an inexact augmented Lagrangian method for constrained MPC," in *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*. IEEE, 2012, pp. 650–655.

[38] J. L. Jerez, P. J. Goulart, S. Richter, G. A. Constantinides, E. C. Kerrigan, and M. Morari, "Embedded online optimization for model predictive control at megahertz rates."

[39] B. W. Kernighan, D. M. Ritchie, and P. Ejeklint, *The C Programming Language*. Prentice-Hall Englewood Cliffs, 1988, vol. 2.

[40] GNU-GCC, "The GNU Compiler Collection," https://gcc.gnu.org/.

[41] J. Lofberg, "YALMIP: A toolbox for modeling and optimization in MATLAB," in *Computer Aided Control Systems Design, 2004 IEEE International Symposium on*. IEEE, 2004, pp. 284–289.

[42] R. Fourer, D. Gay, and B. Kernighan, *AMPL*. Boyd & Fraser, 1993, vol. 119.

[43] M. Grant and S. Boyd, "CVX: Matlab software for disciplined convex programming, version 2.1."

[44] J. Mattingley and S. Boyd, "CVXGEN: A code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, no. 1, pp. 1–27, 2012.

[45] J. Mattingley, Y. Wang, and S. Boyd, "Code generation for receding horizon control," in *Proc. of the IEEE International Symposium on Computer-Aided Control System Design, Yokohama, Japan*, 2010.

[46] A. Domahidi, E. Chu, and S. Boyd, "ECOS: An SOCP solver for embedded systems," in *Proc. European Control Conf.*, 2013, pp. 3071–3076.

[47] C. Jones, A. Domahidi, M. Morari, S. Richter, F. Ullmann, and M. Zeilinger, "Fast predictive control: real-time computation and certification," in *4th IFAC Nonlinear Predictive Control Conference*, 2012, pp. 94–98.

[48] F. Ullmann, "FiOrdOs: A Matlab Toolbox for C-Code Generation for First Order Methods," Master's thesis, ETH Zurich, 2011.

[49] B. Houska, H. J. Ferreau, and M. Diehl, "ACADO toolkit—An open-source framework for automatic control and dynamic optimization," *Optimal Control Applications and Methods*, vol. 32, no. 3, pp. 298–312, 2011.

[50] B. Houska, H. Ferreau, and M. Diehl, "An auto-generated real-time iteration algorithm for nonlinear MPC in the microsecond range," *Automatica*, 2011.

[51] E. C. Kerrigan and J. M. Maciejowski, "Feedback min-max model predictive control using a single linear program: Robust stability and the explicit solution," *International Journal of Robust and Nonlinear Control*, vol. 14, no. 4, pp. 395–413, 2004.

[52] A. Bemporad and M. Morari, "Robust model predictive control: A survey," in *Robustness in Identification and Control*. Springer, 1999, pp. 207–226.

[53] D. Van Hessem and O. Bosgra, "A conic reformulation of model predictive control including bounded and stochastic disturbances under state and input constraints," in *Decision and Control, 2002, Proc. of the 41st IEEE Conference on*, vol. 4. IEEE, 2002, pp. 4643–4648.

[54] W. S. Levine, *The Control Handbook*. CRC press, 1996.

[55] G. F. Franklin, M. L. Workman, and D. Powell, *Digital Control of Dynamic Systems*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[56] B. D. Anderson and J. B. Moore, *Optimal Control: Linear Quadratic Methods*. Courier Dover Publications, 2007.

[57] H. Kwakernaak and R. Sivan, *Linear Optimal Control Systems*. Wiley-interscience New York, 1972, vol. 1.

[58] E. C. Kerrigan, J. L. Jerez, S. Longo, and G. A. Constantinides, "Number representation in predictive control," in *Proc. IFAC Conf. Nonlinear Model Predictive Control, Noordwijkerhout, NL*. Citeseer, 2012, pp. 60–67.

[59] G. C. Goodwin, J. I. Yuz, J. Aguero, and M. Cea, "Sampling and sampled-data models," in *American Control Conference (ACC), 2010*. IEEE, 2010, pp. 1–20.

[60] H. Kopetz, *Real-time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2011.

[61] P. Marwedel, *Embedded System Design: Embedded Systems Foundations of Cyber-physical Systems*. Springer Science & Business Media, 2010.

[62] R. Baheti and H. Gill, "Cyber-physical systems," *The Impact of Control Technology*, pp. 161–166, 2011.

[63] J. Yiu, *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*. Newnes, 2013.

[64] P. Zometa, M. Kögel, T. Faulwasser, and R. Findeisen, "Implementation aspects of model predictive control for embedded systems," in *Proc. American Control Conf.*, 2012, pp. 1205–1210.

[65] P. A. Laplante, "Real-time systems design and analysis," 1993.

[66] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[67] A. Chatzigeorgiou and G. Stephanides, *Evaluating Performance and Power of Object-oriented vs. Procedural Programming in Embedded Processors.* Springer, 2002.

[68] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management.* Springer, 1995, pp. 1–116.

[69] I. Puaut, "Real-time performance of dynamic memory allocation algorithms," in *Real-Time Systems, 2002. Proc.. 14th Euromicro Conference on.* IEEE, 2002, pp. 41–49.

[70] M. MISRA *et al.*, *MISRA-C: 2004 Guidelines for The Use of The C Language in Critical Systems.*

[71] M. N. Zeilinger, C. N. Jones, D. M. Raimondo, and M. Morari, "Real-time MPC-Stability through robust MPC design," in *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proc. of the 48th IEEE Conference on.* IEEE, 2009, pp. 3980–3986.

[72] M. Cannon, B. Kouvaritakis, and J. A. Rossiter, "Efficient active set optimization in triple mode MPC," *Automatic Control, IEEE Transactions on*, vol. 46, no. 8, pp. 1307–1312, 2001.

[73] M. S. Lobo, L. Vandenberghe, S. Boyd, and H. Lebret, "Applications of second-order cone programming," *Linear algebra and its applications*, vol. 284, no. 1, pp. 193–228, 1998.

[74] P. Zometa, H. Heinemann, S. Lucia, M. Kögel, and R. Findeisen, "Efficient stochastic model predictive control for embedded systems based on second-order cone programs," in *Proc. European Control Conf.*, 2016, pp. 166–171.

[75] B. Kouvaritakis, M. Cannon, and V. Tsachouridis, "Recent developments in stochastic MPC and sustainable development," *Annual Reviews in Control*, vol. 28, no. 1, pp. 23–35, 2004.

[76] CVXOPT, "CVXOPT: Homepage," http://cvxopt.org/.

[77] OpenOpt.

[78] N. I. Gould and P. L. Toint, "A quadratic programming bibliography," *Numerical Analysis Group Internal Report*, vol. 1, 2000.

[79] U. Maeder and M. Morari, "Offset-free reference tracking with model predictive control," *Automatica*, vol. 46, no. 9, pp. 1469–1476, 2010.

[80] D. Limón, I. Alvarado, T. Alamo, and E. F. Camacho, "MPC for tracking piecewise constant references for constrained linear systems," *Automatica*, vol. 44, no. 9, pp. 2382–2387, 2008.

[81] S. Olaru and S.-I. Niculescu, "Predictive control for linear systems with delayed input subject to constraints," in *Proc. IFAC World Congress*, 2008, pp. 11 208–11 213.

[82] T. G. Hovgaard, K. Edlund, and J. Bagterp Jorgensen, "The potential of economic MPC for power management," in *Proc. Conf. Decision and Control*. IEEE, 2010, pp. 7533–7538.

[83] M. Kögel, P. Zometa, and R. Findeisen, "On tailored model predictive control for low cost embedded systems with memory and computational power constraints," in *Technical report.*, 2012.

[84] A. Mesbah, S. Streif, R. Findeisen, and R. D. Braatz, "Stochastic nonlinear model predictive control with probabilistic constraints," in *Proc. American Control Conf.* IEEE, 2014, pp. 2413–2419.

[85] P. O. Scokaert and J. B. Rawlings, "Feasibility issues in linear model predictive control," *AIChE Journal*, vol. 45, no. 8, pp. 1649–1659, 1999.

[86] E. C. Kerrigan and J. M. Maciejowski, "Soft constraints and exact penalty functions in model predictive control," in *Control 2000 Conference, Cambridge*, 2000.

[87] N. Haverbeke, M. Diehl, and B. De Moor, "A structure exploiting interior-point method for moving horizon estimation," in *Proc. Conf. Decision and Control, Chinese Control Conf.*, 2009, pp. 1273–1278.

[88] S. Mehrotra, "On the implementation of a primal-dual interior point method," *SIAM Journal on optimization*, vol. 2, no. 4, pp. 575–601, 1992.

[89] D. Dueri, J. Zhang, and B. Açikmese, "Automated custom code generation for embedded, real-time second order cone programming," in *IFAC World Congress*, 2014, pp. 1605–1612.

[90] J. Mattingley, Y. Wang, and S. Boyd, "Receding horizon control: automatic generation of high-speed solvers," *IEEE Control Systems Magazine*, vol. 31, no. 3, pp. 52–65, 2011.

[91] E. A. Yildirim and S. J. Wright, "Warm-start strategies in interior-point methods for linear programming," *SIAM Journal on Optimization*, vol. 12, no. 3, pp. 782–810, 2002.

[92] A. Shahzad, E. C. Kerrigan, and G. A. Constantinides, "A warm-start interior-point method for predictive control," 2010.

[93] A. Alessio and A. Bemporad, "A survey on explicit model predictive control," in *Nonlinear Model Predictive Control*. Springer, 2009, pp. 345–369.

[94] M. Mönnigmann and M. Kastsian, "Fast explicit MPC with multiway trees," in *Proc. of the 18th IFAC World Congress*, 2011.

[95] M. Kögel and R. Findeisen, "Fast predictive control of linear, time-invariant systems using an algorithm based on the fast gradient method and augmented Lagrange multipliers," in *Control Applications (CCA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 780–785.

[96] Y. Nesterov, "A method of solving a convex programming problem with convergence rate O (1/k2)," in *Soviet Mathematics Doklady*, vol. 27, no. 2, 1983, pp. 372–376.

[97] ——, *Introductory Lectures on Convex Optimization: A Basic Course*. Kluwer Academic Publishers, 2004.

[98] D. P. Bertsekas, *Constrained Optimization and Lagrange Multiplier Methods*. Athena Scientific, 1996.

[99] S. Richter, C. Jones, and M. Morari, "Real-time input-constrained MPC using fast gradient methods," in *Proc. of the 48th IEEE Conference on Decision and Control and the 28th Chinese Control Conference*, 2009, pp. 7387–7393.

[100] H. Waschl, D. Alberer, and L. del Re, "Automatic tuning methods for MPC environments," *Computer Aided Systems Theory–Eurocast 2011*, vol. 6928, pp. 41–48, 2012.

[101] ——, "Numerically efficient self tuning strategies for MPC of integral gas engines," in *Proc. of the 18th IFAC World Congress*, 2011, pp. 2482–2487.

[102] P. Zometa, M. Kögel, and R. Findeisen, "muAO-MPC: Homepage," http://ifatwww.et.uni-magdeburg.de/syst/muAO-MPC/, 2013.

[103] S. Lucia, D. Navarro, O. Lucia, P. Zometa, and R. Findeisen, "Optimized FPGA Implementation of Model Predictive Control for Embedded Cyber-physical Systems Using High Level Synthesis Tools," *IEEE Transactions on Industrial Informatics (submitted)*, 2016.

[104] Chibios, "Chibios/RT Homepage," www.chibios.org.

[105] gnu-arm, "GNU tools for ARM embedded processors," https://launchpad.net/gcc-arm-embedded.

[106] R. J. Simpson, R. Palacios, H. Hesse, and P. Goulart, "Predictive control for alleviation of gust loads on very flexible aircraft," in *55th AIAA Structures, Structural Dynamics, and Materials Conference*, 2014, pp. 2014–0843.

[107] G. Takacs, P. Zometa, R. Findeisen, and B. Rohal'-Ilkiv, "Efficiency and performance of embedded model predictive control for active vibration attenuation," in *Proc. European Control Conf.*, 2016, pp. 1334–1340.

[108] ——, "Embedded model predictive vibration control on low-end 8-bit microcontrollers via automatic code generation," in *Int. Congress on Sound & Vibration*, 2016, pp. 1–6.

[109] D. Jaggar *et al.*, "ARM architecture and systems," *IEEE micro*, vol. 17, no. 4, pp. 9–11, 1997.

[110] Takashi Chikamasa, "nxtOSEK/JSP ANSI C/C++ with OSEK/muITRON RTOS for LEGO MINDSTORMS NXT," http://lejos-osek.sourceforge.net/, 2007.

[111] G. Bishop and G. Welch, "An introduction to the Kalman filter," *Proc of SIGGRAPH, Course*, vol. 8, pp. 27 599–3175, 2001.

A. Forming a condensed parametric quadratic program

In the following, we will make reference to the considered MPC setup (2.15). For convenience, we will repeat here some of the relevant information.

Note that MPC problem is subject to the constraints: $x_0 = x$, and $x_{k+1} = Ax_k + Bu_k$, from which we know that $x_1 = Ax_0 + Bu_0$, and $x_2 = Ax_1 + Bu_1 = A(Ax_0 + Bu_0) + Bu_1 = A^2Bx_0 + ABu_0 + Bu_1$, and more generally

$$x_j = A^j x_0 + A^{j-1}Bu_0 + ... + Bu_{j-1} \tag{A.1}$$

hold. Recall the sequences $\mathbf{u} = \{u_0, u_1, ..., u_{N-1}\}$ and $\mathbf{x}(x, \mathbf{u}) = \{x_0, x_1, x_2, ..., x_N\}$. Based on the latter, we define the sub-sequence $\mathbf{x}_{1:N}(x_0, \mathbf{u}) = \{x_1, x_2, ..., x_N\}$. For simplicity, we will in the following omit the dependency of $\mathbf{x}$ and $\mathbf{x}_{1:N}$ on its parameters.

We consider first the special case of regulation to the origin with only input constraints. In the case of regulation of the origin, the model predictive control (MPC) cost function is given by

$$V(\mathbf{x}, \mathbf{u}) = \frac{1}{2}\sum_{i=0}^{N-1}(\|x_i\|_Q^2 + \|u_i\|_R^2) + \frac{1}{2}\|x_N\|_P^2. \tag{A.2}$$

It can be equivalently written as

$$V(\mathbf{x}, \mathbf{u}) \;=\; \tfrac{1}{2}(\|x_0\|_Q^2 + \|\mathbf{x}_{1:N}\|_{\mathbf{Q}}^2 + \|\mathbf{u}\|_{\mathbf{R}}^2), \tag{A.3}$$

where

$$\mathbf{Q} = \begin{bmatrix} Q & 0 & \cdots & 0 \\ 0 & Q & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & P \end{bmatrix}, \; \mathbf{R} = \begin{bmatrix} R & 0 & \cdots & 0 \\ 0 & R & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & R \end{bmatrix}.$$

Furthermore, using (A.1) we can write $\mathbf{x}_{1:N}(x_0, \mathbf{u}) = \mathbf{A}x_0 + \mathbf{B}\mathbf{u}$ where:

$$\mathbf{A} = \begin{bmatrix} A \\ A^2 \\ \vdots \\ A^N \end{bmatrix}, \ \mathbf{B} = \begin{bmatrix} B & 0 & \cdots & 0 \\ AB & B & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ A^{N-1}B & A^{N-2}B & \cdots & B \end{bmatrix}.$$

Recall that the $\|y\|_M^2 = y^\intercal My$. We can then equivalently express (A.3) as

$$V(x_0, \mathbf{u}) = \frac{1}{2}(x_0^\intercal Q x_0 + (\mathbf{A}x_0 + \mathbf{B}\mathbf{u})^\intercal \mathbf{Q}(\mathbf{A}x_0 + \mathbf{B}\mathbf{u}) + \mathbf{u}^\intercal \mathbf{R}\mathbf{u}).$$

After expanding and rearranging some terms it can be written as

$$\begin{aligned} V(x_0, \mathbf{u}) &= \tfrac{1}{2}(\mathbf{B}\mathbf{u})^\intercal \mathbf{Q}\mathbf{B}\mathbf{u} + \mathbf{u}^\intercal \mathbf{R}\mathbf{u} + 2(\mathbf{B}\mathbf{u})^\intercal \mathbf{Q}\mathbf{A}x_0 + (\mathbf{A}x_0)^\intercal \mathbf{Q}\mathbf{A}x_0 + x_0^\intercal Q x_0) \\ &= \tfrac{1}{2}(\mathbf{u}^\intercal(\mathbf{B}^\intercal \mathbf{Q}\mathbf{B} + \mathbf{R})\mathbf{u} + \mathbf{u}^\intercal(2\mathbf{B}^\intercal \mathbf{Q}\mathbf{A}x_0) + (\mathbf{A}x_0)^\intercal \mathbf{Q}\mathbf{A}x_0 + x_0^\intercal Q x_0) \end{aligned}$$

Finally, it can be concisely expressed as

$$V(x_0, \mathbf{u}) = \frac{1}{2}\mathbf{u}^\intercal H \mathbf{u} + g(x_0)^\intercal \mathbf{u} + c(x_0) \tag{A.4}$$

with $H = \mathbf{B}^\intercal \mathbf{Q}\mathbf{B} + \mathbf{R}$, $g(x_0) = Gx_0$, $G = \mathbf{B}^\intercal \mathbf{Q}\mathbf{A}$, and $c(x_0) = \frac{1}{2}((\mathbf{A}x_0)^\intercal \mathbf{Q}\mathbf{A}x_0 + x_0^\intercal Q x_0)$.

The input constraints set $\mathcal{U} = U^N$ can be equivalently defined as

$$\mathcal{U} = \{\mathbf{u} \in \mathbb{R}^{Nm} \mid \underline{\mathbf{u}} \leq \mathbf{u} \leq \overline{\mathbf{u}}\}, \ \ \underline{\mathbf{u}} = \begin{bmatrix} \underline{u} \\ \vdots \\ \underline{u} \end{bmatrix}, \ \overline{\mathbf{u}} = \begin{bmatrix} \overline{u} \\ \vdots \\ \overline{u} \end{bmatrix}. \tag{A.5}$$

Finally, with the cost (A.4), the constraints (A.5) and $x_0 = x$, the parametric quadratic program $\mathbb{P}_I(x)$ has the form

$$\begin{aligned} \underset{\mathbf{u}}{\text{minimize}} \ \ & \frac{1}{2}\mathbf{u}^\intercal H \mathbf{u} + g(x_0)^\intercal \mathbf{u} \\ \text{subject to} \ \ & \underline{\mathbf{u}} \leq \mathbf{u} \leq \overline{\mathbf{u}} \end{aligned} \tag{A.6}$$

Note that we do not include the term $c(x_0)$ from (A.4) because it does not influence the solution.

Following a similar approach, the considered mixed constrained in (2.15) can be con-

cisely written as $\underline{\mathbf{z}}(x_0) \leq E\mathbf{u} \leq \overline{\mathbf{z}}(x_0)$, where:

$$\underline{\mathbf{z}}(x_0) = \begin{bmatrix} \underline{e} - K_x x_0 \\ \vdots \\ \underline{e} - K_x A^{N-1} x_0 \\ \underline{f} - F A^N x_0 \end{bmatrix}, \; \overline{\mathbf{z}}(x_0) = \begin{bmatrix} \overline{e} - K_x x_0 \\ \vdots \\ \overline{e} - K_x A^{N-1} x_0 \\ \overline{f} - F A^N x_0 \end{bmatrix},$$

and

$$E = \begin{bmatrix} K_u & 0 & \cdots & 0 \\ K_x B & K_u & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ K_x A^{N-2} B & K_x A^{N-3} B & \cdots & K_u \\ F A^{N-1} B & F A^{N-2} B & \cdots & F B \end{bmatrix}.$$

We can then extend (A.6) to the case with mixed constraints. The parametric quadratic program $\mathbb{P}_{II}(x)$ has the form

$$\begin{aligned} \underset{\mathbf{u}}{\text{minimize}} \;\; & \frac{1}{2}\mathbf{u}^\intercal H \mathbf{u} + g(x_0)^\intercal \mathbf{u} \\ \text{subject to} \;\; & \underline{\mathbf{u}} \leq \mathbf{u} \leq \overline{\mathbf{u}} \\ & \underline{\mathbf{z}}(x) \leq E\mathbf{u} \leq \overline{\mathbf{z}}(x) \end{aligned} \tag{A.7}$$

B. System Matrices

B.1. Simulation examples

The continuous-time robot arm system is given by

$$A_c = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -17.2 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -16.1 \end{bmatrix}, B_c = \begin{bmatrix} 0 & 0 \\ 2.62 & 0 \\ 0 & 0 \\ 0 & 2.48 \end{bmatrix},$$

$$x_c^\mathsf{T} = \begin{bmatrix} \theta_1 & \omega_1 & \theta_2 & \omega_2 \end{bmatrix}, u_c^\mathsf{T} = \begin{bmatrix} u_1 & u_2 \end{bmatrix},$$

where θ, ω, u denote respectively the angular position [rad], angular speed [rad/s], and input for each link [%]. The subindices 1, 2 denote the link. The weighting matrices are

$$Q = \mathrm{diag}(\begin{bmatrix} 1.14E4 & 2.24E1 & 1.14E4 & 2.94E1 \end{bmatrix}),$$

$$R = \mathrm{diag}(\begin{bmatrix} 2.20E{-}1 & 2.37E{-}1 \end{bmatrix}), P = Q.$$

The robot arm is subject to the following constraints: $-100 \leq u_1 \leq 100$, $-25 \leq u_2 \leq 25$, $-1 \leq \omega_i \leq 1, i = 1, 2$.

The discrete-time aircraft system is given by

$$A_d = \begin{bmatrix} 0.239 & 0 & 0.178 & 0 & 0 \\ -0.372 & 1 & 0.270 & 0 & 0 \\ -0.990 & 0 & 0.139 & 0 & 0 \\ -48.9 & 64.1 & 2.40 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, B_d = \begin{bmatrix} -1.23 \\ -1.44 \\ -4.48 \\ -1.80 \\ 1 \end{bmatrix},$$

$$x^\mathsf{T} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix},$$

where the x_1 is the angle of attack in [rad], x_2 is the pitch angle [rad], x_3 is the derivative of x_2 in [rad/s], x_4 is the altitute in [m], x_5 was added to consider the slew rate constraint in the input [rad/s]. The altitute rate is given by $\dot{x}_4 = -128.2x_1 + 128.2x_2$ in [m/s]. The

input u is the elevator angle in [rad]. The weighting matrices are

$$Q = \mathrm{diag}([\,1.22E3\ 3.12E3\ 6.12E3\ 5.08E{-}1\ 3.46E2\,]),$$

$$R = [\,3.46E2\,]\,, P = Q.$$

The aircraft is subject to the following constraints: $-0.262 \leq u \leq 0.262$, $-0.524 \leq \dot{u} \leq 0.524$, $-0.349 \leq x_2 \leq 0.349$, and $-30 \leq \dot{x}_4 \leq 30$.

B.2. Application examples

The following matrices and parameters were used by the MPC controller: penalty parameter $\mu = 8192$, diagonal weighting matrices $Q = \mathtt{diag}(10^{-2}, 1, 10^{-2}, 10^{-2}, 2 \cdot 10^{-1})$, $R = \mathtt{diag}(10^{-1}, 10^{-1})$, P is the solution to the discrete algebraic Riccati equation, the contraints matrix $K_x = [0, 0, 0, 1, 0]$ and the corresponding bounds $e_{lb} = -0.1$, $e_{ub} = 0.1$ have been scaled by the factor $c_E = 6.5 \cdot 10^{-1}$ to reduce the numerical condition of the internal problem. The LQR have been tuned with the following weighting matrices $Q_f = \mathtt{diag}(10^{-2}, 1)$, $R_f = [10^{-3}]$, $Q_l = \mathtt{diag}(10^{-2}, 10^{1}, 2 \cdot 10^{-1})$, $R_l = [10^{-1}]$.