Master Thesis

Complexity of first-order methods for fast embedded model predictive control

Author(s):
Kouzoupis, Dimitris

Publication Date:
2014

Permanent Link:
https://doi.org/10.3929/ethz-a-010255501

Rights / License:
In Copyright - Non-Commercial Use Permitted
Complexity of First-Order Methods for Fast Embedded Model Predictive Control

Master Thesis

Dimitris Kouzoupis

Supervision

Dr. Joachim Ferreau
Dr. Helfried Peyrl
Dr. Paul Goulart
Prof. John Lygeros

August 2014
Abstract

Fast and efficient numerical methods for solving Quadratic Programming problems (QPs) in the area of Model Predictive Control (MPC) exist nowadays in so many variations that choosing the right solver for a specific application is no longer a trivial procedure. Although active-set and interior-point methods have monopolized the interest of control engineers over the past few decades, research in first-order methods has been recently revived and several algorithms based on Yurii Nesterov’s fast gradient method (FGM) [34] have emerged. Some of the properties that contributed to this development is the simplicity of the algorithmic scheme, the guaranteed convergence rates and the ability to compute small to medium accuracy solutions in high sampling rates [23].

Following the growing interest in the field, the purpose of this Thesis is to provide a detailed overview of first-order solvers under a unifying framework and analyze their computational complexity in nonlinear MPC applications, where the generated QPs may comprise time-varying data. The results of the complexity analysis are then used as a guide to design an efficient MPC controller for an industrial example of ABB Corporate Research.
Nomenclature

Symbols

\( A^T \) Transpose of matrix \( A \)
\( \mathbb{R}^n \) Set of real column vectors of length \( n \)
\( \mathbb{R}_+^n \) Set of real positive column vectors of length \( n \)
\( \mathbb{R}^{m \times n} \) Set of real \( m \) by \( n \) matrices
\( \mathbb{S}^n \) Set of symmetric \( n \) by \( n \) matrices
\( \mathbb{S}_+^n \) Set of symmetric positive semidefinite matrices
\( \mathbb{S}_{++}^n \) Set of symmetric positive definite matrices
\( \mathcal{I}_\mathcal{X} \) Indicator function of set \( \mathcal{X} \), as Equation (3.3)
\( \Delta_n(a,b) \) Distance metric between two vectors, as defined in Equation (5.2)
\( [\cdot]_\mathcal{X} \) Projection on set \( \mathcal{X} \)
\( \sigma_{\text{max}}(A) \) Maximum singular value of matrix \( A \)
\( \lambda_{\text{min}}(A) \) Minimum eigenvalue of matrix \( A \in \mathbb{S}^n \)
\( \lambda_{\text{max}}(A) \) Maximum eigenvalue of matrix \( A \in \mathbb{S}^n \)

Indicies

\( u \) Upper bound
\( l \) Lower bound
\( \min \) Minimum
\( \max \) Maximum

Acronyms and Abbreviations

QP Quadratic Programming (problem)
SDP Semidefinite Programming (problem)
MPC Model Predictive Control
RHC Receding Horizon Control
KKT Karush-Kuhn-Tucker
LTI Linear Time-Invariant
LTV Linear Time-Variant
FGM Fast Gradient Method
FPGA Field-Programmable Gate Array
Contents

Abstract v
Nomenclature vii
List of Figures xiii
List of Tables xv

1 Introduction 1
  1.1 Motivation .............................................. 1
  1.2 Contributions ........................................... 1
  1.3 Outline .................................................. 2

2 Quadratic Programming in Model Predictive Control 3
  2.1 Survey of existing methods ................................ 4
  2.2 The MPC problem .......................................... 7
  2.3 Condensed formulation ..................................... 8
  2.4 Sparse formulations ....................................... 8

3 Overview of First-Order Methods 11
  3.1 Primal Fast Gradient Method ............................... 11
  3.1.1 Background ............................................. 11
  3.1.2 The algorithm ........................................... 12
  3.1.3 Generalized variant ..................................... 13
  3.2 Richter’s Dual Fast Gradient Method ....................... 14
  3.2.1 Problem class .......................................... 14
  3.2.2 The algorithm ........................................... 15
  3.2.3 Lipschitz constant ...................................... 16
  3.2.4 Generalized variant ..................................... 16
  3.2.5 Implementation of soft constraints ....................... 17
  3.3 Bemporad’s Dual Fast Gradient Method ..................... 20
  3.3.1 The algorithm ........................................... 21
  3.3.2 Lipschitz constant ...................................... 22
  3.3.3 Generalized variant ..................................... 25
  3.4 Alternating Direction Method of Multipliers ............... 25
  3.4.1 Precursors of ADMM ..................................... 25
  3.4.2 The algorithm in sparse formulation ..................... 27
  3.4.3 The algorithm in condensed formulation .................. 30
# Complexity of Fast Gradient Methods

4.1 Measuring program efficiency ........................................ 31
4.2 Elementary operations .............................................. 32
4.3 Preliminaries and notation ........................................ 35
4.4 Structure of Fast Gradient Methods ................................ 36
4.5 Complexity of problem transformation ............................ 38
   4.5.1 xuy formulation ............................................. 38
   4.5.2 xu formulation .............................................. 38
   4.5.3 uy formulation .............................................. 39
   4.5.4 u formulation .............................................. 40
   4.5.5 Conclusions ................................................ 42
4.6 Complexity of Primal FGM .......................................... 42
   4.6.1 u formulation .............................................. 43
4.7 Complexity of Richter’s Dual FGM ................................ 44
   4.7.1 xuy formulation ............................................. 45
   4.7.2 xu formulation .............................................. 49
   4.7.3 uy formulation .............................................. 50
   4.7.4 Conclusions ................................................ 52
4.8 Complexity of Bemporad’s Dual FGM ............................... 53
   4.8.1 xu formulation .............................................. 56
   4.8.2 u formulation .............................................. 59
   4.8.3 Lipschitz constants and generalized matrices ............... 60
   4.8.4 Conclusions ................................................ 64
4.9 Summary table .................................................... 67

# Benchmarking of First-Order Solvers

5.1 A general MPC benchmarking suite ................................ 69
5.2 Ill-conditioning .................................................... 73
5.3 Polyhedral inequality constraints .................................. 74
5.4 Different sparse formulations of Richter’s Dual FGM .......... 75

# Nonlinear Case Study

6.1 Problem description ................................................ 79
6.2 Suitable algorithms and formulations ......................... 80
6.3 Complexity of favorable combinations ............................ 81
   6.3.1 Richter’s Dual FGM ........................................ 81
   6.3.2 Bemporad’s Dual FGM ...................................... 83
   6.3.3 Comparison of complexities ................................ 85
6.4 Simulations ....................................................... 85
6.5 Interpretation of results .......................................... 88

# Conclusion

7.1 Summary .......................................................... 91
7.2 Future work ....................................................... 91

# Problem Formulations

A.1 xuy formulation .................................................. 93
   A.1.1 Objective function ......................................... 93
   A.1.2 Constraints ................................................ 94
List of Figures

4.1 Illustration of matrices for an example with 4 states, 2 inputs, 2 outputs and $N = 10$. .................................................. 48
4.2 Scaling of complexity when number of states is doubled. ................. 53
4.3 Scaling of complexity when number of inputs is doubled. .................. 54
4.4 Scaling of complexity when number of outputs is doubled. .................. 54
4.5 Total complexity for an example with 6 states, 2 inputs and 4 outputs. ... 55
4.6 Total complexity from a different view. ........................................ 55
4.7 Scaling of complexity when number of states is doubled. .................. 65
4.8 Scaling of complexity when number of inputs is doubled. .................. 65
4.9 Scaling of complexity when number of outputs is doubled. .................. 66

5.1 KKT tolerances, number of iterations, execution time and number of active constraints for nonlinear string benchmark problem. ................. 72
5.2 Performance profile of first-order solvers [27]. .................................. 73
5.3 Number of iterations for problem with ill-conditioned Hessian and unstable model. ......................................................... 75
5.4 Effect of polyhedral constraints on convergence of generalized Algorithm 2. 76
5.5 Theoretical complexity for quadcopter benchmark example. ............... 78

6.1 Complexity of problem transformation/initialization (left) and one iteration (right) as function of the horizon length. ......................... 86
6.2 Closed-loop simulation of nonlinear model. ..................................... 87
6.3 Number of iterations for convergence to $U^*$. ................................ 87
6.4 Number of flops for Algorithm 2. .................................................. 88
6.5 Number of flops for Algorithm 3. .................................................. 89
List of Tables

4.1 Complexity of elementary operations ........................................ 35
4.2 Structure of FGM and factors that affect each step. ....................... 37
4.3 Complexity summary of problem transformation ............................. 43
4.4 Problem transformation and initialization complexity of Algorithm 2. . . 52
4.5 Complexity per iteration of Algorithm 2. .................................... 52
4.6 Permissible $L$ depending on time-varying nature of problem data. ....... 62
4.7 Problem transformation and initialization complexity of Algorithm 3. .. 64
4.8 Complexity per iteration of Algorithm 3. .................................... 65
4.9 Initialization complexity of FGMs for different combinations of time-varying data. ................................................................. 67

5.1 Maximum number of iterations in closed-loop for different $\delta$. ........... 77

6.1 Theoretical complexity of the discussed methods measured in flops. ....... 85
6.2 Minimum, maximum and average number of iterations of first-order solvers. 88
Chapter 1

Introduction

As an introduction, we intend to motivate the project, highlight its theoretical and practical contributions and briefly describe its structure.

1.1 Motivation

Although the fast gradient method (FGM) was discovered by Yurii Nesterov already in 1983 [34], it had until recently no impact on the control community, where interior-point and active-set methods monopolized the interest of the researchers. However, embedded Model Predictive Control (MPC) applications with controllers that can operate at megahertz rates and on cheap hardware created the need for light Quadratic Programming (QP) solvers that are able to return low to medium accuracy solutions in very short time intervals. This evolution led to the revival of gradient methods and their use in control applications, with one of the major contributors being Stefan Richter and his PhD Thesis on ”Computational Complexity Certification of Gradient Methods for Real-Time Model Predictive Control” [38]. His work had a great impact on the control community where numerous inspiring papers made gradient methods an active field of research [19,36,39]. This activity recently brought several variants of Nesterov’s fast gradient method into existence in such a short period of time that it is currently not clear for a control engineer which method is more suitable in each context. The limited benchmarking information and the inconsistent notation between papers motivated us to establish a common framework for all the newly proposed methods and assess their strong and weak points via a proper assessment environment [27]. Finally and more importantly, the potential of fast gradient methods in nonlinear MPC applications convinced us that a detailed complexity analysis of the algorithms when part of the QP data are time-varying could further extend their applicability and help control engineers to make easier design choices.

1.2 Contributions

Due to the wide scope of the Thesis, several contributions of higher and lower importance are spread throughout its content. The purpose of this section is to describe in a concise way which elements are new, at least to the author’s knowledge.

- Transformation of the algorithmic schemes of GPAD [36] and generalized dual fast gradient methods [19] to match our common framework, which is in agreement with the notation used in [38].
• Detailed complexity analysis of online computational overhead when fast gradient methods are used in a time-varying setup. To quantify the cost of the operations the notion of flops\(^1\) was adopted, similarly to [20], and the problem structure was fully taken into account, as in an efficient embedded implementation of the algorithms.

• Implementation of soft output constraints for Richter’s dual fast gradient method without the need of polyhedral inequality constraints in the formulation (see Section 3.2.5). The modification was inspired by ideas briefly discussed in [19, 23, 38] but our detailed derivation, that involves both quadratic and linear weights on the slack variables, is based on the KKT (Karush-Kuhn-Tucker) conditions of the inner problem.

• Investigation of new sparse formulations for MPC problems where both state and output dynamics are considered. We choose to name these variants \(xuy, xu\) and \(uy\) formulation depending on which variables are kept in the optimization vector (see Section 2.4). Their differences in complexity (e.g. Section 4.7) and performance (e.g. Section 5.4) are explored whenever interesting conclusions can be drawn.

• Application of an efficient sparse Cholesky factorization presented in [13] on the decomposition of the generalized matrix \(L\) of Richter’s dual fast gradient method (see Section 4.7.1).

• Detailed documentation of the available Lipschitz constants in GPAD, highlighting of equivalences and exploitation of algorithm’s structure to extend the scope of the generalized variant in a nonlinear MPC setup (see Section 6.2).

1.3 Outline

After a concise overview of the existing approaches for solving QP problems in the area of MPC, Chapter 2 presents the adopted linear time-varying optimal control problem and describes a number of QP formulations that can be derived from it. This part also establishes the notation that is used throughout the Thesis. Chapter 3 develops a common framework for all recently presented fast gradient method variants. The Alternating Direction Method of Multipliers (ADMM) is also part of the discussed first-order solvers as its impact on the control community is considered significant. The core of this Thesis is in Chapter 4, where we analyze the online complexity of time-varying optimal control problems when the underlying algorithm is one of the fast gradient method variants. In Chapter 5, a general MPC benchmarking suite is introduced [27] and used as a tool to highlight certain characteristics of the discussed solvers and formulations. The benchmarking suite together with the complexity results of Chapter 4 are used on an industrial nonlinear MPC example of ABB Corporate Research in Chapter 6 to illustrate their role in the selection of the most efficient implementation. Finally, Chapter 7 comprises a summary of the Thesis and suggests future work.

\(^1\)Floating point operations.
Chapter 2

Quadratic Programming in Model Predictive Control

Discrete-time linear optimal control problems that constitute the core of linear Model Predictive Control, are a special class of highly structured quadratic programming (QP) problems. Moreover, many algorithms for nonlinear MPC rely on fast solutions of QP sub-problems, as discussed for instance in [15]. Let us associate a general QP with the following optimization problem:

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2}z^T H z + f^T z \\
\text{subject to} & \quad Az = b \\
& \quad Cz \leq d,
\end{align*}
\]

(2.1)

with the vector of optimization variables \(z \in \mathbb{R}^n\), the Hessian matrix \(H \in \mathbb{S}^+_n\), the gradient vector \(f \in \mathbb{R}^n\), the equality constraint matrix \(A \in \mathbb{R}^{m_e \times n}\), the equality constraint vector \(b \in \mathbb{R}^n\), the inequality constraint matrix \(C \in \mathbb{R}^{m_i \times n}\) and the inequality constraint vector \(d \in \mathbb{R}^n\). When the gradient or constraint vectors depend linearly on a parameter \(w \in \mathbb{R}^m\), the QP is called \textit{parametric}.

If the above optimization problem was unconstrained or comprised only equality constraints, its solution would have an analytical expression.\(^1\) Therefore, the element that renders the solution non-trivial is the presence of inequalities. Existing numerical methods treat these inequalities in various ways, giving raise to several different approaches.

These approaches, together with their main characteristics are briefly summarized in Section 2.1. Section 2.2 presents the discrete-time linear time-varying MPC formulation that is adopted throughout this Thesis. The reason why we consider a time-varying setup stems from the fact that the data can potentially be the linearized counterparts of an underlying nonlinear process. Depending on the choice and ordering of optimization variables in the MPC formulation, the resulting QP of (2.1) has different size and structure. The possible choices are discussed in Sections 2.3 and 2.4. Exploiting our general setup that comprises states, inputs and outputs, we comment on special strategies that will become more meaningful as we discuss the properties of certain QP solvers.

\(^1\)This does not necessarily mean that an iterative approach would not be more efficient.
2.1 Survey of existing methods

In this section we summarize existing methods for solving QP problems that arise in MPC applications and mention relevant software implementations. An initial distinction is between explicit and iterative methods. The latter solve an optimization problem at each time step while the first ones pre-calculate all possible solutions and the online effort reduces to a search in a look-up table. Iterative methods can be further classified in active-set, interior-point and gradient methods, depending on the way they handle the inequality constraints. The last class of methods constitutes the main focus of this Thesis. Many more iterative approaches have been proposed that do not clearly fit in one of the categories above. Some of them that share the simplicity of the gradient methods are briefly discussed in the last paragraph. The survey on explicit, active-set and interior-point methods is taken literally from [15] to a large extend while the survey on gradient methods follows closely the analysis in [38].

**Explicit methods.** In cases where short sampling times render the online solution of an optimization problem computationally prohibitive, explicit methods might be the only alternative assuming that the problem has a reasonably small size and memory is cheap. These methods solve all possible QP instances ahead of time and the online burden reduces to a search of the stored solution in a look-up table. More precisely, the feasible state space is divided into polyhedra, called critical regions. Within these regions, the optimal solution is calculated as an affine function of the initial state. To solve the optimization problems arising at each time step, the polyhedron to which the initial state belongs to must be identified and the corresponding affine function must be evaluated.

A serious drawback of these methods is that the number of critical regions grows exponentially in the number of constraints, limiting their application to problems with few differential states and short horizons. Otherwise, offline computation time, memory requirements and online effort for finding the current critical region become prohibitively large. Approximate methods have been developed in an effort to overcome these difficulties. They extend the scope of explicit MPC to slightly higher dimensional problems at the cost of sub-optimal solutions.

A recent survey of explicit MPC methods can be found in [2] while an open-source implementation, comprising both exact and approximate methods is provided by the Multi-Parametric Toolbox (MPT) for MATLAB [22].

**Active-set methods.** The fundamental idea of active-set methods is to assume a set of active inequality constraints in the optimal solution and solve the resulting equality-constrained QP, which is computationally inexpensive. This set of active constraints is repeatedly updated until the optimal one is found. In principle, one can distinguish between two main variants of active-set methods depending on whether they solve the primal or the dual QP.

Primal active-set methods generate a sequence of primal feasible iterates given a feasible starting point. When also dual feasibility holds, the optimal solution is recovered. If no feasible starting point is provided by the user, the so-called Phase I is employed to generate one, or to conclude that the QP is infeasible. On the contrary, dual active-set methods, that essentially solve the dual QP using a primal active-set approach, do not need a Phase I since obtaining a dual feasible starting point is trivial. However, they are
only applicable to strictly convex QPs and intermediate solutions are not primal feasible any more. Note that due to the nested nature of the dual problem, dual methods iterate on both primal and dual variables.

Function \texttt{quadprog} in MATLAB implements a primal active-set method while dual active-set methods are available in the code \texttt{qpas} [41] and \texttt{QPSchur} [3]. Finally, an efficient implementation of a parametric active-set strategy designed for use in MPC is available in the open-source software \texttt{qpOASES} [17]. The latter is frequently used in our simulations to assess the quality of the solution of different first-order solvers.

\textbf{Interior-point methods.} A second important class of iterative QP solvers are the interior-point methods where we can again distinguish between two main variants: \textit{Primal barrier methods} and \textit{primal-dual} methods.

Primal barrier methods replace the inequality constraints in (2.1) with a weighted barrier function in the objective. Due to this function, violation of the inequality constraints becomes "infinitely" expensive. A frequently used barrier function is the logarithmic one [43]. In that case, the optimization problem becomes

\[
\begin{align*}
\text{minimize} \quad & \frac{1}{2} z^THz + f^Tz + \kappa \sum_{i=1}^{m_i} \log(C_i z - d_i) \\
\text{subject to} \quad & Az = b,
\end{align*}
\]  

(2.2)

where $C_i$ is the $i^{th}$ row of the inequality constraint matrix and $d_i$ the $i^{th}$ element of the inequality constraint vector. The resulting equality-constrained convex nonlinear programming problem is solved using Newton’s method. To ensure convergence to the solution of the original problem, the procedure is repeated with a decreasing value of $\kappa$. Changing factor $\kappa$ moderately ensures that the inner Newton iteration remains in the region of quadratic convergence and thus solves the inner problem within very few iterations.

Primal-dual methods combine the inner and outer loop of primal barrier methods by reducing $\kappa$ at each iteration of Newton’s method. Unlike primal barrier methods, both primal and dual variables are treated equally and are updated within each Newton iteration. Primal-dual methods avoid the need of a feasible starting point and they often work better in practice [7, 42]. However, fewer convergence results exist.

A code generation framework for a primal-dual method that handles generic multi-stage control problems and even supports quadratic inequalities is offered in [12].

\textbf{Gradient methods.} The gradient method is a well-established optimization method, based on the natural concept of iterative descent of the objective function. Its roots can be traced back to the work of Cauchy in 1847 [10]. An accelerated variant of this method was developed by Yurii Nesterov in 1983 [34], with a theoretical convergence that is provably faster than the one of the classic gradient method and the same iteration cost. The speed-up of the so-called \textit{fast gradient method} is due to the fact that not only the current, but also the previous iterate is taken into account in a controlled way so that some kind of "inertia effect" leads to faster convergence.

Research in gradient methods essentially disappeared by the time the first interior-point methods came up. However important properties like their simple algorithmic scheme and guaranteed convergence rates made them recently very popular to the control community.

\footnote{An extension to the general convex case is presented in [5].}
and especially to the field of embedded optimization. One of the leading contributors to this revival has been Stefan Richter and his PhD Thesis on computational complexity certification of gradient methods [38]. Results from this work inspired the content of this Thesis as different flavors of the fast gradient method, and more precisely of its latest generalization for constrained minimization called \textit{proximal gradient method}, are discussed in several sections.

A main characteristic of the fast gradient method that makes it suitable for embedded MPC applications is the calculation of low to medium accuracy solutions at a reasonable effort. Comparing to the previously described approaches, fast gradient method algorithms are easy-to-code, have low memory requirements and provide a lot of freedom for parallelization. Moreover, theoretical iteration bounds can guarantee convergence to a predefined accuracy that is comparable to the observed practical convergence. All these properties, make fast gradient methods considerable candidates for fast embedded optimization on dedicated hardware, like for instance Field-Programmable Gate Arrays (FPGAs). Depending on the type of constraints in the problem formulation, we distinguish between two main variants: \textit{Primal} and \textit{dual} fast gradient methods.

The primal fast gradient method, as its name suggests, solves the optimization problem in the primal domain. Since at each iteration the current iterate must be projected on the feasible set, a restrictive assumption is that this projection is easy-to-evaluate. In an MPC context this translates to optimal control problems with simple input constraints. The equality constraints imposed by the system dynamics must be incorporated in the objective function, to avoid projecting on the intersection of the two convex sets. Significant advantages of the primal fast gradient method are the linear convergence rate for strongly convex problems and the tight iteration bounds.

As soon as more complicated constraints enter the problem formulation, e.g. bounds on outputs, the projection on the feasible set is no longer trivial and the problem is solved in the dual domain. This implies that the guaranteed convergence rate changes from linear to the slower sub-linear rate, as strong convexity is lost. Moreover, conservatism is introduced in the iteration bounds since we can not derive a tight bound for the worst case minimal distance between an initial dual iterate and a Lagrange multiplier. Depending on which constraints are relaxed in the dual formulation, the equality or inequality constraints, we distinguish between two variants that are described in Sections 3.2 and 3.3 respectively. In [38], a partial Lagrange relaxation is proved to converge always faster than its fully relaxed counterpart. Finally, recently proposed generalized variants of dual fast gradient methods that allow for different step sizes in different gradient directions are also considered in our analysis [19]. Simulations have shown that they often improve the convergence speed by several orders of magnitude, especially for severely ill-conditioned problems.

\textbf{Other methods.} Further iterative methods have been proposed that do not clearly fit in one of the above categories. Among them, the Alternating Direction Method of Multipliers, or \textit{ADMM} [6], a multiplicative update dual optimization algorithm by Brand [9] and a primal-dual first-order method recently proposed in [29].

---

3To justify this we mention that the most computationally expensive part of each iteration is a matrix-vector multiplication.

4The ratio between the certified number of iterations and the practically observed one is less than one order of magnitude [38].
2.2 The MPC problem

The discrete-time linear optimal control problem that we will consider throughout this Thesis is the following:

\[
\begin{align}
\text{minimize} \quad & \sum_{k=0}^{N-1} \frac{1}{2} (y_k - y_r^k)^T \begin{pmatrix} Q_k & S_k \\ S_k^T & R_k \end{pmatrix} (y_k - y_r^k) + (q_k - q_r^k)^T (u_k - u_r^k) \\
& + \frac{1}{2} (x_N - x_r^N)^T Q_N (x_N - x_r^N) \\
\text{subject to} \quad & x_0 = x \\
& x_{k+1} = A_k x_k + B_k u_k + f_k \\
& y_k = C_k x_k + D_k u_k + e_k \\
& y_k^l \leq y_k \leq y_k^u \\
& u_k^l \leq u_k \leq u_k^u \\
& d_k \leq M_k y_k + N_k u_k \leq d_k^u \\
& d_N^l \leq T x_N \leq d_N^u.
\end{align}
\]

where \( x_k \in \mathbb{R}^{n_x} \), \( y_k \in \mathbb{R}^{n_y} \) and \( u_k \in \mathbb{R}^{n_u} \) denote the differential state, the process output and the control input at time \( k \), respectively. We assume a linear time-varying dynamic model, similarly to [27], which is subject to linear constraints that might also change in time. Aim is to track possibly time-varying reference values \( y_r^k \) and \( u_r^k \), while we assume the symmetric weighting matrices \( Q_k \in \mathbb{S}^{n_y} \) and \( R_k \in \mathbb{S}^{n_u} \) to be positive semi-definite. Moreover, a symmetric and positive semi-definite penalty \( Q_N \in \mathbb{S}^{n_x} \) as well as linear constraints on the terminal state \( x_N \) may be imposed for stability reasons. The dimensions of the inequality constraint matrices are defined here for later reference as \( M_k \in \mathbb{R}^{m_s \times n_y} \), \( N_k \in \mathbb{R}^{m_s \times n_u} \) and \( T \in \mathbb{R}^{m_t \times n_x} \). Formulation (2.3) generalises the common linear time-invariant MPC problem formulation to also allow for linear time-varying MPC formulations as they may arise in nonlinear MPC algorithms when linearizing the problem at the current nonlinear iterate. Also MPC problems based on a \( \Delta u \) formulation can be posed within this framework by introducing for instance auxiliary differential states.

The optimization problem of Equation (2.3) is a specially structured parametric QP with the initial state \( x \) as parameter. Since output and state trajectories are uniquely determined by the initial state and the control inputs \( u_k \), one can chose to eliminate part or all of these variables, yielding QP problems of different dimension and structure. In the special case where the optimal control problem comprises only inputs and states, we can clearly distinguish between two alternatives. Either eliminate all state variables and derive a small but dense QP or keep them in the formulation and form a large QP with exploitable structure. These two formulations are known in literature as condensed and sparse respectively. In the general case, where the output dynamics are also taken into account, there are in fact further choices to be considered. One can either keep all states, inputs and outputs as optimization variables, eliminate only the outputs or eliminate the states, yielding the different sparse formulations discussed in Section 2.4.
2.3 Condensed formulation

Choosing to eliminate all output and state variables from formulation (2.3) yields a dense QP with the smallest possible dimension. Using a vectorized notation, the problem can be re-written as

$$\begin{align*}
\text{minimize} & \quad \frac{1}{2} z^T H_c z + f_c^T(x) z \\
\text{subject to} & \quad C_c z \leq d_c(x) \\
& \quad z^l \leq z \leq z^u,
\end{align*}$$

where $z = [u_0^T \ u_1^T \ldots u_{N-1}^T]^T \in \mathbb{R}^{n_c}$, $H_c \in \mathbb{S}_{++}^{n_c}$, $f_c \in \mathbb{R}^{n_c}$, $C_c \in \mathbb{R}^{p_{i} \times n_c}$, $d_c \in \mathbb{R}^{p_i}$ and $n_c = Nn_u$. Constant terms are omitted in the optimization problem (2.4) since they only affect the optimal objective value and not the optimal solution. Some solvers handle the constraints by gathering simple and polyhedral inequalities together, yielding the equivalent QP:

$$\begin{align*}
\text{minimize} & \quad \frac{1}{2} z^T H_c z + f_c^T(x) z \\
\text{subject to} & \quad C_c z \leq d_c(x),
\end{align*}$$

with $\tilde{C}_c \in \mathbb{R}^{\tilde{p}_i \times n_c}$ and $\tilde{d}_c \in \mathbb{R}^{\tilde{p}_i}$. The new matrix and vector of inequality constraints are then simply formed as

$$\tilde{C}_c = \begin{bmatrix} C_c & I \\ -I \end{bmatrix}, \quad \tilde{d}_c = \begin{bmatrix} d_s \\ z^u_c \\ -z^l_c \end{bmatrix}.$$ (2.6)

The problem quantities $H_c, f_c, C_c$ and $d_c$ are provided in the Appendix, Section A.4.

Remark 2.3.1. Based on Equation (A.32a) of the Appendix, a sufficient condition for the Hessian matrix $H_c$ to be strictly positive definite is $\mathcal{R} \succ 0$, where $\mathcal{R}$ is a block diagonal matrix comprising the input weights $R_k$.

2.4 Sparse formulations

Not eliminating all or part of the equality constraints yields a QP of bigger dimension but with special structure. This structure can be both necessary for the employed solver and useful to perform certain operations efficiently. In all cases the resulting QP has the form

$$\begin{align*}
\text{minimize} & \quad \frac{1}{2} z^T H_s z + f_s^T z \\
\text{subject to} & \quad A_s z = b_s(x) \\
& \quad C_s z \leq d_s \\
& \quad z^l \leq z \leq z^u,
\end{align*}$$

with $H_s \in \mathbb{S}_{++}^{n_s}$, $f_s \in \mathbb{R}^{n_s}$, $A_s \in \mathbb{R}^{m_e \times n_s}$, $b_s \in \mathbb{R}^{m_e}$, $C_s \in \mathbb{R}^{m_i \times n_s}$ and $d_s \in \mathbb{R}^{m_i}$. The number of variables $n_s$ depends on the chosen sparse formulation and the available options are

8
discussed in the paragraphs that follow. Equivalently to (2.5), merging the inequalities yields the equivalent problem

$$\begin{align*}
\text{minimize} & \quad \frac{1}{2} z^T H_s z + f_s^T z \\
\text{subject to} & \quad A_s z = b_s(x) \\
& \quad \tilde{C}_s z \leq \tilde{d}_s,
\end{align*}$$

(2.8)

with $\tilde{C}_s \in \mathbb{R}^{\tilde{m}_i \times n_s}$ and $\tilde{d}_s \in \mathbb{R}^{\tilde{m}_i}$. Again terms $H_s, f_s, C_s$ and $d_s$ are given in the Appendix, Sections A.1 to A.3. Hereinafter, the dependency of the problem on the initial condition $x$ is dropped to simplify notation. Note that since in (2.5) and (2.8), all inequalities are grouped together, we have $\bar{p}_i = \bar{m}_i$.

**xuy formulation.** A first possible option is to keep all states, outputs and inputs as optimization variables. The result is an optimization problem of significantly large dimension but with very sparse matrices. In order to impose a banded structure that will be exploited in Chapter 4, the optimization variables are ordered as

$$z = [x_0^T \ u_0^T \ y_0^T \ \ldots \ y_{N-1}^T x_N^T]^T \in \mathbb{R}^{n_s},$$

(2.9)

with $n_s = N(n_x + n_u + n_y) + n_x$. Note that since in formulation (2.3), there are no weights on the state variables, $H_s$ will be always positive semi-definite, even if all input and output weight matrices are strictly positive definite. For solvers that only work when $H \succ 0$ a regularization term must be introduced and, depending on the existing weights, the resulting problem can be severely ill-conditioned. However, many of the methods analyzed in Chapter 3 are able to cope with this ill-conditioning very efficiently.

**xu formulation.** Eliminating the output variables via Equation (2.3d) but keeping the states as optimization variables, yields also a sparse QP but of smaller dimension. The optimization vector is sorted here as

$$z = [x_0^T \ u_0^T \ \ldots \ x_{N-1}^T u_{N-1}^T x_N^T]^T \in \mathbb{R}^{n_s},$$

(2.10)

with $n_s = N(n_x + n_u) + n_x$. An important difference is that even if matrices $Q_k, R_k$ are diagonal and $S_k$ is zero, the Hessian of the problem will be in general block diagonal as in Equations (A.6) and (A.7) of the Appendix. However, the algorithm of Section 3.2 needs a pure diagonal Hessian which renders this formulation unsuitable. As a special case, we consider problem instances where matrices $D_k$ are zero, $C_k$ do not couple any states. Then, the blocks in (A.7) are also diagonal and the algorithm can again be applied. We shall call this special case *xu* formulation.

Note that the common sparse formulation of MPC problems that only comprises states and inputs is a special case of this formulation. Setting $C_k = I, D_k = 0$ and $e_k = 0 \ \forall k = 0 \ldots N - 1$, equality (2.3d) yields $y_k = x_k$ and the outputs $y_k$ are eliminated from the problem simply by replacing them with $x_k$.

**uy formulation.** Finally, by condensing the states and keeping only inputs and outputs as optimization variables, we can form a QP with dense equality constraint matrix $A$, but sparse inequality constraints and objective function. Moreover, the dimension of the problem is reduced significantly, especially in cases with much larger number of states than
outputs. This alternative, that to the author’s knowledge has not been used so far in the literature, can be proven advantageous for the algorithm of Section 3.2 as it preserves the diagonal Hessian while reducing the amount of equality constraints significantly. Its complexity for the aforementioned algorithm is typically smaller than the other sparse formulations for small horizons and large number of states. More details on this matter are discussed in Chapter 4.

The ordering of the optimization variables does not play a significant role in this case. To match the previous orderings we can define the optimization vector as

\[ z = [u_0^T \ y_0^T \ \ldots \ u_{N-1}^T \ y_{N-1}^T \ x_N^T]^T \in \mathbb{R}^{n_s}, \tag{2.11} \]

with \( n_s = N(n_u + n_y) + n_x \). But we will also use in some parts the order

\[ z = [y_0^T \ \ldots \ y_{N-1}^T \ x_N^T \ u_0^T \ \ldots \ u_{N-1}^T]^T, \tag{2.12} \]

in order to ease notation and calculations. In case there are no constraints or weight on the terminal state \( x_N \), the latter can be excluded from the optimization variables. This statement holds also for the other two formulations but it is more significant here as the effect of the number of states on the complexity of the algorithm is reduced to the minimum.

**Remark 2.4.1.** For the sake of a unified notation, the condensed formulation of Section 2.3 is also referred to as u formulation in many parts of this Thesis.
Chapter 3

Overview of First-Order Methods

This chapter develops a unified framework for first-order Quadratic Programming solvers. Sections 3.1 to 3.3 present all recent variants of Nesterov’s fast gradient method [34] applied on MPC. Their advantage compared to other broadly applied approaches like active-set or interior-point methods is that they have a much simpler structure. This implies that they have lower complexity and memory requirements at the expense of more iterations for convergence. When applied in the primal domain, the fast gradient method comes with meaningful upper bounds on the number of iterations, which is a significant advantage in an embedded MPC context. In the dual domain, these bounds are more conservative but not necessarily impractical. Section 3.4 describes two variants of the Alternating Direction Method of Multipliers [6]. The convergence rates of this method are in general conservative while its observed convergence is often very competitive. A disadvantage comparing to the previous algorithms is that there are no known ways\(^1\) to optimally tune its penalty parameter \(\rho\).

3.1 Primal Fast Gradient Method

When the MPC problem formulation comprises only box input constraints\(^2\) (2.3f), the fast gradient method of Nesterov can be applied directly on the primal domain, yielding fast convergence rates, low complexity and tight iteration bounds (see [38] for a detailed analysis). This section provides an intuition on how the method works, presents the algorithm in a predictive control context and discusses a recently proposed generalized variant that allows for different step sizes in different directions.

3.1.1 Background

Assume we want to minimize a function \(f: \mathbb{R}^n \to \mathbb{R}\) subject to a set of constraints. Further assume that \(f\) is convex, differentiable and has a Lipschitz continuous gradient with constant \(L\). This implies that

\[
||\nabla f(z) - \nabla f(y)||_2 \leq L||z - y||_2,
\]

(3.1)

\(^1\)Optimal parameter tuning for some special cases is reported in [18]. However, when it comes to MPC, the assumptions to characterize this tuning as optimal are no longer valid and the resulting penalty parameter is only an "educated guess".

\(^2\)Or in general, when the feasible set has an easy-to-evaluate projection operator.
for all $z, y \in \mathbb{R}^n$ and it is equivalent to

$$f(z) \leq f(y) + \langle \nabla f(y), z - y \rangle + \frac{L}{2}||z - y||_2^2. \quad (3.2)$$

If we define $g(z)$ as the indicator function of the feasible set $C$ imposed by the constraints:

$$g(z) = \begin{cases} 0 & \text{if } z \in C \\ \infty & \text{otherwise,} \end{cases} \quad (3.3)$$

we can derive an equivalent unconstrained optimization problem with objective function

$$F(z) = f(z) + g(z). \quad (3.4)$$

Using (3.2) we can straightforwardly derive an upper bound for $F$,

$$F(z) \leq Q(z, y) := f(y) + \langle \nabla f(y), z - y \rangle + \frac{L}{2}||z - y||_2^2 + g(z), \quad (3.5)$$

for any $y \in \mathbb{R}^n$. Minimizing this upper bound over $z$ for a certain point $y_k$ yields the iterate:

$$z_k = \arg \min_z Q(z, y_k) = \arg \min_z \frac{L}{2}(z - y_k)^T(z - y_k) + (z - y_k)^T \nabla f(y_k) + g(z) = \arg \min_z \frac{L}{2}||z - \tilde{y}_k||_2^2 + g(z), \quad (3.6)$$

where $\tilde{y}_k = y_k - \frac{1}{L} \nabla f(y_k)$. This basically means that $z_k$ is the orthogonal projection of $\tilde{y}_k$ on the feasible set imposed by the constraints. One natural choice for the point $y_k$ is $z_{k-1}$, which leads to the standard gradient method for constrained optimization. The fast gradient method on the other hand, uses as $y_k$ a combination of the current and previous iterates in order to improve the convergence rate with some kind of inertia effect. Note that for the unconstrained case, Equation (3.6) reduces to $z_k = \tilde{y}_k$.

Based on the above analysis, that is taken from [44], we can conclude that "the fast gradient method solves a constrained optimization problem by minimizing a sequence of quadratic approximations of the objective function $F(z)$ at specially chosen points $y$."
Algorithm 1 Primal Fast Gradient Method

1: Initialize 0 < √μ/L ≤ α₀ < 1, y₀ = z₀ ∈ \mathcal{Z}
2: for k = 0 : K_{\text{max}} do
3:  z_{k+1} = \left[ y_k - L^{-1}(H_c y_k + f_c) \right]_{\mathcal{Z}} \quad \triangleright \text{gradient step and projection to feasible set}
4:  α_{k+1} = \frac{1}{2} \left( \sqrt{(α_k^2 - \frac{μ}{L})^2 + 4α_k^2 - (α_k^2 - \frac{μ}{L})} \right)
5:  β_k = a_k(1 - a_k)/(a_k^2 + a_{k+1})
6:  y_{k+1} = z_{k+1} + β_k(z_{k+1} - z_k)
7: end for

simply clipping the unconstrained solution to the feasible set. In our case, this leads to the update

\begin{equation}
    z_{k+1} = \left[ y_k - L^{-1}(H_c y_k + f_c) \right]_{\mathcal{Z}},
\end{equation}

where \([\cdot]_{\mathcal{Z}}\) represents the projection on the set \(\mathcal{Z} = \{ z \in \mathbb{R}^{n_c} : z^l \leq z \leq z^u \}\). The steps of the method are summarized in Algorithm 1. Since the objective function in (2.4) is quadratic, the convexity parameter and the Lipschitz constant of the gradient are simply

\begin{align}
    μ &= \lambda_{\text{min}}(H_c) \\
    L &= \lambda_{\text{max}}(H_c).
\end{align}

Note that \(H_c\) comprises both weights and system dynamics. Therefore, if these quantities are time-varying, the two parameters have to be calculated online (see Section 4.6).

3.1.3 Generalized variant

This section describes a generalization of the standard fast gradient method where the Lipschitz constant is replaced by an appropriate positive-definite matrix, potentially yielding faster convergence. The algorithm was first proposed in [44] for TV-based image restoration problems.

Given a positive-definite matrix \(L\), we define the \(L\) inner product as \(\langle x, y \rangle_L = x^T L y\) and the \(L\)-norm as \(||x||_L = \sqrt{\langle x, x \rangle_L}\). Based on these definitions, inequality (3.2) can be generalized in the following way:

\begin{equation}
    f(x) \leq f(y) + \langle \nabla f(y), x - y \rangle + \frac{1}{2} ||x - y||_L^2,
\end{equation}

while the optimization problem in Equation (3.6) becomes

\begin{equation}
    x_k = \arg \min_x g(x) + \frac{1}{2} ||x - \tilde{y}_k||_L^2.
\end{equation}

The motivation for replacing the scalar Lipschitz constant with a positive-definite matrix is that allowing different steps in different directions may lead to faster convergence. On the contrary, in the original method the step size is restricted to the inverse of the largest Lipschitz constant.
The authors prove that as long as (3.10) is satisfied, the convergence properties of the standard fast gradient method are preserved. Substituting the quadratic form of $f$ yields the equivalent condition

$$\frac{1}{2}(x - y)^T L(x - y) \geq \frac{1}{2}(x - y)^T H_c(x - y) \Rightarrow L \succeq H_c.$$  \hfill (3.12)

Moreover, to still be able to solve problem (3.11) analytically, matrix $L$ must be diagonal. These two conditions lead us to the following Semidefinite Programming (SDP) problem to determine the biggest permissible step $L^{-1}$:

$$\begin{align*}
\text{minimize} & \quad \text{trace}(L) \\
\text{subject to} & \quad L \succeq H_c,
\end{align*}$$  \hfill (3.13)

where $\mathcal{L}$ is the set of diagonal matrices. The standard fast gradient method is straightforwardly recovered when the Lipschitz constant is chosen as

$$L = ||H_c||^2 I.$$  \hfill (3.14)

**Remark 3.1.3.1.** The generalized primal fast gradient method is only computationally tractable when matrix $H_c$ is time invariant, since it is not desirable to solve the SDP in (3.13) online. This implies that both system dynamics and weight matrices must be constant.

### 3.2 Richter’s Dual Fast Gradient Method

The first dual fast gradient method variant, as proposed in [38], solves the input and state constrained MPC problem via partial Lagrange relaxation. More precisely, dual variables are introduced for the equality and polyhedral inequality constraints in (2.7) and the problem is solved in the dual domain. The inner problem, that comprises only simple constraints, is either solved analytically or using a nested fast gradient method.

As in the adopted MPC formulation both state and output dynamics are considered, the method can be applied on any of the sparse formulations presented in Section 2.4. Since each formulation yields a QP with different sparsity patterns and number of equality constraints, we expect both convergence speed and complexity to be affected. This topic is further discussed in Sections 4.7 and 5.4.

#### 3.2.1 Problem class

In its general form, as implemented in the code generation tool *FiOrdOs* [40], the algorithm solves constrained linear-quadratic problems with the following structure

$$\begin{align*}
\text{minimize} & \quad \frac{1}{2} x^T H x + f^T x + c \\
\text{subject to} & \quad A_i x \leq b_i \\
& \quad A_e x = b_e \\
& \quad x \in \mathcal{X} = \mathcal{X}_1 \times \ldots \times \mathcal{X}_N,
\end{align*}$$  \hfill (3.15)

with $x \in \mathbb{R}^n$, $A_i \in \mathbb{R}^{n_i \times n}$, $A_e \in \mathbb{R}^{n_e \times n}$ and Hessian $H \in \mathbb{S}^n_{++}$. The sets $\mathcal{X}_i$ are called *elementary simple sets*. An elementary simple set is closed, convex and has an easy-to-compute projection operator. For a detailed list of these sets the reader is referred
to [38]. Relaxing equality and polyhedral inequality constraints we derive the following dual problem

\[
\begin{align*}
\text{maximize} & \quad g(\lambda_e, \lambda_i) \\
\text{subject to} & \quad \lambda_i \geq 0,
\end{align*}
\] (3.16)

where \( g(\lambda_e, \lambda_i) \) is in turn the solution of the inner optimization problem

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2} x^T H x + f^T x + c + \lambda_e^T (A_e x - b_e) + \lambda_i^T (A_i x - b_i) \\
\text{subject to} & \quad x \in \mathcal{X}.
\end{align*}
\] (3.17)

The gradient of the outer optimization problem (3.16) is by inspection

\[
\nabla_\lambda g(\lambda) = \begin{bmatrix} A_e x^* - b_e \\ A_i x^* - b_i \end{bmatrix},
\] (3.18)

where \( \lambda = [\lambda_e^T \lambda_i^T]^T \) and \( x^* \) is the minimizer of the inner optimization problem (3.17). It is clear from the above that the calculation of the gradient at each iteration requires the exact solution of the inner problem. In FiOrdOs, the inner problem can be either solved iteratively, with a nested primal fast gradient method, or analytically if the available structure allows it.\(^3\) Using nested loops is not desirable for performance reasons and therefore we shall restrict our attention to the latter case. Namely, to QP problems with diagonal matrix \( H \). When this assumption holds, the optimization variables are decoupled and \( x^* \) is obtained simply by clipping the unconstrained solution to the feasible simple set \( \mathcal{X} \).

### 3.2.2 The algorithm

In this section, we describe how the fast gradient method is applied to the dual of the sparse QP in (2.7). We assume a diagonal Hessian\(^4\) \( H_s \) to avoid nested loops and no polyhedral inequality constraints because in practice they might slow down the convergence significantly. Algorithm 2 summarizes the method. Comparing to Algorithm 1, we note the extra step for the solution of the inner problem, the absence of the projection of operator since the Lagrange multipliers are unconstrained and the fact that the convexity parameter \( \mu \) is zero.

When the nested optimization problem in Step 3 of the algorithm is fully decoupled, its solution has the following analytical form:

\[
z_{k+1} = [-H_s^{-1}(f_s + A_s^T y_k)]_{z'_s \leq z_{k+1} \leq z_s^*}. \] (3.19)

If matrix \( C_s \) in (2.7) is non-empty, we replace \( A_s \) and \( b_s \) above with

\[
\hat{A}_s = \begin{bmatrix} A_s \\ C_s \end{bmatrix}, \quad \hat{b}_s = \begin{bmatrix} b_s \\ d_s \end{bmatrix},
\] (3.20)

and in Step 4 of the algorithm we clip to the positive orthant \( \mathbb{R}_+ \) all dual variables that correspond to inequality constraints. As mentioned in the beginning of this section, this modification might yield slower convergence rates as the Lipschitz constant of the gradient when forming \( \hat{A}_s \) is bigger and consequently the steps towards the optimal solution smaller.

---

\(^3\)For a limited class of problems, a third option is to solve the inner problem parametrically via explicit MPC. See for instance [37].

\(^4\)This implies that matrices \( Q_k, R_k \) in (2.3) are diagonal and coupling weights \( S_k \) are zero.
### Algorithm 2 Dual Fast Gradient Method

1: **Initialize** \( \alpha_0 = (\sqrt{5} - 1)/2, \lambda_0 \in \mathbb{R}^{m_e}, y_0 = \lambda_0 \in \mathbb{R}^{m_e} \)

2: **for** \( k = 0 : K_{\text{max}} \) **do**

3: \( z_{k+1} = \text{argmin}_{z \in Z} \frac{1}{2} z^T H_s z + f_s^T z + y_k^T (A_s z - b_s) \)  \( \triangleright \) inner problem

4: \( \lambda_{k+1} = y_k + L^{-1} (A_s z_{k+1} - b_s) \)  \( \triangleright \) gradient step

5: \( \alpha_{k+1} = (\sqrt{\alpha_k^4 + 4\alpha_k^2} - \alpha_k^2)/2 \)

6: \( \beta_k = a_k (1 - a_k)/(a_k^2 + a_{k+1}) \)

7: \( y_{k+1} = \lambda_{k+1} + \beta_k (\lambda_{k+1} - \lambda_k) \)

8: **end for**

### 3.2.3 Lipschitz constant

An important parameter of this method that greatly affects theoretical and practical convergence is the upper bound on the Lipschitz constant of the gradient. Since the inverse of \( L \) is the size of the gradient step, the tighter the bound on it, the faster the algorithm converges. Having assumed that \( H_s \) is strictly positive definite, a first option for the Lipschitz constant, that is provided in Theorem 9.1 of [38], is

\[
L = \frac{||A_s||_2^2}{\lambda_{\min}(H_s)}, \tag{3.21}
\]

where \( ||A_s||_2 \) is the spectral norm of the equality constraint matrix (i.e. \( \sigma_{\max}(A_s) \)) and \( \lambda_{\min} \) the smallest eigenvalue. The result is improved in Theorem 9.7 of the same report, when \( H_s \) is *not* a multiple of the identity matrix. The new bound is

\[
L = ||A_s H_s^{-1/2}||_2^2, \tag{3.22}
\]

and if \( H_s \) is indeed a multiply of the identity matrix, Equations (3.21) and (3.22) yield simply the same result. Note that (3.22) can be equivalently written in the following way

\[
L = ||A_s H_s^{-1} A_s^T||_2, \tag{3.23}
\]

as for example employed in [19]. Finally, we remind that for symmetric matrices, the maximum singular value coincides with the maximum eigenvalue.

### 3.2.4 Generalized variant

As already discussed in Section 3.1.3, the Lipschitz constant of the gradient in fast gradient methods can be seen as a quadratic upper bound on the objective function. This upper bound serves in every iteration of the algorithm as an approximation of the function to be minimized. If this bound does not approximate well the function, slow convergence is expected. By instead allowing for a quadratic bound with different curvatures in different directions, the newly proposed generalized fast gradient methods can improve significantly the convergence properties.
As shown in [19], the analogous condition to (3.12) for the positive definite matrix \( L \) to substitute the scalar Lipschitz constant is:

\[
L \succeq A_s H_s^{-1} A_s^T.
\]  

(3.24)

However, since in Step 4 of Algorithm 2 the variables \( \lambda \) are free and no projection on a feasible set is required, we can directly use \( L = A_s H_s^{-1} A_s^T \). As demonstrated in later simulations (see Section 5.2), this modification can greatly improve the practical convergence of the method and yield promising results even in severely ill-conditioned cases.

In the presence of polyhedral constraints, a part of the dual variables are restricted to the positive orthant and therefore matrix \( L \) must have a block diagonal structure with a full upper block \( (L_1) \), corresponding to the dual variables of the equality constraints and a diagonal lower block \( (L_2) \) for the dual variables of the inequality constraints. We denote the set of these matrices here as \( \tilde{L} \) and the corresponding SDP to calculate the desired matrix is the following:

\[
\begin{align*}
\text{minimize} & \quad \text{trace}(L) \\
\text{subject to} & \quad L \succeq \begin{bmatrix}
A_s \\
C_s
\end{bmatrix}
H_s
\begin{bmatrix}
A_s^T \\
C_s^T
\end{bmatrix}.
\end{align*}
\]

(3.25)

However, this option is not feasible in a time-varying setup since we cannot afford to solve an SDP online. Moreover, even in the Linear Time-Invariant (LTI) case, the convergence of the algorithm can be significantly worse, especially when polyhedral constraints are active in the optimal solution (see Section 5.3).

Remark 3.2.4.1. Simulations with the benchmarking suite presented in Section 5.1 showed that the performance of both standard and generalized variants of Algorithm 2 is also severely affected by low weights on outputs whose bounds are active on the optimal solution.

### 3.2.5 Implementation of soft constraints

In this section, we demonstrate how soft output bounds can be tackled without the use of polyhedral inequality constraints that can negatively affect the convergence of the algorithm. A similar modification for ADMM is proposed in [23] while the current implementation was motivated by ideas briefly discussed in [19,38].

Assume we want to soften the output constraints of Equation (2.3e) in the following sense:

\[
y_k^l - s_k \leq y_k \leq y_k^u + s_k,
\]

(3.26)

with \( s_k \in \mathbb{R}_+^n \). Each slack variable is assigned a quadratic and a linear weight which adds the following term in the objective function (2.3a)

\[
\sum_{k=0}^{N-1} \frac{1}{2} s_k^T W_k s_k + w_k^T s_k,
\]

(3.27)

with \( w_k \in \mathbb{R}_+^n \) and \( W_k \in \mathbb{S}_{++}^n \) diagonal. Adding the slacks \( s_k \) in the optimization variables and transforming the box constraints to polyhedral, could severely affect the performance of the discussed algorithm. In this section we will describe a way to tackle soft constraints without the need of additional Lagrange multipliers.
In the previously described version of the algorithm, the solution of the inner problem (Algorithm 2, Step 3) was obtained by clipping the unconstrained solution on the feasible set. Namely, the Cartesian product of all input and output bounds. The same approach is not applicable here due to the coupling introduced by the slack variables. However, since the equality constraints are eliminated in the inner problem, every slack variable \( s_i \) is coupled only with the corresponding output \( y_i \), where subscript \( i \) refers to the \( i \)th vector element of the \( k \)th time step.

Therefore, the inner problem is decomposed in \( N(n_u + n_y) \) sub-problems that can be solved independently. An instance of these problems, for a certain output and slack variable is:

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2} y_i^2 Q_{ii} + (f_i + \lambda^T A_i) y_i + \frac{1}{2} s_i^2 W_{ii} + w_i s_i \\
\text{subject to} & \quad y_i \leq y_i^u + s_i \\
& \quad y_i \geq y_i^l - s_i \\
& \quad s_i \geq 0,
\end{align*}
\]

where the constant terms have been dropped and the vector \( y_k \) in Algorithm 2 has been renamed to \( \lambda \) in order to avoid confusion with the output term \( y_i \). The subscript \( k \) is also omitted because it refers to the iteration number which is not relevant here. \( Q_{ii} \) and \( W_{ii} \) refer to the \( i \)th diagonal element of the relevant weight matrices at time step \( k \). \( f_i \) refers to the vector element of \( f \) that multiplies the \( i \)th output at the \( k \)th time step. \( w_i \) is the linear weight of \( i \)th slack at the \( k \)th time step and finally \( A_i \) represents the column of the matrix \( A \) that multiplies the \( i \)th output at the \( k \)th time step in (2.7).

Before we proceed, let us assume that the first two inequality constraints in (3.28) are missing and therefore \( y_i \) and \( s_i \) are decoupled. The optimal value for \( s_i \) would be zero in that case since the weights and the slack variable are non-negative. If in addition, the unconstrained solution \( y_i^{unc} \) happens to lie between the bounds \( y_i^l \) and \( y_i^u \), then the pair \( (y_i^{unc}, 0) \) is the optimal solution of the problem. We will prove this argument more rigorously by exploiting the KKT optimality conditions [25,30] of the problem. Moreover, we will derive formulas to calculate the optimal solution when \( y_i^{unc} \) is outside the given bounds.

Dualizing all inequality constraints and deriving the Lagrangian of the sub-problem yields the following expression

\[
\mathcal{L}(y_i, s_i, \mu_i, \nu_i, \xi_i) = \frac{1}{2} Q_{ii} y_i^2 + (f_i + \lambda^T A_i) y_i + \frac{1}{2} W_{ii} s_i^2 + w_i s_i + \\
+ \mu_i(y_i - s_i - y_i^u) + \nu_i(-y_i - s_i + y_i^l - \xi_i s_i),
\]

\[\text{(3.29)}\]

\[\text{5}\text{In practice, depending on the employed sparse formulation, there might be additional unconstrained sub-problems comprising the state variables but this fact is not relevant to our analysis here.}\]
while the corresponding KKT conditions are:

\[
\begin{align*}
Q_{ii}y_i + f_i + \lambda^T A_i + \mu_i - \nu_i &= 0 \quad \text{stationarity} \\
W_{ii}s_i + w_i - \mu_i - \nu_i - \xi_i &= 0 \quad (3.30a)
\end{align*}
\]

\[
\begin{align*}
y_i - s_i - y_i^* &\leq 0 \\
-y_i - s_i + y_i^* &\leq 0 \\
-s_i &\leq 0 \quad \text{primal feasibility} \\
\mu_i &\geq 0 \\
\nu_i &\geq 0 \quad \text{dual feasibility} \\
\xi_i &\geq 0 \\
\mu_i(y_i - s_i - y_i^*) &= 0 \\
\nu_i(-y_i - s_i + y_i^*) &= 0 \quad \text{complementary slackness.} \\
\xi_i s_i &= 0 \quad (3.30d)
\end{align*}
\]

We now distinguish between four cases for the unconstrained solution \(y_{i}^{unc}\).

**Inside bounds.** When the unconstrained solution lies within the given output bounds

\[
y_i^l \leq y_{i}^{unc} \leq y_i^u,
\]

then by inspection, setting \(s_i^* = 0, \mu_i^* = 0, \nu_i^* = 0, \xi_i^* = w_i\) and

\[
y_i^* = y_{i}^{unc} = -\frac{1}{Q_{ii}}(f_i + \lambda^T A_i)
\]

satisfies all the KKT conditions above and therefore it is an optimal solution. Moreover, as both \(Q_{ii}\) and \(W_{ii}\) are strictly positive, the optimization problem is strictly convex and this solution is unique.

**Close to bounds.** Due to the presence of the linear penalty on the slack variables, a region is formed next to the bounds where the violation is not strong enough to induce a slack variable. More precisely, these regions are

\[
y_i^u \leq y_{i}^{unc} \leq y_i^u + \frac{1}{Q_{ii}}w_i
\]

\[
y_i^l - \frac{1}{Q_{ii}}w_i \leq y_{i}^{unc} \leq y_i^l
\]

Guessing that in these regions the optimal solution is \(y_i^* = y_i^u\) and \(s_i^* = 0\), we will show that all KKT conditions hold, which implies that this is indeed an optimal solution. The same reasoning holds for the case where \(y_i^* = y_i^l\), due to symmetry of the problem.

First of all, setting \(\nu_i^* = 0\) implies that the complementary slackness condition holds. The first stationarity condition then yields

\[
Q_{ii}y_i^u + f_i + \lambda^T A_i + \mu_i^* = 0 \Rightarrow \mu_i^* = Q_{ii}(y_{i}^{unc} - y_i^u) \geq 0.
\]

Moreover, from the second stationarity condition we get the equality

\[
\mu_i^* = w_i - \xi_i^*.
\]
Imposing a positive Lagrange multiplier $\xi_i$ yields the condition
\[
w_i \geq \mu_i^* \Rightarrow \frac{1}{Q_{ii}} w_i \geq y_{i}^{\text{unc}} - y_i^u \Rightarrow y_i^u + \frac{1}{Q_{ii}} w_i \geq y_{i}^{\text{unc}},
\] (3.36)
which holds trivially from (3.33) in the region we are examining. It is straightforward to show that all remaining optimality conditions hold as well at this point.

**Outside upper bound.** When the violation of the unconstrained solution is bigger than the one indicated in (3.33), or in other words when
\[
y_{i}^{\text{unc}} > y_i^u + \frac{1}{Q_{ii}} w_i,
\] (3.37)
the slack variable must attain a positive value to compensate for it. Setting $y_i^* = y_i^u + s_i^*$, $\nu_i^* = 0, \xi_i^* = 0$ and substituting in (3.30a) yields
\[
\mu_i^* = W_{ii} s_i^* + w_i
\] (3.38)
and
\[
Q_{ii} y_i^* + f_i + \lambda^T A_i + W_{ii} s_i^* + w_i = 0 \\
\Rightarrow Q_{ii} y_i^* + (Q_{ii} + W_{ii}) s_i^* + f_i + w_i + \lambda^T A_i = 0 \\
\Rightarrow s_i^* = -\frac{1}{Q_{ii} + W_{ii}} (Q_{ii} y_i^* + f_i + w_i + \lambda^T A_i).
\] (3.39)
Note that using the unconstrained output as calculated in (3.32), we can reformulate the above equation to
\[
s_i^* = \frac{Q_{ii}}{Q_{ii} + W_{ii}} (y_{i}^{\text{unc}} - y_i^u - \frac{1}{Q_{ii}} w_i).
\] (3.40)

**Outside lower bound.** Using the same reasoning, when
\[
y_{i}^{\text{unc}} < y_i^l - \frac{1}{Q_{ii}} w_i,
\] (3.41)
the optimal solution is $y_i^* = y_i^l - s_i^*$ with a slack variable
\[
s_i^* = \frac{1}{Q_{ii} + W_{ii}} (Q_{ii} y_i^l + f_i - w_i + \lambda^T A_i) = \frac{Q_{ii}}{Q_{ii} + W_{ii}} (y_i^l - y_{i}^{\text{unc}} - \frac{1}{Q_{ii}} w_i).
\] (3.42)

### 3.3 Bemporad’s Dual Fast Gradient Method

The Accelerated Dual Gradient-Projection algorithm, or GPAD, solves input and state constrained MPC problems by relaxing the inequality constraints and keeping or condensing the equality constraints imposed by the dynamics of the system. An advantage of this method is that it can straightforwardly handle polyhedral constraints and therefore it is not restricted to simple sets. Moreover, when there are no constraints active in the
optimal solution, the cold-started algorithm converges in one step, since the optimal dual solution is the zero vector.

There are two flavors of the same algorithm proposed in literature (see [4,36]) depending on the way they treat the inner problem. The first variant eliminates all equality constraints and solves the dense unconstrained inner problem using the explicit inverse of the Hessian $H_c$ while the second keeps the inner problem sparse,\(^6\) employs a modified Riccati recursion to factorize the resulting KKT system and ultimately computes the optimal input and state trajectories in an efficient manner. Two major advantages of the second variant is that the complexity of the inner problem grows linearly with the horizon $N$ and that it is numerically stable even when the controlled plant is unstable. However, it has in general a smaller degree of parallelization. Which method is to be preferred depends on the dimensions of the given problem, on which quantities are time-varying and on the available hardware (see analysis in Section 4.8).

### 3.3.1 The algorithm

GPAD is also based on the second constant scheme of Nesterov [34], but its similarities with the previous method are not immediately apparent from the form of the algorithm in [4] or [36]. In Algorithm 3 the steps have been reformulated to match our common framework.

**Algorithm 3 GPAD**

1: **Initialize** $\alpha_0 = 1, y_0 = 0 \in \mathbb{R}^\hat{m}$

2: **for** $k = 0 : K_{\text{max}}$ **do**

3: $z_{k+1} = \arg \min_z \mathcal{G}(z) + y_k^T(Cz - \tilde{d})$ \(\triangleright \) inner problem

4: $\mu_{k+1} = \left[ y_k + L^{-1} \left( \tilde{C}z_{k+1} - \tilde{d} \right) \right]_+$ \(\triangleright \) gradient step and projection to $\mathbb{R}_+$

5: $\alpha_{k+1} = \left( \sqrt{\alpha_k^2 + 4\alpha_k^2} - \alpha_k^2 \right) / 2$

6: $\beta_k = a_k(1-a_k) / (a_k^2 + a_{k+1})$

7: $y_{k+1} = \mu_{k+1} + \beta_k(\mu_{k+1} - \mu_k)$

8: **end for**

Since the method introduces dual variables for all inequality constraints, polyhedral and simple, it is directly applied on the QP of (2.5) or (2.8). Although $\tilde{C}_c$ and $\tilde{C}_s$ are different in content and size, it is worth mentioning that the residuals $\tilde{C}_sz - \tilde{d}_s$ and $\tilde{C}_cz - \tilde{d}_c$ are always equivalent, as long as we preserve the order of constraints in both formulations. This is why the relevant subscripts are missing in Algorithm 3.

Function $\mathcal{G}(z)$ in Step 3 of Algorithm 3 is the objective function of the inner problem and it is therefore dependent on the formulation. More precisely we have

$$\mathcal{G}(z) = \begin{cases} \frac{1}{2}z^TH_cz + f_c^Tz & \text{for condensed formulation} \\ \frac{1}{2}z^TH_sz + f_s^Tz + I_\mathcal{E}(z) & \text{for sparse formulation} \end{cases}$$ (3.43)

\(^6\)By sparse formulation here we mean the xu formulation of Section 2.4, which coincides with what the authors suggest in the appendix of [4], when output equations are considered in the problem.
where \( I_E(z) \) is the indicator function for the equality constraints \( A_s z = b_s \) in QP (2.8), defined similarly to Equation (3.3). In both cases, the solution of the inner optimization problem has an analytic expression. If the problem is condensed:

\[
z_{k+1} = -H_c^{-1}(f_c + \tilde{C}_c^T y_k),
\]

while in case it is kept sparse:

\[
z_{k+1} = H_s^{-1} \left( A_s^T (A_s H_s^{-1} A_s^T)^{-1} (A_s H_s^{-1}(f_s + \tilde{C}_s^T y_k) + b_s) - \tilde{C}_s^T y_k - f_s \right),
\]

which follows from solving the KKT system of the equality-constrained optimization problem. In [36], the authors present an alternative method to calculate \( z_{k+1} \), whose complexity grows linearly with the horizon \( N \). Using a modified Riccati recursion, they initially factorize the KKT matrix and then the solution of the inner problem is reduced to one forward and one backward solve. For the LTI case, the factorization can be even performed offline. Moreover, this approach does not need the explicit inverse of \( H_s \) which implies that it can also handle positive semi-definite Hessian matrices, as long they are positive definite in the null-space of \( A_s \). Which of the three methods is more appropriate depends mainly on the sparsity of the matrices \( H_s, A_s, \tilde{C}_s \) and the time-varying nature of the problem data. For an efficient implementation, the one with the lower number of flops must be chosen.

**Equivalence between GPAD and Algorithm 3.** In this paragraph, we briefly show how GPAD was transformed from its original form, to match our existing framework. We begin by deriving an equivalent expression for the update of variable \( \beta \). Note that since each iteration involves only the current value of \( \beta \), subscript \( k \) can be omitted.\(^7\) According to Nesterov in [34], the update equation for the sequence of variables \( \alpha_k \) is

\[
\alpha_{k+1}^2 = (1 - \alpha_k) \alpha_k^2.
\]

Starting from the expression for \( \beta \) in Algorithms 2 and 3 and using Equation (3.46) we derive the equation of [36] in the following way:

\[
\beta_{k+1} = \frac{\alpha_k (1 - \alpha_k)}{\alpha_k^2 + \alpha_{k+1}} = \frac{\alpha_k^2 (\alpha_k^{-1} - 1)}{\alpha_k^2 + \alpha_{k+1}} = \frac{\alpha_k^2 (\alpha_k^{-1} - 1)}{\alpha_k + 1 (1 - \alpha_k^{-1})^{-1} + \alpha_{k+1}} = \frac{\alpha_k^2 (\alpha_k^{-1} - 1)}{\alpha_{k+1} (1 - \alpha_{k+1})^{-1} + \alpha_{k+1}} = \frac{\alpha_k^2 (\alpha_k^{-1} - 1)}{\alpha_{k+1} (\alpha_k^{-1} - 1)},
\]

where in the original algorithm, variables \( a_k \) are called \( \theta_k \) and the latter have a slightly different index because Step 7 of Algorithm 3 is Step 1 in [36]. These two steps can be interchanged as in the first iteration of the original algorithm, we get \( w_k = y_k \) and therefore Step 1 has no impact on the results the first time that is executed.

**3.3.2 Lipschitz constant**

As in all fast gradient methods, the smaller the bound on the Lipschitz constant, the bigger the gradient step and therefore the faster we converge to the optimal point. In this section we provide a concise overview of all the alternatives proposed in literature.

\(^7\)The same does not hold for \( \alpha \) since current and previous iterates are used in the expression for \( \beta \).
In [36], where the sparse variant of GPAD is presented, three options for \( L \) are discussed. The first two are analogous to Section 3.2.3, where the matrix of equality constraints is replaced by the matrix of inequality constraints. More precisely, we have

\[
L = \frac{||\tilde{C}_s||_2^2}{\lambda_{\text{min}}(H_s)}
\]

and

\[
L = ||\tilde{C}_s H_s^{-1/2}||_2^2 = ||\tilde{C}_s H_s^{-1} \tilde{C}_s^T||.
\]

However, both equations do not take into account the state dynamics of the system and therefore they are expected to yield less tight Lipschitz constants. Moreover, if \( H_s \) has not full rank these expressions are not applicable. Since in GPAD the inner problem can be always computed analytically, the dual function has an explicit form and the tightest Lipschitz constant is simply the spectral norm of its Hessian. This is the third option for \( L \) in [36] and the only option in [4]. Although the Hessian of the dual is the same independently of how the inner problem is solved, the one expression is derived from the sparse matrices and the other from the condensed counterparts. In the following sections we derive the dual problem for both cases, in order to determine their quadratic term and consequently the desired expression for the Lipschitz constant.

**Sparse formulation.** For the sparse formulation, we use the QP from (2.8). Since equality constraints are not relaxed, the inner problem has the following form

\[
\text{minimize } \frac{1}{2} z^T H_s z + f_s^T z + \mu^T (\tilde{C}_s z - \tilde{d}_s)
\]

subject to \( A_s z = b_s \).

Writing down the KKT optimality conditions yields

\[
\begin{bmatrix}
H_s & A_s^T \\
A_s & 0
\end{bmatrix}
\begin{bmatrix}
z^*(\mu) \\
\lambda^*(\mu)
\end{bmatrix} =
\begin{bmatrix}
-\tilde{C}_s^T \mu - f_s \\
b_s
\end{bmatrix},
\]

where \( \lambda \) are the dual variables for the equality constraints, that are not relevant for the algorithm. Assuming that \( H_s \) is strictly positive definite in the null-space of \( A_s \), the inverse of the KKT matrix is well defined and we can write

\[
\begin{bmatrix}
K_{11} & K_{12} \\
K_{21} & K_{22}
\end{bmatrix} =
\begin{bmatrix}
H_s & A_s^T \\
A_s & 0
\end{bmatrix}^{-1}.
\]

This gives us the following explicit solution for the inner problem

\[
z^*(\mu) = -K_{11} (\tilde{C}_s^T \mu + f_s) + K_{12} b_s.
\]

The dual problem, as a function of \( z^*(\mu) \), is

\[
\text{maximize } \frac{1}{2} z^*(\mu)^T H_s z^*(\mu) + f_s^T z^*(\mu) + \mu^T (\tilde{C}_s z^*(\mu) - \tilde{d}_s)
\]

subject to \( \mu \geq 0 \).

\[\text{Or the matrix comprising equalities and polyhedral inequalities in case } C_s \text{ is not empty.}\]

\[\text{i.e. } z^T H_s z > 0 \text{ whenever } z \neq 0 \text{ and } A_s x = 0, \text{ which is usually true in an MPC context.}\]
Using (3.53) and focusing on the quadratic term we get
\[
\frac{1}{2} \mu^T H_d \mu = \frac{1}{2} \mu^T \tilde{C}_s (2K_{11} - K_{11} H K_{11}) \tilde{C}_s^T \mu = \frac{1}{2} \mu^T \tilde{C}_s K_{11} \tilde{C}_s^T \mu,
\] (3.55)
where the rightmost equality is proved in [19] and with \( H_d \) we refer to the Hessian of the dual problem. Therefore, a tight Lipschitz constant is
\[
L = ||\tilde{C}_s K_{11} \tilde{C}_s^T||_2.
\] (3.56)

**Condensed formulation.** For the condensed formulation, we use problem (2.5). Since the equality constraints are condensed, the inner problem is unconstrained and equal to
\[
\min_{z} \frac{1}{2} z^T H_c z + f_c^T z + \mu^T (\tilde{C}_c z - \tilde{d}_c).
\] (3.57)
Setting the derivative to zero, we obtain the optimal solution of the inner problem
\[
z^*(\mu) = -H_c^{-1} (f_c + \tilde{C}_c^T \mu)
\] (3.58)
and the dual problem
\[
\max_{\mu} - \frac{1}{2} z^*(\mu)^T H_c z^*(\mu) - \mu^T \tilde{d}_c
\]
subject to \( \mu \geq 0 \).

Substituting \( z^*(\mu) \) from (3.58) and focusing on the quadratic term again we get
\[
\frac{1}{2} \mu^T H_d \mu = \frac{1}{2} \mu^T (\tilde{C}_c H_c^{-1} \tilde{C}_c^T) \mu.
\] (3.60)
Therefore, the Lipschitz constant that is derived from the condensed data is
\[
L = ||\tilde{C}_c H_c^{-1} \tilde{C}_c^T||_2.
\] (3.61)

Note that in the condensed formulation, \( H_c \) is positive definite under mild assumptions (see Section 2.3) and therefore the inverse of the Hessian here is well defined.

**Remark 3.3.2.1.** In a time-varying setup where the Lipschitz constant must be calculated online, the authors in [4] suggest the use of other norms that can upper bound this value. Alternatives are the Frobenius norm \( ||H||_F = \sqrt{\sum_{i,j=1}^{n} |H_{i,j}|^2} \) and the induced 1-norm \( ||H||_1 = \max_{j \in \mathbb{N}_{1,n}} \sum_{i=1}^{n} |H_{i,j}| \) which by symmetry of \( H \) coincides with the induced \( \infty \)-norm.

**Remark 3.3.2.2.** The matrices in (3.56) and (3.61) represent both the Hessian of the dual problem, which does not depend on the formulation. Therefore, the two expressions owe to be equal.

**Remark 3.3.2.3.** Taking a closer look at Algorithm 3, we notice that the way the inner problem is solved and the way the Lipschitz constant is calculated are independent. The reason being that \( z_{k+1} \) affects the algorithm only via the residual \( \tilde{C}_c z_{k+1} - \tilde{d} \) which is the same regardless of the way the primal vector is calculated. Therefore, we infer that any of the above Lipschitz constants can be used for either the sparse or the condensed variant of the algorithm. This remark is particularly useful in a nonlinear MPC setup, as we demonstrate in Chapter 4.
3.3.3 Generalized variant

To substitute the scalar Lipschitz constant with a positive definite matrix, the latter must satisfy the condition

\[ L \succeq \tilde{C}_s H^{-1} \tilde{C}_s^T. \]  

(3.62)

While in [19] the matrices above refer to the sparse formulation, the same holds for the condensed case. As discussed in the previous section, this is equivalent to the tighter condition

\[ L \succeq \tilde{C}_s K_{11} \tilde{C}_s^T, \]  

(3.63)

where \( K_{11} \) is defined in (3.52). Since all dual variables in this method must be positive, Step 4 of Algorithm 3 stays unchanged only if matrix \( L \) has a diagonal structure. In that case, we have to solve an SDP to determine the tightest generalized matrix. Namely,

\[
\min_{L \in \mathcal{L}} \text{trace}(L) \\
\text{subject to } (3.62) \text{ or } (3.63).
\]

(3.64)

Another approach, viable when \( H_s \) is diagonal and \( \tilde{C}_s \) very sparse, is to define the generalized matrix as \( L = \tilde{C}_s H_s^{-1} \tilde{C}_s^T + \epsilon I \) and replace Step 4 with the optimization problem

\[
\mu_{k+1} = \arg\min_{\mu \geq 0} \frac{1}{2} \| \mu - (y_k + L^{-1}(\tilde{C}z_{k+1} - \tilde{d})) \|_L,
\]

(3.65)

based on (3.11). The above problem should be solved parametrically for fast execution, as explained in [19]. Note that the term \( \epsilon I \) is added because in MPC, matrix \( \tilde{C}_s \) is usually thin and therefore \( \tilde{C}_s H_s^{-1} \tilde{C}_s^T \) is rank deficient.

3.4 Alternating Direction Method of Multipliers

The ADMM algorithm was first introduced in the mid-1970s by Gabay, Mercier, Glowinski, and Marrocco. However, it is based on operator splitting methods which can be traced back to the 1950s. In this section, we first briefly review certain optimization methods, precursors to the alternating direction method of multipliers, and then present two variants of the algorithm that can treat the sparse or condensed formulation of the MPC problem in (2.3).

3.4.1 Precursors of ADMM

This section describes the optimization methods that contributed towards the development of ADMM. For that we closely follow the analysis in [6].

**Dual Ascent.** Consider the equality-constrained convex optimization problem

\[
\min_z f(x) \\
\text{subject to } Gz = h,
\]

(3.66)

with Lagrangian function

\[
L(x, \lambda) = f(x) + \lambda^T(Gz - h).
\]

(3.67)
The corresponding dual problem comprises the unconstrained maximization of \( g(\lambda) \), which is defined as

\[
g(\lambda) = \inf_{x} L(x, \lambda).
\] (3.68)

The dual ascent method, solves the dual optimization problem iteratively via the following updates:

\[
x_{k+1} = \arg\min_{x} L(x, \lambda_k)
\] (3.69a)

\[
\lambda_{k+1} = \lambda_k + \alpha_k (Gx_{k+1} - h),
\] (3.69b)

where \( \alpha_k > 0 \) is the step size. The first step is a minimization over \( x \) and the second a dual variable update. Note that Step (3.69b) uses gradient information to update the dual variable, in the same way as gradient methods, since from (3.67), \( \nabla_{\lambda} g(\lambda_k) = Gx_{k+1} - h \).

If \( \alpha_k \) is chosen appropriately and several other assumptions hold (e.g. \( f \) strictly convex) \( x_k \) converges to an optimal primal and \( y_k \) to an optimal dual point.

**Dual decomposition.** Assume that the objective is separable, allowing to split the variable \( x \) into \( N \) sub-vectors \( x_n \in \mathbb{R}^{n_n} \). Then the objective function of (3.66) can be rewritten in the form

\[
f(x) = \sum_{n=1}^{N} f_n(x_n).
\] (3.70)

Matrix \( G \) can be partitioned column-wise so that \( Gx = \sum_{n=1}^{N} G_n x_n \) and therefore, the Lagrangian is also separable in \( x \). Namely,

\[
L(x, \lambda) = \sum_{n=1}^{N} L_n(x_n, \lambda) = \sum_{n=1}^{N} (f_n(x_n) + \lambda^T G_n x_n - \frac{1}{N} \lambda^T h).
\] (3.71)

This means that the minimization Step (3.69a) splits into \( N \) separate problems that can be solved in parallel. Their solutions are then gathered to perform the dual update in (3.69b). Note that this split is similar to the approach in Algorithm 2, when the inner problem has a diagonal Hessian and all optimization variables are independent.\(^{10}\)

**Method of multipliers.** In order to "robustify" the dual ascent method and guarantee convergence with less strong assumptions, the method of multipliers makes use of the **augmented Lagrangian**

\[
L_\rho(x, \lambda) = f(x) + \lambda^T (Gx - h) + \frac{\rho}{2} ||Gx - h||^2_2,
\] (3.72)

where \( \rho > 0 \) is a regularity parameter. Note that \( L_0 \) is the standard Lagrangian of the problem. The augmented Lagrangian can be thought of as the Lagrangian of the equivalent optimization problem

\[
\begin{align*}
\text{minimize} & \quad f(x) + \frac{\rho}{2} ||Gx - h||_2^2 \\
\text{subject to} & \quad Gx = h,
\end{align*}
\] (3.73)

\(^{10}\) Although in that case we have simple bounds instead of equality constraints
since for any feasible $x$, the additional term vanishes. The benefit of this modification is that $g_{\rho}(\lambda)$ can be proved to be differentiable under rather mild conditions on the original problem. Applying dual ascent to (3.73) yields the method of multipliers:

$$x_{k+1} := \arg \min_x L_{\rho}(x, \lambda_k)$$  \hspace{1cm} (3.74a)  
$$\lambda_{k+1} := \lambda_k + \rho(Gx_{k+1} - h).$$  \hspace{1cm} (3.74b)

This algorithm converges under far less restrictive assumptions than dual ascent, including cases when $f$ takes on the value $+\infty$ or it is not strictly convex. By using the regularity parameter $\rho$ as step size in the dual update, we guarantee dual feasibility of the iterates $(x_{k+1}, \lambda_{k+1})$. Primal feasibility is achieved on the limit, as $Gx_{k+1} - h \to 0$.

The greatly improved convergence properties of the method of multipliers over dual ascent do not come without a cost. The augmented Lagrangian is not separable anymore and therefore the algorithm can be decomposed. This issue is overcome by the Alternating Direction Method of Multipliers.

**Alternating Direction Method of Multipliers.** ADMM intends to blend the decomposability of dual ascent with the superior convergence properties of the method of multipliers. The algorithm solves problems of the form

$$\min_{x,z} f(x) + g(z)$$  
subject to $Ax + Bz = c$.  \hspace{1cm} (3.75)

The only difference from the general equality-constrained problem in (3.66) is that the variable, called $x$ there, has been split into two parts, called $x$ and $z$ here, and the objective function is separable across this splitting. As in the method of multipliers, the augmented Lagrangian is

$$L_{\rho}(x, z, \lambda) = f(x) + g(z) + \lambda^T(Ax + Bz - c) + \frac{\rho}{2}||Ax + Bz - c||_2^2,$$  \hspace{1cm} (3.76)

and ADMM consists of the following iterations:

$$x_{k+1} = \arg \min_x L_{\rho}(x, z_k, \lambda_k)$$  \hspace{1cm} (3.77a)  
$$z_{k+1} = \arg \min_z L_{\rho}(x_{k+1}, z, \lambda_k)$$  \hspace{1cm} (3.77b)  
$$\lambda_{k+1} = \lambda_k + \rho(Ax_{k+1} + B_{k+1} - c),$$  \hspace{1cm} (3.77c)

with $\rho > 0$. While in the method of multipliers the augmented Lagrangian is minimized jointly with respect to the primal variables, here $x$ and $z$ are updated in an alternating or sequential fashion, which accounts for the term *alternating direction*.

**3.4.2 The algorithm in sparse formulation**

In this section, we present an ADMM-based algorithm described in [23] for input and state (or output) constrained MPC. To ensure that the sub-problems in (3.77) can be solved analytically, the method allows only for simple constraints on the optimization variables.
Based on the sparse formulation in (2.7), we re-write the optimization problem at hand as

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2} z^T H_s z + f_s^T z \\
\text{subject to} & \quad A_s z = b_s \\
& \quad z^l \leq z \leq z^u.
\end{align*}
\]

(3.78)

Since projecting on the intersection of these constraints is known to be a complicated task, the algorithm partitions the variables into two groups to maintain the possibility of decoupled projections. We begin by introducing a copy \( x \) of the original decision variables (consensus form) and solve the equivalent problem:

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2} z^T H_s z + f_s^T z + \mathcal{I}_E(z) + \mathcal{I}_I(x) + \frac{\rho}{2} ||z - x||^2_2 \\
\text{subject to} & \quad z = x.
\end{align*}
\]

(3.79)

The functions \( \mathcal{I}_E : \mathbb{R}^{n_s} \rightarrow \mathbb{R} \) and \( \mathcal{I}_I : \mathbb{R}^{n_s} \rightarrow \mathbb{R} \) are indicator functions for the equality and inequality constraints in (3.78). For instance,

\[
\mathcal{I}_E(z) := \begin{cases} 
0 & \text{if } A_s z = b_s \\
\infty & \text{otherwise}
\end{cases}
\]

(3.80)

We remind that the inclusion of the regularizing term \( \frac{\rho}{2} ||z - x||^2_2 \) has no impact on the solution due to the equality constraint \( z = x \) while it allows us to drop the smoothness and strong convexity conditions on the objective function. The dual problem of (3.79) is

\[
\begin{align*}
\text{maximize} & \quad g_\rho(\lambda), \\
\text{where} & \quad g_\rho(\lambda) = \min_{z,x} \frac{1}{2} z^T H_s z + f_s^T z + \mathcal{I}_E(z) + \mathcal{I}_I(x) + \lambda^T (z - x) + \frac{\rho}{2} ||z - x||^2_2.
\end{align*}
\]

(3.81)

Similarly to (3.77), the algorithm iterates between the following three steps:

\[
\begin{align*}
z_{k+1} &= \arg \min_{z} L_\rho(x_k, z, \lambda_k) \quad \text{(3.83a)} \\
x_{k+1} &= \arg \min_{x} L_\rho(x, z_{k+1}, \lambda_k) \quad \text{(3.83b)} \\
\lambda_{k+1} &= \lambda_k + \rho(z_{k+1} - x_{k+1}). \quad \text{(3.83c)}
\end{align*}
\]

The value of the penalty parameter \( \rho \) is usually tuned during the implementation as it is problem-dependent. Step (3.83a) comprises the following equality-constrained convex optimization problem:

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2} z^T H_s z + f_s^T z + z^T \lambda_k + \frac{\rho}{2} (z^T z - 2z^T x_k) \\
\text{subject to} & \quad A_s z = b_s,
\end{align*}
\]

(3.84)

where \( x_k, \lambda_k \) are fixed and all constant terms have been dropped. Using the KKT optimality conditions, we will derive an analytical solution for (3.84). The Lagrangian of the problem is

\[
\mathcal{L} = \frac{1}{2} z^T (H_s + \rho I) z + z^T (f_s + \lambda_k - \rho x_k) + \mu^T (A_s z - b_s),
\]

(3.85)
and setting the gradient equal to zero yields
\[
\nabla_z \mathcal{L} = 0 \iff (H_s + \rho I)z + f_s + \lambda_k - \rho x_k + A_s^T \mu = 0 \iff (H_s + \rho I)z + A_s^T \mu = -f_s - \lambda_k + \rho x_k.
\]

Using (3.86) and imposing primal feasibility, results in the following linear system of equations
\[
\begin{bmatrix}
H_s + \rho I & A_s^T \\
A_s & 0
\end{bmatrix}
\begin{bmatrix}
z_{k+1} \\
\mu_{k+1}
\end{bmatrix} =
\begin{bmatrix}
-f_s - \lambda_k + \rho x_k \\
b_s
\end{bmatrix},
\]
\[
(3.87)
\]
where only vector \(z_{k+1}\) is relevant for the method. Note that in contrast to GPAD, the presence of the penalty parameter \(\rho\) let us drop the assumption on positive definiteness of \(H_s\) in the null-space of \(A_s\). Therefore, since \(A_s\) has by construction always full row rank, the inverse of the KKT matrix is always well defined. Setting again
\[
\begin{bmatrix}
K_{11} & K_{12} \\
K_{21} & K_{22}
\end{bmatrix} = \begin{bmatrix}
H_s + \rho I & A_s^T \\
A_s & 0
\end{bmatrix}^{-1},
\]
\[
(3.88)
\]
the solution of step (3.83a) in each ADMM iteration is
\[
z_{k+1} = K_{11}(-f_s - \lambda_k + \rho x_k) + K_{12}b_s.
\]
\[
(3.89)
\]
Step (3.83b), omitting the constant terms comprises an inequality-constrained optimization problem
\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2} x^T(\rho I)x - x^T(\lambda_k + \rho z_{k+1}) \\
\text{subject to} & \quad x^l \leq x \leq x^u.
\end{align*}
\]
\[
(3.90)
\]
Since the quadratic term above is diagonal and the constraints simple, the solution is obtained similarly to the inner problem of Algorithm 2. Namely, by clipping the unconstrained solution
\[
x_{k+1} = z_{k+1} + \frac{1}{\rho} \lambda_k
\]
\[
(3.91)
\]
on the feasible set. The summary of this ADMM variant is provided in Algorithm 4.

**Algorithm 4** ADMM: sparse formulation

1: Initialize \(x_0 \in \mathbb{R}^{n_s}, \lambda_0 \in \mathbb{R}^{n_s}, \rho \in \mathbb{R}^+\)

2: for \(k = 0 : K_{\text{max}}\) do

3: \(z_{k+1} = K_{11}(-f_s - \lambda_k + \rho x_k) + K_{12}b_s\)

4: \(x_{k+1} = \left[z_{k+1} + \frac{1}{\rho} \lambda_k\right]_{x^l \leq x_{k+1} \leq x^u}\)

5: \(\lambda_{k+1} = \lambda_k + \rho(z_{k+1} - x_{k+1})\)

6: end for

**Remark 3.4.2.1.** This variant of ADMM uses the same splitting of variables as GPAD of Section 3.3 [19].
Remark 3.4.2.2. The same approach as in Section 3.2.5 can be applied here to tackle soft constraints without introducing undesirable polyhedral inequality constraints [23].

3.4.3 The algorithm in condensed formulation

The dense QP (2.5) can be written on the standard ADMM form (3.75) by introducing a slack vector $x$ and assigning an infinite penalty on its negative components, as suggested in [18]. More precisely we have

$$\begin{align*}
\min_{z,x} & \quad \frac{1}{2}z^TH_cz + f_c^Tz + I_+(x) \\
\text{subject to} & \quad \tilde{C}_c z - \tilde{d}_c + x = 0,
\end{align*}$$

(3.92)

where $I_+$ is the indicator function for the positive orthant. The associated augmented Lagrangian is

$$\mathcal{L}_\rho = \frac{1}{2}z^TH_cz + f_c^Tz + \mathcal{I}_+(x) + \lambda^T(\tilde{C}_cz - \tilde{d}_c + x) + \frac{\rho}{2}||\tilde{C}_cz - \tilde{d}_c + x||^2.$$  

(3.93)

The minimization Step (3.77a) is unconstrained in this case, with analytical solution

$$z_{k+1} = -(H_c + \rho \tilde{C}_c^T \tilde{C}_c)^{-1}(f_c + \rho \tilde{C}_c^T (x_k - \tilde{d}_c + \frac{\lambda_k}{\rho})).$$  

(3.94)

On the other hand, Step (3.77b) after removing the constant terms is simply

$$\begin{align*}
\min_{x} & \quad \frac{1}{2}x^T(\rho I)x + (\lambda_k + \rho(\tilde{C}_cz_{k+1} - \tilde{d}_c))^T x \\
\text{subject to} & \quad x \geq 0.
\end{align*}$$

(3.95)

As in the previous variant, due to decomposable nature of the above optimization problem, the solution is recovered by clipping the unconstrained one on the positive orthant. Therefore we have

$$x_{k+1} = \left[-\tilde{C}_cz_{k+1} + \tilde{d}_c - \frac{\lambda_k}{\rho}\right]_+.$$  

(3.96)

The second ADMM variant is summarized in Algorithm 5.

**Algorithm 5** ADMM: condensed formulation

1: Initialize $z_0 \in \mathbb{R}^{n_c}$, $\lambda_0 \in \mathbb{R}^{\tilde{p}_i}$, $\rho \in \mathbb{R}_+$
2: for $k = 0 : K_{\text{max}}$ do
3: $z_{k+1} = -(H_c + \rho \tilde{C}_c^T \tilde{C}_c)^{-1}(f_c + \rho \tilde{C}_c^T (x_k - \tilde{d}_c + \frac{1}{\rho}\lambda_k))$
4: $x_{k+1} = \left[-\tilde{C}_cz_{k+1} + \tilde{d}_c - \frac{1}{\rho}\lambda_k\right]_+$
5: $\lambda_{k+1} = \lambda_k + \rho(\tilde{C}_cz_{k+1} - \tilde{d}_c + x_{k+1})$
6: end for
Chapter 4

Complexity of Fast Gradient Methods

In the present chapter, we analyze the computational complexity of a Quadratic Programming problem (QP) in the area of MPC, assuming that a fast gradient method is used as a solver. This complexity primarily depends on the chosen algorithm, formulation and on whether some of the variables in the problem formulation (2.3) are varying with time. All discussed algorithms are naturally divided in three parts. The problem transformation, where the MPC problem is converted to a specific QP formulation, the initialization of the algorithm and the iterative process. The first two parts are executed only once for each optimization problem while for the iterative process we estimate a complexity per iteration. To quantify the complexity we employ the notion of flops, i.e. floating point operations, as used in [20] to measure the efficiency of a program.

Section 4.1 introduces the flop count as a metric for the computational complexity of an algorithm. Section 4.2 summarizes the cost of important elementary operations that constitute the core of the subsequent analysis. Section 4.3 gathers all the necessary definitions and adopted notation. Section 4.4 describes the common structure of fast gradient methods. Section 4.5 aims to estimate the cost for transforming an MPC problem to a sparse or condensed QP formulation. Sections 4.6 to 4.8 analyze the complexity of the discussed fast gradient methods in the primal and dual domain. Finally, in Section 4.9, we provide a table to quickly assess the suitability of fast gradient methods under all feasible time-varying setups.

4.1 Measuring program efficiency

The efficiency of an algorithm depends on several parameters. The most important aspects that will concern us throughout this chapter are the amount of storage and required arithmetic. One way to quantify the required arithmetic, that we associate with the term computational complexity, is based on the notion of a flop. A flop is a floating point operation, i.e. an addition, multiplication or division.\(^1\) Although flop counting is a crude approach to the measuring of program efficiency, it can still provide insightful information regarding the scaling of the required arithmetic with respect to the problem dimensions. In an optimal control context, these dimensions are the number of states, inputs and outputs as well as the horizon length \(N\). Finally, since lower order terms do not contribute significantly to the flop count, they are omitted in our analysis similarly to the convention in [20].

\(^1\)Each operation contributes to the overall flop count independently, in contrast to another common convention where one multiplication followed by an addition cost only one flop.
4.2 Elementary operations

In this section, we assess the complexity of elementary operations that constitute the core of most numerical methods. For this assessment, we employ the notion of flops, as introduced in Section 4.1.

Matrix-vector multiplication

Multiplying a Matrix $A \in \mathbb{R}^{m \times n}$ with a vector $c \in \mathbb{R}^n$ requires $n$ multiplications and $n$ additions for each component of the result. Therefore in total $mn$ multiplications and $mn$ additions or

$$2mn \text{ flops. (4.1)}$$

In case matrix $A$ is sparse, one multiplication and one addition per zero element are avoided. The cost of the multiplication is then reduced to $2mn - 2n_z$, where $n_z$ is the total number of zeros in $A$. Finally, note that a matrix-vector multiplication can be efficiently parallelized, yielding much faster execution times on suitable platforms.

Matrix-matrix multiplication

When matrix $A \in \mathbb{R}^{m \times n}$ multiplies $C \in \mathbb{R}^{n \times p}$, we simply need $p$ times more additions and multiplications than in a matrix-vector product. This results in

$$2pmn \text{ flops. (4.2)}$$

Exploiting existing structure in matrices $A$ and $C$ might significantly reduce the required arithmetic. We examine here some special cases that will come up in the subsequent analysis.

1. When the product of $A \in \mathbb{R}^{m \times n}$ with another matrix is known to be symmetric (e.g. $AA^T$), we can reduce the amount of storage and number of floating point operations by not calculating all the elements of the result explicitly. More precisely, we need to calculate only the upper or lower triangular part that has $\frac{m^2 + m}{2}$ entries. Taking into account that each element requires $n$ multiplications and $n$ additions, the total cost is equal to

$$\frac{m^2 + m}{2} \cdot 2n \approx m^2 n \text{ flops, (4.3)}$$

instead of $2m^2n$ for the general case.

2. Assume $A, B \in \mathbb{R}^{m \times n}$ are lower triangular matrices. We can exploit this structure to calculate the total number of flops for the multiplication $AB^T$. We consider only the case where $m \leq n$ as it is more relevant to our analysis. The number of multiplications required to calculate the $k^{th}$ row of the result is

$$\left( \sum_{i=1}^{k} i \right) + (m-k)k = \frac{k}{2} (2m + 1 - k). \quad (4.4)$$

Summing for all rows yields

$$\sum_{k=1}^{m} \frac{k}{2} (2m + 1 - k) = \frac{m^3}{2} + \frac{m^2}{4} - \frac{m^3}{6} \approx \frac{m^3}{3} \quad (4.5)$$

32
multiplications, where to derive (4.4) and (4.5) we used the heuristics:

\[ \sum_{p=1}^{q} p = \frac{q(q+1)}{2} \approx \frac{q^2}{2} \]  

\[ \sum_{p=1}^{q} p^2 = \frac{q^3}{3} + \frac{q^2}{2} + \frac{q}{6} \approx \frac{q^3}{3}. \]  

(4.6a)  

(4.6b)

similarly to [20]. Taking into account the equal number of additions, we have in total

\[ \frac{2}{3} m^3 \text{ flops}, \]  

instead of \(2m^2n\) for the general case. This means that for square matrices, the cost is reduced by a factor of three.

3. Many matrices that we will encounter are structured in blocks. For cases where these blocks have the same dimensions, the previous results can be straightforwardly extended.

For example, assume \(E \in \mathbb{R}^{pm_b \times pn_b}\) is a matrix with \(p^2\) blocks of dimension \(m_b \times n_b\). Further assume that the structure of \(E\) is block triangular and we wish to calculate the product \(EE^T\). Then, following (4.5), we need \(\frac{p^3}{3}\) matrix-matrix multiplications of \(2m_b^2n_b\) flops each and \(\frac{p^3}{3}\) additions of \(m_b^2\) flops. The matrix additions can be ignored as they introduce only terms of lower order and therefore the cost is approximated by \(\frac{2}{3} p^3 m_b^2 n_b\) flops. The aforementioned number of flops should also be divided by two since the result of the multiplication is symmetric.

**Cholesky factorization**

Cholesky factorization is the decomposition of a matrix \(M \in \mathbb{S}_{++}^n\) into the product of a lower triangular matrix \(L\) and its transpose. Using Algorithm 4.2.1 of [20] for the factorization requires a total of

\[ \frac{1}{3} n^3 \text{ flops.} \]  

(4.8)

Having derived \(L\), we can compute the matrix-vector product \(y = M^{-1}x\) without explicitly calculating the inverse. More precisely, based on the following equivalent statements:

\[ My = x \]  

(4.9a)

\[ L(L^T y) = x \]  

(4.9b)

\[ Lz = x, \]  

(4.9c)

we can first calculate \(z\) solving system (4.9c) with a forward substitution and then \(y\) from the equality \(L^T y = z\) with a backward substitution. Each substitution induces a cost of \(n^2\) flops as shown in the next section.

**Forward/Backward substitution**

In the general case where the lower triangular matrix \(L\) from the factorization step is dense, solving the system in Equation(4.9c) requires

\[ \sum_{i=1}^{n} (2i - 1) = n(n + 1) - n = n^2 \text{ flops.} \]  

(4.10)
The cost for the backward substitution is by symmetry the same. When matrix $L$ is sparse, the complexity of the substitution can be greatly reduced as we see in later sections.

**Remark 4.2.1.** Due to the data dependency in a backward or forward substitution, the operation cannot be as efficiently parallelized as a simple matrix-vector multiplication.

**Matrix forward/backward substitution**

When more than one substitutions are performed with the same triangular matrix $L$, we can write the operations in the more compact form $LY = X$, where matrix $Y \in \mathbb{R}^{n \times m}$ is built column by column via $m$ forward substitutions. The complexity in this case is

$$mn^2 \text{ flops.}$$

(4.11)

Note that the operations can be performed in $m$ parallel threads for a more efficient implementation.

**Matrix inversion**

Given the Cholesky factorization of a positive definite matrix, we can calculate the explicit inverse in two different ways [28]:

- **Equation solving.** Using the identity $MM^{-1} = I$, we first substitute $M$ with its factorization, yielding $LL^TM^{-1} = I$. Setting $N = LL^T$, we can solve the system of equations $LN = I$ with one forward matrix substitution and subsequently the system $L^T N^{-1} = N$ with one backward matrix substitution in order to calculate the explicit inverse $M^{-1}$. When efficiently implemented, the matrix solves require $\frac{2}{3}n^3$ flops each [28], leading to a total complexity of

$$\frac{5}{3}n^3 \text{ flops.}$$

(4.12)

- **Triangular matrix operations.** An alternative way to calculate the inverse of $M$ is via the inverse of its Cholesky factor $L$. The decomposition $M = LL^T$ implies that $M^{-1} = (L^T)^{-1}L^{-1} = (L^{-1})^TL^{-1}$. The inverse $L^{-1}$ can be calculated with one matrix substitution from the system $LL^T = I$, that according to the previous method requires $\frac{2}{3}n^3$ flops. Given $L^{-1}$, $M$ can be simply calculated via one matrix-matrix multiplication that according to Equation (4.7) and taking into account the symmetry of the result requires $\frac{4}{3}n^3$ flops. The total cost of the inversion, including the Cholesky decomposition, is in this case

$$\frac{4}{3}n^3 \text{ flops.}$$

(4.13)

Different factorization schemes can be also used to derive the inverse matrix, e.g. the LU decomposition, but several properties of the Cholesky factorization made it the most popular approach in literature [8]. Note that if matrix $M$ is not positive definite, the inversion via Cholesky factorization requires two additional matrix-matrix multiplications. One to calculate the positive definite matrix $\tilde{M} = M^TM$ and one to calculate the inverse of the original matrix as $M^{-1} = \tilde{M}^{-1}MT$.

Finally, in the special case where matrix $M$ is diagonal, its inverse is calculated by simply inverting the diagonal elements. Therefore, the cost of calculating $M^{-1}$ is only $n$ flops. Recall also that the inverse of a block diagonal matrix is block diagonal and the inversion can be decomposed in $p$ parts, where $p$ is the number of blocks.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
<th>Cost (flops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix-vector multiplication</td>
<td>$f = Ac$</td>
<td>$2mn$</td>
</tr>
<tr>
<td>Matrix-matrix multiplication</td>
<td>$F = AC$</td>
<td>$2pmn$</td>
</tr>
<tr>
<td>Symmetric matrix-matrix multiplication</td>
<td>$S = AA^T$</td>
<td>$m^2n$</td>
</tr>
<tr>
<td>Dense Cholesky factorization</td>
<td>$M = \mathcal{L} \mathcal{L}^T$</td>
<td>$\frac{1}{3}n^3$</td>
</tr>
<tr>
<td>Forward/backward substitution</td>
<td>$\mathcal{L}c = e$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Matrix forward/backward substitution</td>
<td>$A\mathcal{L} = B$</td>
<td>$mn^2$</td>
</tr>
</tbody>
</table>

$A, B \in \mathbb{R}^{m \times n}, C \in \mathbb{R}^{n \times p}, F \in \mathbb{R}^{m \times p}, M, \mathcal{L} \in \mathbb{R}^{n \times n}, S \in \mathbb{S}^m,$

$c, e \in \mathbb{R}^n, f \in \mathbb{R}^m.$

Table 4.1: Complexity of elementary operations

Summary

The discussed elementary operations are summarized in Table 4.1. Using these operations and exploiting the structure of the QP matrices will enable us to estimate the complexity of all primal and dual fast gradient methods.

4.3 Preliminaries and notation

Before we proceed with the complexity analysis, we dedicate this section to establish a common notation and define some basic terms that are used extensively throughout this chapter.

Online and offline cost. The aim of this chapter is to estimate the complexity of algorithmic steps in terms of flops. However, for embedded MPC applications it seems reasonable to take into account only the operations that must be performed online. In our analysis, we assume that all quantities that can be pre-calculated induce zero (online) cost. In other words, the reported complexity should be interpreted as the amount of arithmetic that is required for the operation and cannot be transferred offline.

Varying problem data. The amount of calculations that can be transferred offline in an embedded MPC application depends solely on which problem quantities are varying and which remain constant. Assuming that at each sampling time the controller must solve an optimization problem of the form (2.3), each variable can be in either of the three following states: Constant, if its value does not vary throughout the closed-loop simulation, problem-varying, if it is constant in each optimization problem but varies from one QP instance to the next and finally time-varying if it varies also within the prediction horizon. The reason for the distinction between time-varying and problem-varying quantities is that the latter might require less arithmetic operations and less memory in certain parts of the analyzed methods. To keep our analysis tractable, we have sorted the problem variables in three groups:

- Dynamics - $A_k, B_k, C_k, D_k, f_k, e_k$.
- Weights - $Q_k, R_k, S_k, Q_N, q_k, r_k$.
• Constraints - \( M_k, N_k, T \).

Any of these groups can be constant, problem-varying or time-varying. As an example, consider an MPC formulation with the following parametric prediction model:

\[
x_{k+1} = A(p)x_k + B(p)u_k + f.
\]  

(4.14)

Assuming that the value of the parameter \( p \) changes every time the control input is applied, the dynamics of the problem should be characterized as problem-varying, according to our convention.

Finally, as some steps of the algorithms may depend on one or more of the aforementioned groups, it is convenient for the subsequent analysis to define the following binary variables:

\[
c_d = \begin{cases} 
1 & \text{if } tvd \\
0 & \text{otherwise}
\end{cases},
\]

\[
c_w = \begin{cases} 
1 & \text{if } tvd || tvw \\
0 & \text{otherwise}
\end{cases},
\]

\[
c_c = \begin{cases} 
1 & \text{if } tvd || tvc \\
0 & \text{otherwise}
\end{cases},
\]

(4.15)

where \( tvd \) stands for time-varying dynamics, \( tvw \) for time-varying weights, \( tvc \) for time-varying constraints and \( || \) represents the or operator.

**Terminal state terms.** In problem formulation (2.3), the terminal state \( x_N \) is both penalized via the weight \( Q_N \) and constrained via Equation (2.3h). Although these options are provided for stability reasons [32], they are not always necessary and in many problems they are omitted. Since especially for models with large number of states the complexity of an algorithmic step might be considerably affected by the inclusion of the terminal state in the optimization variables, we chose to annotate the respective terms with green color to make the distinction between optional and required terms more clear to the reader.

**Weights.** Although in the general problem formulation (2.3) we include coupling weights \( S_k \) and impose no restrictions on the weight matrices \( Q_k \) and \( R_k \), our attention in this chapter is restricted to a more special case. Namely, we only consider diagonal matrices \( Q_k, R_k \) and \( S_k = 0 \). The reason for this restriction is twofold. On the one hand, it is a minimum requirement for Algorithm 2 to run without nested loops and on the other hand, we simplify our calculations significantly by excluding cases that are not very common in practice.

### 4.4 Structure of Fast Gradient Methods

The complexity associated with the solution of a QP in the area of optimal control is affected by three main factors. The employed algorithm, the choice of optimization variables (i.e. formulation) and the time-varying nature of the problem data. All methods can be naturally divided into three main parts, the transformation of the MPC problem to a QP, the initialization of the algorithm and the iterative process. The first two parts are performed once for each optimization problem while the iterative part is executed as many times as the predefined (or required for convergence) number of iterations. Note that in a linear, time-invariant setup, both problem transformation and initialization can be performed ahead of time, inducing zero online cost. On the other hand, when all or

---

2Note that only the structure of the polyhedral constraints may affect the complexity of the algorithms, not changes in upper and lower bounds.

3Apart from the condition on positive definiteness.
Table 4.2: Structure of FGM and factors that affect each step.

<table>
<thead>
<tr>
<th>FGM parts</th>
<th>Factors that affect complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem transformation</td>
<td>Formulation, time-varying data</td>
</tr>
<tr>
<td>Initialization</td>
<td>Formulation, time-varying data, method</td>
</tr>
<tr>
<td>Iterations</td>
<td>Formulation, method</td>
</tr>
</tbody>
</table>

Some of the data are varying, a significant overhead is introduced that must be taken into account when assessing the suitability of a method from a computational point of view. In Table 4.2 we summarize the factors that affect the complexity of each entity.

**A. Problem transformation.** This part involves all the steps to convert the MPC problem of Equation (2.3) to a specially structured QP. It is performed only once for each optimization problem, it does not depend on the employed algorithm and it consists of the following entities:

1. **Optional condensing.** For the u and uy formulation, the outputs must be expressed as a function of the inputs and the initial condition. More precisely, matrices $\bar{A}, \bar{B}, \bar{F}$ of Equation (A.22) have to be calculated.

2. **Objective function.** The calculation of the quadratic and linear QP weights involve arithmetic operations only for the u and xu formulation.

3. **Polyhedral constraints.** Matrices $C_c, d_c$ for the u formulation or $C_s, d_s$ for the xu formulation are constructed in this step.

4. **Initial condition.** The incorporation of the initial state $x$ in the parameters of the QP induces an online cost only for the u and uy formulation. In contrary to the previous steps, the calculations have to be performed online even in the LTI case, since $x$ is not known ahead of time. However, the induced complexity is usually dominated by other operations, as discussed in the respective sections.

Note that the zero online cost of some steps under certain formulations is due to the fact that the operation might comprise only re-arrangement of matrices and vectors. This is clearly depicted in Table 4.3 of Section 4.5.

**B. Initialization.** The initialization part is also executed once and it involves all the pre-processing that is required for the iterative process. Since the latter is algorithm-specific, we discuss the steps of the initialization separately for each method.

**C. Iterative process.** Since the iterative part is executed several times, we calculate the cost per iteration instead of the total cost of the previous two parts. For all fast gradient method variants, the procedure can be divided into the following three steps:

1. Solution of inner problem (only in dual domain).
2. Gradient step and projection on feasible set.
3. Update of solution vector using a combination of current and previous iterates.

These steps, that contain at most matrix-vector products, depend on the algorithm and the chosen formulation but they are not affected by the time-varying nature of the problem data. Note that the update of the solution vector comprises only vector
additions and scalar-vector multiplications. Therefore, the complexity of the step is always dominated by the previous operations.

4.5 Complexity of problem transformation

As discussed in the previous section (see Table 4.2), building the QP out of the optimal control problem does not depend on the employed algorithm. The paragraphs that follow, analyze the complexity of each formulation under all possible time-varying cases.

4.5.1 xuy formulation

The main characteristic of this formulation is that it yields a very sparse QP. Therefore, the problem transformation requires no flops. The special structure, as well as the large problem size, will affect the initialization part of the methods as discussed in later sections.

4.5.2 xu formulation

Eliminating the output variables from the problem induces a cost that is associated with the calculation of the objective function and the polyhedral inequality constraints.

A2. Objective function

The Hessian matrix $H_k$ has the block diagonal structure of Equation (A.6) in the Appendix. Its first $N$ blocks, whose structure is shown in (A.7), include the matrix-matrix products $C_k^T Q_k C_k$, $C_k^T Q_k D_k$ and $D_k^T Q_k D_k$, while the last block comprises only the terminal state weight $Q_N$. Ignoring the operations on the diagonal weight matrices, the products require in total

$$N n_x^2 n_y + 2 N n_x n_u n_y + N n_u^2 n_y = N n_y (n_x + n_u)^2 \text{flops,}$$

Taking into account the symmetry of the results. The linear term $f_s$, defined by Equations (A.6) and (A.8), includes at most matrix-vector products and therefore its cost can be neglected.

The objective function of this formulation depends on the dynamics and the weights. If both are constant, the online cost of this step is zero. If any of the two is time-varying, we have to consider the amount of flops reported in (4.16) and finally, if both dynamics and weights are problem-varying, the dependency on the horizon vanishes as the first $N$ blocks of the Hessian are identical.

A3. Polyhedral constraints

Existing polyhedral constraints on the outputs, increase the flop count being expressed as a function of states and inputs in the following way:

$$d_k^l \leq M_k y_k + N_k u_k \leq d_k^u \Rightarrow d_k^l - M_k e_k \leq M_k C_k x_k + (M_k D_k + N_k) u_k \leq d_k^u - M_k e_k. \quad (4.17)$$

Therefore, the cost of this stage is in general

$$2 N m_x n_y (n_x + n_u) \text{flops}, \quad (4.18)$$

ignoring the matrix-vector products and matrix additions. Since the polyhedral constraints depend on the dynamics and the constraints, the online cost of the step is zero...
if both are constant. If any of the two is time-varying, we have to consider the amount of flops reported above and finally, if both are problem-varying, the dependence on the horizon \( N \) is dropped.

**Remark 4.5.2.1.** As a special case of the \( xu \) formulation, we consider the setup where there are no polyhedral inequality constraints and the Hessian \( H_s \) is diagonal. For the latter condition to hold, we assume that all \( D_k \) system matrices are equal to zero and matrices \( C_k \) are restricted to one non-zero element per row. In other words, each output is only allowed to be a multiple of a state, ensuring that the products \( C_k^T Q_k C_k \) are diagonal. This case, that we call from now on \( xu^* \) formulation has negligible problem transformation cost and its advantage is that it renders the \( xu \) formulation feasible for Algorithm 2.

### 4.5.3 uy formulation

When the states are eliminated from the formulation, the equality constraints become dense while at the same time objective function and inequality constraints remain sparse. The only steps of the problem transformation that require arithmetic operations are the condensing and the incorporation of the initial condition. The effect of the latter however is negligible unless the number of states is very large and the number of iterations small.

#### A1. Condensing

Condensing essentially means expressing the vector of output variables \( Y \), as a function of the inputs \( U \) and the initial condition \( x \). More precisely, we can write

\[
Y = \bar{A}x + \bar{B}U + \bar{F} + \mathcal{E},
\]

where the same notation as in Section A.4 of the Appendix is used. Observing the structure of matrices \( \bar{A} \) and \( \bar{B} \) and \( \bar{F} \) in Equation (A.23), we infer that for the \( k \)th time step we have to perform:

- \( k \) matrix-matrix multiplications of dimension \( \mathbb{R}^{n_y \times n_x} \cdot \mathbb{R}^{n_x \times n_x} \)
  \[p_1 = C_k A_{k-1}, \quad p_2 = p_1 A_{k-2}, \ldots \quad p_k = p_{k-1} A_0.\]
- \( k \) matrix-matrix multiplications of dimension \( \mathbb{R}^{n_y \times n_x} \cdot \mathbb{R}^{n_x \times n_u} \)
  \[g_1 = C_k B_{k-1}, \quad g_2 = p_1 B_{k-1}, \ldots \quad g_k = p_{k-1} B_0.\]
- \( k \) matrix-vector multiplications of dimension \( \mathbb{R}^{n_y \times n_x} \cdot \mathbb{R}^{n_x} \)
  \[v_1 = C_k f_{k-1}, \quad v_2 = p_1 f_{k-1}, \ldots \quad v_k = p_{k-1} f_0.\]

Neglecting the cost of matrix-vector multiplications and summing the operations up to step \( N - 1 \) yields a total of\(^4\)

\[
\left( \sum_{k=1}^{N-1} k \right) 2n_y n_x (n_x + n_u) = N(N - 1)n_y n_x (n_x + n_u) \approx N^2 n_y n_x (n_x + n_u) \text{ flops},
\]

where we used again the heuristic of Equation (4.6a). In case \( n_x \) is considerably bigger than \( N \) and the terminal state is part of the optimization variables, we should also sum

\(^4\)The proposed way for building matrices \( \bar{A}, \bar{B} \) and \( \bar{F} \) is not unique, but assuming \( n_y \leq n_x \) it requires the least amount of flops among the considered implementations.
the cost of the last equality constraint in (A.23) that comprises the dynamics of \(x_N\). This increases the complexity by approximately

\[
2Nn_x^2(n_x + n_u) \text{ flops.} \tag{4.21}
\]

Finally, in the LTI case where matrices \(\bar{A}, \bar{B}\) and \(\bar{F}\) have a simpler structure, we need to calculate only \(N-1\) products of dimension \((n_y \times n_x) \cdot (n_x \times n_x)\) and another \(N-1\) products of dimension \((n_y \times n_x) \cdot (n_x \times n_u)\). These operations yield the linear to the horizon cost of

\[
2Nn_x(n_x + n_u) \text{ flops.} \tag{4.22}
\]

For the terminal state equation, the complexity reported in Equation (4.21) is still valid.

**Remark 4.5.3.1.** The condensing step for both problem-varying and time-varying dynamics can be parallelized in various ways so as to reduce the complexity of the operation. In the time-varying case for instance, the products \(p_i, g_i\) and \(v_i\) can be calculated simultaneously in three separate threads.

### A4. Initial condition

In every sparse formulation, only the equality constraint vector \(b_s\) depends on the initial condition. Although in the xuy and xu formulation the incorporation of \(x\) comprises only its substitution in the first \(n_x\) entries of the vector, in the uy formulation we need to calculate the expression:

\[
b_s = \bar{A}x + \bar{F} + \mathcal{E}. \tag{4.23}
\]

In case the number of states is significantly big and the algorithm is executed for a small number of iterations, we may also want to take into account the complexity of the multiplication \(\bar{A}x\) that requires

\[
2Nn_xn_y + n_x^2 \text{ flops.} \tag{4.24}
\]

#### 4.5.4 u formulation

The condensed formulation introduces a significant overhead when the QP data are varying since it affects all parts of the problem transformation. Namely, one needs to condense the dynamics, calculate the objective function, form the polyhedral inequality constraints and incorporate the initial condition \(x\).

### A1. Condensing

The condensing is exactly the same as in the uy formulation. We need to calculate matrices \(\bar{A}, \bar{B}, \bar{F}\) of Equation (A.23) and the cost is again

\[
N^2n_yn_x(n_x + n_u) + 2Nn_x^2(n_x + n_u) \text{ flops,} \tag{4.25}
\]

with the quadratic dependence on the horizon becoming linear when the dynamics are problem-varying.
A2. Objective function

According to Equation (A.32a) of the Appendix, the cost for the computation of the Hessian is dominated by the product \( \bar{B}Q\bar{B}^T \), where the multiplication with the diagonal weight matrix does not contribute to the complexity. To ease our subsequent calculations we re-write this product as

\[
\bar{B}Q\bar{B}^T = \begin{bmatrix} \bar{B} \end{bmatrix} Q \begin{bmatrix} \bar{B}^T \end{bmatrix} = \begin{bmatrix} \bar{B}_1Q\bar{B}_1^T & \bar{B}_1Q\bar{B}_2^T \\ \bar{B}_2Q\bar{B}_1^T & \bar{B}_2Q\bar{B}_2^T \end{bmatrix},
\]

(4.26)

where \( \bar{B} \) has been divided in two parts. A block triangular upper part \( \bar{B}_1 \in \mathbb{R}^{n_y \times Nn_u} \) and a full lower part \( \bar{B}_2 \in \mathbb{R}^{n_x \times Nn_u} \) that corresponds to the dynamic equation of the terminal state. The blocks of \( \bar{B}_1 \) have all \( n_y \) rows and \( n_u \) columns. Therefore, we can apply the results of Section 4.2 to calculate the product \( \bar{B}_1Q\bar{B}_1^T \). This brings us to the cost of Equation (4.27), where the first term comes from the matrix-matrix products and the second from the matrix-matrix additions. The latter is only reported for the sake of completeness as products dominate additions and therefore the term has no contribution to the derived complexity. Note that the expression is divided by two because the final result is symmetric:

\[
\frac{1}{2} \left( \frac{N^3}{3} 2n_y^2n_u + \frac{N^3}{3} n_y^2 \right) = \frac{1}{3} N^3 n_y^2 n_u \text{ flops.}
\]

(4.27)

As \( n_x \) grows relatively to \( N \), the multiplications \( \bar{B}_1Q\bar{B}_2^T \) and \( \bar{B}_2Q\bar{B}_2^T \) affect the complexity of the product. To take these terms into account, we need to increase the cost by another

\[
N^2 n_x n_y + N n_x^2 n_u \text{ flops.}
\]

(4.28)

We remind that these terms appear only in case we penalize or constrain the terminal state \( x_N \), otherwise block \( \bar{B}_2 \) is empty.

The complexity of the linear term \( f_c \) is dominated by the product \( \bar{B}^T QA \). To estimate the cost of this operation we first focus on the multiplication of \( \bar{B} \) with a vector. Counting the number of zero entries in \( \bar{B} \), we find \( n_z \) zero blocks of dimension \( n_y \times n_u \), with

\[
n_z = \sum_{i=1}^{N} (N - i) = \frac{N^2 - 1}{2}.
\]

(4.29)

This yields \( \frac{1}{2} N^2 n_y n_u \) zero elements. A dense matrix-vector multiplication for the given dimensions requires \( 2Nn_u(Nn_y + n_x) \) flops. Taking into account the sparsity we have in total

\[
N^2 n_y n_u + 2N n_x n_u \text{ flops,}
\]

(4.30)

where we subtracted two times the amount of zero entries from the complexity of the full matrix-vector product. Multiplying this result with \( n_x \), i.e. the number of columns of \( A \), yields

\[
N^2 n_x n_y n_u + 2N n_x^2 n_u \text{ flops,}
\]

(4.31)

for the calculation of \( f_c \). As a final remark, note that the objective function of the \( u \) formulation also depends on the dynamics and the weights. If both are constant the online cost of this step is zero. However, if any of the two is varying, we have to consider the full amount of flops derived in this section.
A3. Polyhedral constraints

Equation (A.34) reveals that the dominating operations in the calculation of $C_c$ and $d_c$ are the multiplications $\mathcal{M}\overline{B}$ and $\mathcal{M}\overline{A}$. For the first product, we count

\[ N^2 m_s n_u + 2N m_t n_x n_u \text{ flops}, \quad (4.32) \]

while for the second

\[ 2N m_s n_u n_x + 2m_t n_x^2 \text{ flops}. \quad (4.33) \]

Since the polyhedral constraints in the $u$ formulation depend on the dynamics and the constraints, we infer that if both are constant the online cost of the step is zero. If any of the two is time-varying, we have to consider the amount of flops reported above. Finally, if both are only problem-varying, the quadratic dependence on the horizon in Equation (4.32) becomes linear.

A4. Initial condition

In this last section we estimate the complexity for condensing the initial condition $x$. This cost is always present, even in the LTI case, and it is not negligible when the number of states is big and the algorithm is executed for a small number of iterations. Since $x$ affects the linear weight $f_c$ and the inequality constraint vector $d_c$, we estimate the number of flops for the calculation of these quantities. The first term, according to Equation (A.32b), requires the matrix-vector multiplication $(\overline{B}^T \textbf{Q}\overline{A})x$ which requires

\[ 2N n_u n_x \text{ flops}. \quad (4.34) \]

The second term, based on (A.34), comprises the products $\overline{A}x$ and $\mathcal{M}\overline{A}x$ which in turn require

\[ 2N n_x (n_y + m_s) + 2n_x^2 + 2m_t n_x \text{ flops}. \quad (4.35) \]

4.5.5 Conclusions

A summary of the complexity for the problem transformation is provided in Table 4.3. The binary variables $c_d, c_w, c_c$ signal cases where problem and time-varying data require different amount of floating point operations. In the parenthesis next to each step we annotate the dependency on the problem data. The letter $d$ stands for dynamics, $w$ for weights and $c$ for polyhedral inequality constraints. If any of the quantities in the parenthesis is not constant, the complexity of the step should be recovered from the table. Otherwise, the step can be transferred offline and the online complexity is zero.

4.6 Complexity of Primal FGM

The first algorithm we analyze is the primal fast gradient method for the input-constrained MPC problem as discussed in Section 3.1. Since the feasible set of the optimization problem must have an easy-to-evaluate projection operator, this method is applied to the condensed formulation, where the QP that is derived from (2.3) comprises only simple input bounds and no equality constraints.
Problem transformation steps

<table>
<thead>
<tr>
<th>Condensing (d)</th>
<th>Objective function (d,w)</th>
<th>Constraints (d,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xuy</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xu</td>
<td>0</td>
<td>(Nc = n_y(n_x + n_u)^2)</td>
</tr>
<tr>
<td>xu*</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>uy</td>
<td>(2N \left(\frac{N}{2}\right)^c_d n_y n_x (n_x + n_u))</td>
<td>0</td>
</tr>
<tr>
<td>u</td>
<td>(2N \left(\frac{N}{2}\right)^c_d n_y n_x (n_x + n_u))</td>
<td>(Nc N m_s n_y n_u + 2 N m_s n_y n_x)</td>
</tr>
<tr>
<td></td>
<td>+ (2N n^2_x (n_x + n_u))</td>
<td>0</td>
</tr>
</tbody>
</table>

\(c_d, c_w, c_c\) as defined in Equation (4.15)

Table 4.3: Complexity summary of problem transformation

4.6.1 \(u\) formulation

After the problem is condensed following the steps of Section 4.5.4, the only operation that the initialization part comprises is the calculation of the parameters \(\mu\) and \(L\). The iterative process, since there is no inner problem to solve in the primal domain, consists of the gradient step, the projection on the feasible set and the iterate update.

B1. Calculation of \(L\) and \(\mu\)

Two parameters that are crucial for the convergence of the method are the Lipschitz constant of the gradient \(L\) and the convexity parameter \(\mu\). Since the objective function in our case is quadratic, \(L\) is equal to the maximum and \(\mu\) to the minimum eigenvalue of the Hessian. However, if either the dynamics or the weights are varying and the Hessian must be calculated on the fly, performing an online eigenvalue decomposition might not be computationally attractive. Note that such an operation requires

\[9N^3 n^3_u\] flops, \((4.36)\)

when a standard tridiagonalization-based algorithm is used (see Table 3.1 in [33]). Quoting [39], "it suffices to have an upper bound on the Lipschitz constant \(L\) and a lower bound on the convexity parameter \(\mu\) in order to make the algorithmic scheme of fast gradient methods work\(^5\). According to Equation (A.32a) the minimum eigenvalue of the Hessian is lower-bounded by the minimum element of the diagonal matrix \(R\). For the parameter \(L\) on the other hand, since \(H_c\) is symmetric and the eigenvalues coincide with the singular values, we can use the 1-norm or\(^6\) the Frobenius norm to upper-bound \(\|H_c\|_2\). Finally, another alternative for problems with unknown parameters \(L\) and \(\mu\) is a backtracking variant of the algorithm, discussed in [35].

\(^5\)The tighter the bounds on the parameters, the better the convergence rates.

\(^6\)Recall that \(\|H_c\|_1 = \|H_c\|_\infty\) due to symmetry of the matrix.
C2. Gradient step

We remind that the gradient step of Algorithm 1, neglecting the projection that is computationally cheap, can be written as:

\[ z_{k+1} = y_k - L^{-1}(H_c y_k + f_c), \]  

(4.37)

where \( L \) is scalar in the standard case and diagonal in the generalized variant. In both approaches, the most computationally expensive part of the expression is the matrix-vector product \( H_c y_k \) which trivially requires \( 2N^2n_u^2 \) flops. \(^{(4.38)}\)

C3. Iterate update

This step comprises only two vector additions and a scalar-vector product that require in total \( 3Nn_u \) flops. Clearly, this complexity is dominated by the expression in (4.38) and it can be therefore neglected. The same reasoning hold also for all dual gradient methods.

4.7 Complexity of Richter’s Dual FGM

In this section we calculate the computational complexity of the generalized variant of Algorithm 2. To apply the method, we need to assume diagonal positive definite matrices \( Q_k, R_k, Q_N, \) zero coupling weights \( S_k \) and no polyhedral inequality constraints\(^7\) \( M_k, N_k, T \) as explained in Section 3.2. Feasible in this context are the xuy, uy and xu\(^*\) formulations that preserve the diagonal structure of the Hessian. Therefore, from the problem transformation steps discussed in Section 4.4, only the condensing of the uy formulation contributes to the complexity of the algorithm. The initialization part for all formulations comprises the following operations:

1. Inversion of the Hessian matrix \( H_s \).
2. Calculation of the generalized matrix \( L \).
3. Factorization of \( L \) (and optional calculation of the inverse).

Aim of this section is to assess the computational complexity of each operation for the three applicable formulations. For the complexity of the iterative process we estimate the cost per iteration, since the involved steps are executed multiple times. As each formulation has advantages and limitations, the optimal choice depends on the control problem and available hardware. Before we proceed, we shortly discuss where each variant can be useful in practice.

**xuy formulation.** Keeping all states and outputs as optimization variables, results in a large problem but of substantial sparsity and structure that simplify many steps of the algorithm. A disadvantage of this approach is that since the method needs a strictly positive definite Hessian, we have to introduce a small weight on the states which in fact deteriorates the conditioning of the problem and slightly alters the optimal solution. The generalized matrix \( L \) that replaces the Lipschitz constant in the gradient step is very effective against ill-conditioning. However, tuning the regularization constant to

\(^7\)Except for simple bounds on the terminal state \( x_N \) that are expressed via matrix \( T \).
balance the deviation from the optimal solution and the numerical instability is a problem-dependent task. Among others, advantages of this formulation is that it imposes no restrictions on the output equations and that due to its sparsity, the dependence of the complexity on the horizon is linear.

**xu** formulation. Eliminating the outputs is only feasible when all system matrices $D_k$ are zero and the products $C_k^T Q_k C_k$ are diagonal, in order to maintain the purely diagonal structure of the Hessian. The special case where no output equations are present, i.e. $y = x$, falls into this category. When applicable, the xu formulation should be preferred over the xuy formulation as it maintains the same sparsity while it yields a smaller problem size. However, it also requires the use of a regularization term when not all states are mapped to the outputs, since the products $C_k^T Q_k C_k$ become rank deficient.

**uy formulation.** Finally, by condensing the states, we avoid the use of a regularization term and reduce the number of optimization variables since the equality constraints are fewer. However, the resulting problem loses its useful structure and the dependence on $N$ becomes quadratic or even cubic, for the factorization of the generalized matrix $L$. Typically this approach is advantageous in problems with short horizons and large number of states. Note that using the generalized variant, both xuy and uy formulations need the same number of iterations to converge, as demonstrated in Section 5.4. Therefore, the right choice depends on the problem size and the available hardware and memory resources.

**4.7.1 xuy formulation**

In this section, we analyze in detail the complexity of initialization and iterative steps for the xuy formulation of Algorithm 2.

**B1. Inversion of Hessian**

The inversion of the diagonal matrix $H_s$ comprises in general

$$N(n_y + n_u + n_x) \text{ flops,} \quad (4.39)$$

namely divisions. If the weights are not time-varying, the cost is instead $n_y + n_u + n_x$ flops while if they are constant, the inverse can be calculated offline and the online cost is zero. In all three cases, this cost is inferior to the other initialization steps and therefore it is neglected in our analysis for all discussed formulations.

**B2. Calculation of generalized matrix $L$**

The calculation of the generalized matrix $L = A_s H_s^{-1} A_s^T$ can be performed by exploiting the structure of the involved matrices. More precisely, $L$ in this formulation has a special tridiagonal block structure which is shown below:

---

8Provided that $Q_k, R_k ≻ 0$.

9The inverse of the regularization term can be always performed offline.
Therefore, the cost of the product is in total
\[ N(n_y + n_x)^2(n_x + n_u) \text{ flops.} \]  

If both dynamics and weights are problem-varying, the dependency on \( N \) vanishes as all blocks are identical. Finally, if they are both constant, the online cost is zero.
B3. Factorization of $L$

Matrix (4.40) has a special structure that allows for an efficient, block-wise factorization. This factorization can either be used to calculate the inverse or to directly calculate the product of $L^{-1}$ with a vector, as discussed in Section 4.2.

Since $A_s$ has by construction full rank (see Equation (A.3)) and $H_s$ is a diagonal positive definite matrix, the product $L = A_sH_s^{-1}A_s^T$ is always symmetric and positive definite. Therefore, $L$ can be factorized via Cholesky decomposition as $L = \mathcal{R}^T\mathcal{R}$ with $\mathcal{R}$ an upper triangular matrix. We use an upper triangular matrix $\mathcal{R}$ for the decomposition here ($\mathcal{R} = L^T$), in order to avoid any confusion between matrix $L$ and its Cholesky factor. The triangular matrix $\mathcal{R}$ has the following bidiagonal block structure:

$$\mathcal{R} = \begin{bmatrix}
R_{1,1} & R_{1,2} & 0 & \cdots & 0 \\
0 & R_{2,2} & R_{2,3} & \cdots & 0 \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
0 & 0 & \cdots & R_{N,N+1} \\
0 & 0 & \cdots & R_{N+1,N+1}
\end{bmatrix}.$$  

(4.42)

The factorization can be calculated sequentially from top to bottom using the algorithm proposed in [13]. This method comprises the following three steps:

$$L_{1,1} = R_{1,1}^T R_{1,1}$$  

(4.43a)

$$L_{k,k+1} = R_{k,k}^T R_{k,k+1} \quad 1 \leq k \leq N$$  

(4.43b)

$$L_{k,k} - R_{k-1,k}^T R_{k-1,k} = R_{k,k}^T R_{k,k} \quad 2 \leq k \leq N + 1$$  

(4.43c)

where the second equation is solved with respect to $R_{k,k+1}$ via matrix forward substitution and Equations (4.43a),(4.43c) by Cholesky factorization of the matrices on the left hand side. The overall cost of the factorization is

$$N(n_y + n_x)n_z^2 + Nn_x(n_y + n_x)^2 + \frac{1}{3}N(n_y + n_x)^3,$$  

(4.44)

where we have taken into account the sparsity of matrices $L_{k,k+1}, R_{k,k+1}$. The first term refers to the forward substitution, the second to the symmetric matrix multiplication and the third to the dense Cholesky factorization. Note that $R_{1,1}$ is the Cholesky factorization of the diagonal matrix $\epsilon_x^{-1}$ and therefore it can be calculated ahead of time.

The complexity of this step is the same for problem or time-varying data. Only when both dynamics and weights are constant, the factorization can be performed offline and the online cost becomes zero.

Remark 4.7.1.1. We avoid the calculation of the explicit inverse in this formulation in order to preserve sparsity. $L^{-1}$ is in general a dense matrix (see Figure 4.1) and a dense matrix vector multiplication would require more flops than the sparse forward and backward solves. Moreover, storing the inverse uses significantly more memory resources, especially due to the large problem size.
C1. Inner problem

The solution of the inner problem, namely Step 3 of Algorithm 2, consists of the equation:

$$z_{k+1} = H_s^{-1}(A_s^T y_k + f_s), \quad (4.45)$$

where we have neglected the projection step that has inferior cost. Taking into account the structure of matrix $A_s^T$

\[
A_s^T = \begin{bmatrix}
I & -C_0^T & -A_0^T & 0 & 0 & 0 & \ldots \\
0 & -D_0^T & -B_0^T & 0 & 0 & 0 & \ldots \\
0 & I & 0 & 0 & 0 & 0 & \ldots \\
0 & 0 & I & -C_1^T & -A_1^T & 0 & \ldots \\
0 & 0 & 0 & -D_1^T & -B_1^T & 0 & \ldots \\
0 & 0 & 0 & I & 0 & 0 & \ldots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \\
\end{bmatrix},
\]

we derive the following cost for the sparse matrix-vector multiplication

$$2N_{n_x}(n_y + n_x) + 2N_{n_u}(n_y + n_x) = 2N(n_y + n_x)(n_x + n_u) \text{ flops.} \quad (4.47)$$

The vector addition that follows needs trivially $N(n_x + n_y + n_u) + n_x$ flops and the multiplication with the diagonal Hessian another $N(n_x + n_y + n_u) + n_x$ flops. The dominant operation in this step is therefore the matrix-vector multiplication and the overall cost is approximated by Equation (4.47).

C2. Gradient Step

We restate for convenience that the gradient step on the dual variables is

$$\lambda_{k+1} = y_k + L^{-1}(A_s z_{k+1} - b_s). \quad (4.48)$$

Here we can ignore again the cost of vector additions and focus on the matrix-vector multiplications $A_s z_{k+1}$ and $L^{-1} r$, where $r$ is the residual $A_s z_{k+1} - b_s$. Exploiting the sparsity of the first product yields

$$2N_{n_y}(n_x + n_u) + 2N_{n_x}(n_x + n_u) = 2N(n_y + n_x)(n_x + n_u) \text{ flops}, \quad (4.49)$$
which is not surprisingly the same as the cost in (4.47). The second product $L^{-1}r$ can be solved by one forward and one backward substitution. Matrix $R$ has a banded structure, with $N$ blocks.\footnote{As a block we consider here the pair: $R_{i,i} R_{i,i+1}$. The last block (or first for the transpose matrix) that comprises only one term $R_{i,j}$ is neglected.} These blocks consist of $n_x + n_y$ rows and the number of non-zero elements starts from $n_x + 1$ and increases by one in each row (see for instance Figure 4.1). Therefore, the complexity of each block is

$$\sum_{i=1}^{n_x+n_y} (2(n_x+i) - 1) = (n_x + n_y)(3n_x + n_y) \text{ flops} \quad (4.50)$$

and summing for all blocks and the two solves yields a total of

$$2N(n_x + n_y)(3n_x + n_y) \text{ flops.} \quad (4.51)$$

Adding Equations (4.49) and (4.51), results in a total cost of

$$2N(n_x + n_y)(4n_x + n_y + n_u) \text{ flops.} \quad (4.52)$$

C3. Iterate update

The update of the dual vector involves only operations of lower order and therefore it is neglected from all subsequent calculations.

4.7.2 xu* formulation

As already discussed, this formulation is more restrictive with respect to the structure of system matrices $C_k, D_k$ but if applicable, it has smaller dimension than the previously discussed formulation and it should be preferred. One common example is the case where we do not have any equations for the output dynamics in the MPC formulation, while weights and constraints refer only to states and inputs. In the sections that follow, we omit the steps that similarly to the previous formulation do not induce any significant cost.

B1. Calculation of generalized matrix $L$

Matrix $L$ has again a tridiagonal block structure as in (4.40), but this time the blocks are

$$L_{1,1} = \tilde{Q}_0^{-1}$$

$$L_{1,2} = -\tilde{Q}_0^{-1}A_0^T$$

$$L_{k,k} = A_{k-2}\tilde{Q}_{k-2}^{-1}A_{k-2}^T + B_{k-2}R_{k-2}^{-1}B_{k-2}^T + \tilde{Q}_{k-1}$$

$$L_{k,k+1} = -\tilde{Q}_{k-1}^{-1}A_{k}^T \quad \text{for } k = 1, \ldots N$$

$$L_{N+1,N+1} = A_{N-1}\tilde{Q}_{N-1}^{-1}A_{N-1}^T + B_{N-1}R_{N-1}^{-1}B_{N-1}^T + P$$

where $\tilde{Q}_k = C_k^T Q_k C_k$ is by assumption diagonal. The cost for the calculation of the blocks $L_{i,j}$ with $i \neq j$ is again neglected while for the $N$ diagonal blocks we need

$$N n_x^2 (n_x + n_u) \text{ flops.} \quad (4.53)$$

In case dynamics and weights do not vary along the horizon, the dependency on $N$ vanishes.
Remark 4.7.2.1. Calculating matrices $\tilde{Q}_k$ induces no significant cost because matrices $Q_k$ are diagonal and $C_k$ are restricted to have only one non-zero element per row.

Remark 4.7.2.2. In general $n_y \leq n_x$ and based on our assumptions matrices $\tilde{Q}_k$ might often be rank deficient. A regularization constant must be added in the diagonal elements to ensure that the Hessian is positive definite.

B2. Factorization of $L$

Following the same reasoning as in the previous formulation, the sparse Cholesky factorization of matrix $L$ needs now

$$Nn_x^3 + Nn_x^3 + \frac{1}{3}Nn_x^3 = \frac{7}{3}Nn_x^3 \text{ flops.}$$  \hfill (4.54)

The Cholesky factorization of the diagonal matrix $L_{1,1}$ is neglected as it requires only $n_x$ divisions.

C1. Inner problem

The sparse matrix vector multiplication $A^Ty_k$ that dominates this step induces approximately

$$2Nn_x(n_x + n_u) \text{ flops,}$$  \hfill (4.55)

given the structure of the equality constraints.

C2. Gradient step

The multiplication $A_s z_{k+1}$ adds another $2Nn_x(n_x + n_u)$ flops to the flop count. Due to the sparsity of matrices $R$ and $R^T$, one forward or backward substitution requires

$$N \sum_{i=1}^{n_x} (2(n_x + i) - 1) + n_x^2 \approx 3Nn_x^2 \text{ flops.}$$  \hfill (4.56)

Therefore, the cost of the two solves is $6Nn_x^2$ and the total cost of the step

$$2Nn_x(4n_x + n_u) \text{ flops.}$$  \hfill (4.57)

4.7.3 uy formulation

Although condensing the states ruins the structure of the problem, the properties of the discussed algorithm are such, that might favor this formulation against others in certain contexts. For instance, this holds when we do not want to perturb the given optimization problem or when the number of states is large and the horizon short. The paragraphs that follow analyze the complexity when the QP is derived using the uy formulation.

B1. Calculation of generalized matrix $L$

As described in the Appendix, Section A.3, the matrix of equality constraints is in this case equal to

$$A_s = \begin{bmatrix} I & -B \end{bmatrix},$$  \hfill (4.58)
where we remind that $\bar{B} = CB + \mathcal{D}$. Deriving $L$ using (4.58) yields:

$$L = A_s H_s^{-1} A_s^T = \begin{bmatrix} I & -\bar{B} \end{bmatrix} \begin{bmatrix} \bar{B}^{-1} \\ -\bar{B}^T \end{bmatrix} = \left[ H_s^{-1} + \bar{B} \bar{H}_s^{-1} \bar{B}^T \right].$$ (4.59)

Since the Hessian matrix is diagonal, the complexity of this step is mainly affected by the multiplication of $\bar{B}$ with its transpose. The cost of this product is already calculated in equations (4.27) and (4.28).

**B2. Factorization of $L$**

Matrix $L$ is in general dense when states are eliminated. Therefore, for the Cholesky factorization we have to consider approximately

$$\frac{1}{3} (N n_y + n_x)^3 \text{ flops.}$$ (4.60)

The advantage is that in this formulation the matrix has a much smaller dimension and therefore its cubic power does not necessarily yield a prohibitive complexity.

Calculating the inverse of $L$ is usually advantageous in this approach since the Cholesky factor will be anyway dense. This calculation increases the cost to $\frac{4}{3} (N n_y + n_x)^3$ flops according to Equation (4.13).

**C1. Inner problem**

For the inner problem step, we consider again the matrix-vector multiplication

$$A_s^T y_k = \begin{bmatrix} I \\ -\bar{B}^T \end{bmatrix} y_k.$$ (4.61)

Since the identity block does not induce any flops, we restrict our attention to the multiplication $\bar{B}^T y_k$ which according to Equation (4.30) requires

$$N^2 n_y n_u + 2N n_x n_u \text{ flops.}$$ (4.62)

**C2. Gradient step**

Here, the matrix-vector multiplication $A_s z_{k+1}$ is reduced again to $\bar{B} U_{k+1}$, where $U_{k+1}$ are the input variables included in $z_{k+1}$. The cost similarly to the previous step is

$$N^2 n_y n_u + 2N n_x n_u \text{ flops.}$$ (4.63)

Solving the multiplication with $L^{-1}$ with a forward and backward substitution adds another

$$2(N n_y + n_x)^2 \text{ flops}$$ (4.64)

to the calculations since matrix $\mathcal{R}$ is in general dense. Using the explicit expression for the inverse needs the same amount of flops but it can be efficiently parallelized, in contrary to the two substitutions. Alternatively, one could think of merging the two operations by pre-calculating the quantities $m_1 = L^{-1} A_s$, $m_2 = L^{-1} b_s$ in the initialization part and writing the gradient step as

$$\lambda_{k+1} = y_k + m_1 z_{k+1} - m_2.$$ (4.65)
Condensing (d) 0 2N \left( \frac{N}{2} \right)^c d n_y n_x (n_x + n_u) + 2N n_x^2 (n_x + n_u) \\
Calculation of L (d,w) N^c \omega (n_y + n_x)^2 (n_x + n_u) \frac{1}{3} N^3 n_y^2 n_u + N^2 n_x n_u n_y + N n_x^2 n_u \\
Factorization of L (d,w) N (n_y + n_x) n_x^2 + N n_x (n_y + n_x)^2 + \frac{1}{3} N (n_y + n_x)^3 \\
\frac{1}{3} c_d, c_w \text{ as defined in } (4.15)

<table>
<thead>
<tr>
<th>Inner problem</th>
<th>xuy formulation</th>
<th>uy formulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2N (n_y + n_x) (n_x + n_u)</td>
<td>N^2 n_u n_y + 2N n_x n_u</td>
<td></td>
</tr>
<tr>
<td>Gradient step</td>
<td>2N (n_y + n_x) (4n_x + n_u + n_y)</td>
<td>N^2 n_u n_y + 2n_x^2 + 2N n_x (n_u + 2n_y)</td>
</tr>
<tr>
<td>Total</td>
<td>2N (n_y + n_x) (5n_x + 2n_u + n_y)</td>
<td>2N^2 (n_y^2 + n_u n_y) + 2n_x^2 + 4N n_x (n_u + n_y)</td>
</tr>
</tbody>
</table>

Table 4.4: Problem transformation and initialization complexity of Algorithm 2.

Table 4.5: Complexity per iteration of Algorithm 2.

However, since \( m_1 \in \mathbb{R}^{(N n_y + n_x) \times (N (n_y + n_u) + n_x)} \), the new matrix-vector product would require approximately

\[
2(N n_y + n_x)(N (n_y + n_u) + n_x) = 2(N n_y + n_x)^2 + 2N^2 n_y n_u + 2N n_x n_u \text{ flops}, \quad (4.66)
\]

which is clearly higher than the sum of Equations (4.63) and (4.64).

### 4.7.4 Conclusions

In this section we summarize the computational complexity of the algorithm for all permissible formulations and comment on the results. Table 4.4 summarizes the complexity of the transformation and initialization parts. Similarly to Section 4.5.5, the parenthesis next to each step indicates which problem quantities affect it. If all of them are constant, the complexity of the step is zero. Table 4.5 on the other hand presents the cost per iteration that does not depend on the time-varying nature of the problem data.

As a general remark, we observe that the dependence on the horizon for the xuy and xu* formulation is linear while for the uy formulation it is cubic in the initialization and square in the iterative part. Therefore, we can infer that the latter formulation may only be preferable in problems with short horizons. Moreover, in absence of terminal state constraints and weight, the complexity of the uy formulation is affected by the number of states only in the condensing and the incorporation of initial condition. The other two formulations have a quadratic dependence on n_x and therefore they are expected to be more computationally expensive in problems with large number of states. We also observe that in all formulations, the number of outputs seem to affect more drastically the complexity than the number of inputs.
In order to illustrate the dependency of the two formulations on the problem dimensions, we plot the total number of flops as a function of the horizon length and look how the complexity scales when one of the fundamental dimensions is doubled. The results for a time-varying example with ten states, two inputs and four outputs are shown in Figures 4.2, 4.3, 4.4. In the calculation of the total cost, we have assumed that the algorithm runs for one hundred iterations. As the curves suggest, increasing the number of states has a higher impact on the x uy formulation while the number of outputs affects the complexity of the uy formulation more. Both formulations are not severely affected when the number of inputs increases.

Finally, in order to demonstrate how the choice of the most efficient formulation depends also on the number of iterations, we plot the complexity curves in the three dimensional space. The results in Figures 4.5 and 4.6 show that although the horizon length above which the x uy outperforms the uy formulation is varying with the number of iterations, the crossing quickly settles to an horizon length of twelve for the specific example. This behavior is due to the fact that above a certain number of iterations, that is problem-dependent, the impact of the initialization and problem transformation cost wears off and the total complexity is dominated by the cost per iteration.

Remark 4.7.4.1. The xu formulation is not listed on Tables 4.4 and 4.5 as it can be recovered by setting ny equal to zero in the x uy formulation.

4.8 Complexity of Bemporad’s Dual FGM

The second variant of the dual fast gradient method (aka GPAD), that relaxes the inequality constraints instead of the equalities, is not restricted to diagonal Hessian matrices which was the main motivation for keeping all states and outputs as optimization variables or condensing only the states. Therefore, we only analyze the xu and u formulation. Since the dual multipliers of the inequality constraints are the same in both cases, the employed formulation only affects the way the inner problem is solved. This inner problem is in general an equality-constrained problem, as all inequalities are relaxed. In the
Figure 4.3: Scaling of complexity when number of inputs is doubled.

Figure 4.4: Scaling of complexity when number of outputs is doubled.
Figure 4.5: Total complexity for an example with 6 states, 2 inputs and 4 outputs.

Figure 4.6: Total complexity from a different view.
condensed case we solve it by eliminating the equalities and setting the derivative of the unconstrained problem to zero, while in the sparse we factorize the KKT matrix and determine the optimal input-state trajectory of each iteration with one forward and one backward solve. For that reason, the initialization part differs substantially for the two formulations.

As pointed out in Remark 3.3.2.3, the calculation of the scalar Lipschitz constant or the generalized matrix is independent of the formulation used to solve the inner problem. Therefore, we dedicate a separate subsection for the complexity of this step. We remind also that an online estimation of \( L \) is only possible in the scalar case, since for the generalized variant an SDP must be solved to determine the step size for each direction.

**Remark 4.8.1.** Although the method is able to deal with non-diagonal weight matrices \( Q_k, R_k \) and non-zero coupling terms \( S_k \), we keep assumption of Section 4.3. On the one hand this setup is very common in practice and on the other, it makes the comparison between the algorithms more meaningful.

### 4.8.1 xu formulation

The paragraphs that follow analyze the complexity of the sparse GPAD variant for both initialization and iterative parts. Since in our problem formulation (2.3) we include the output dynamics that are subsequently condensed, we define new weight and constraint matrices before proceeding with the analysis. More precisely, for the weights we have:

\[
\tilde{Q}_k = C_k^T Q_k C_k \quad (4.67) \\
\tilde{R}_k = R_k + D_k^T Q_k D_k \quad (4.68) \\
\tilde{S}_k = C_k^T Q_k D_k \quad (4.69) \\
\tilde{q}_k = C_k^T (2Q_k e_k - 2Q_k y_r k + q_k) \quad (4.70) \\
\tilde{r}_k = D_k^T (2Q_k e_k - 2Q_k y_r k + q_k) - 2R_k u_r k + r_k, \quad (4.71)
\]

similarly to Equations (A.7) and (A.8) of the Appendix while the polyhedral inequality constraints become\(^\text{11}\)

\[
\tilde{M}_k x_k + \tilde{N}_k u_k \leq \tilde{d}_k, \quad (4.72)
\]

with matrices \( \tilde{M}_k \in \mathbb{R}^{\tilde{m}_s \times n_x}, \tilde{N}_k \in \mathbb{R}^{\tilde{m}_s \times n_u} \) and vector \( \tilde{d}_k \in \mathbb{R}^{\tilde{m}_s} \) defined as follows:

\[
\tilde{M}_k = \begin{bmatrix} M_k C_k \\ -M_k C_k' \\ C_k \\ -C_k \\ 0 \\ 0 \end{bmatrix}, \quad \tilde{N}_k = \begin{bmatrix} M_k D_k + N_k \\ -M_k D_k - N_k \\ D_k \\ -D_k \\ I \\ -I \end{bmatrix}, \quad \tilde{d}_k = \begin{bmatrix} d^u k - M_k e_k \\ -d^l k + M_k e_k \\ y^u k - e_k \\ -y^l k + e_k \\ u^u k \\ -u^l k \end{bmatrix}. \quad (4.73)
\]

Finally, the polyhedral constraint of the terminal state is simply

\[
\tilde{T} x_N \leq \tilde{d}_N, \quad (4.74)
\]

\(^{11}\)The tilde “\( \tilde{\} \)” is introduced to stress the fact that simple and polyhedral inequality constraints with upper and lower bounds are stacked together in a single inequality with an upper bound. The same notation was used for instance in the transformation of (2.7) to (2.8).
with \( \tilde{T} \in \mathbb{R}^{\hat{m}_t \times n_x}, \tilde{d}_N \in \mathbb{R}^{\hat{m}_t} \) and

\[
\tilde{T} = \begin{bmatrix} T & -T \end{bmatrix}, \quad \tilde{d}_N = \begin{bmatrix} d^u_N \\ -d^l_N \end{bmatrix}.
\]  

(4.75)

B1. Factorization of KKT matrix

The algorithm for the factorization of the KKT matrix, presented in [36], is quoted here in Algorithm 6 with minor changes to match our notation and problem formulation. Taking

Algorithm 6 GPAD: Factorization step

1. Initialize \( P_N = Q_N \)

2. for \( k = N - 1 : 0 \) do

3. \( \bar{R}_k = \tilde{R}_k + B_k^T P_{k+1} B_k, \quad \bar{S}_k = \tilde{S}_k + A_k^T P_{k+1} B_k \)

4. \( P_k = \tilde{Q}_k + A_k^T P_{k+1} A_k - \tilde{S}_k \tilde{R}_k^{-1} \bar{S}_k^T \)

5. end for

6. for \( k = 0 : N - 1 \) do

7. \( K_k = -\tilde{R}_k^{-1} \bar{S}_k^T, \quad D_k = -\tilde{R}_k^{-1} \bar{N}_k^T \)

8. \( M_k = -\tilde{R}_k^{-1} B_k^T, \quad d_k = -\tilde{R}_k^{-1} (\tilde{r}_k + B_k^T P_{k+1} f_k) \)

9. \( L_k = (A_k + B_k K_k)^T, \quad C_k = (\tilde{M}_k + \tilde{N}_k K_k)^T \)

10. \( s_k = K_k^T \tilde{r}_k + L_k P_{k+1} f_k + \tilde{q}_k \)

11. end for

into account only arithmetic operations of cubic order (namely matrix-matrix products, factorizations, matrix forward-backward substitutions), the reported complexity of the factorization is

\[
N(3n_x^3 + 6n_u^2 n_u + 6n_u^2 n_x + \frac{1}{3} n_u^3 + 2\hat{m}_s n_x n_u + 2\hat{m}_s n_u^2) \text{ flops.}
\]  

(4.76)

According to the definition of the matrices in (4.73), the term \( \hat{m}_s \) above is equal to \( 2m_s + 2n_y + 2n_u \). However, due to the special structure of \( \hat{M}_k \) and \( \hat{N}_k \), we do not need to perform the whole matrix-matrix multiplication. This enables us to substitute \( \hat{m}_s \) with just \( m_s + n_y \) in Equation (4.76).

Finally, note that the factorization terms \( C_k, M_k, d_k \) should not be confused with the homonymous terms in the system dynamics and polyhedral constraints of (2.3).

Remark 4.8.1.1. Algorithm 6 has the same complexity independently of whether the data are time-varying or problem-varying. However, if both dynamics and weights are constant the complexity reduces to \( 2N\hat{m}_s n_u (n_x + n_u) \) and if the constraints \( M_k, \hat{N}_k \) are also constant the factorization can be performed offline and the online cost becomes zero.
C1. Inner problem

The inner problem after the described factorization is solved via Algorithm 7 and the cost of the operation reported in [36] is

\[ N(4n_x^2 + 6n_xn_u + 2\tilde{m}_s(n_x + n_u)) + 2\tilde{m}_t n_x \text{ flops.} \] (4.77)

In contrary to the case of the factorization, the special structure of the matrices \( \tilde{M}_k, \tilde{N}_k \), that is inherited to matrices \( C_k, D_k \) cannot be exploited here, forcing us to substitute \( \tilde{m}_s \) with the expression \( 2m_s + 2n_y + 2n_u \). For the same reason, \( \tilde{m}_t \) is equal to \( 2m_t \).

The cost of Equation (4.77) can be further reduced if the problem has additional structure (e.g. only box constraints). Note that the term \( y_k \) here is the dual iterate of the constraints at step \( k \) and it should not be confused with the output variables that are anyway eliminated from the formulation.

\textbf{Algorithm 7} GPAD: Backward and forward solve step

1: Initialize \( e_N = T^T y_N \)
2: for \( k = N - 1 : 1 \) do
3: \( e_k = L_k e_{k+1} + C_k y_k + s_k \)
4: end for
5: \( x_0 = x \)
6: for \( k = 0 : N - 1 \) do
7: \( u_k = K_k x_k + D_k y_k + M_k e_{k+1} + d_k \)
8: \( x_{k+1} = A_k x_k + B_k u_k + f_k \)
9: end for

C2. Gradient step

The cost of the dual update equation, that can be written as\(^{\ref{footnote12}}\)

\[ \mu_{k+1} = y_k + L^{-1} \left( \hat{C}_s z_{k+1} - \hat{d}_s \right), \] (4.78)

is dominated by the matrix-vector multiplication \( \hat{C}_s z_{k+1} \). \( L \) is either scalar or a diagonal matrix and therefore the matrix-vector multiplication with \( L^{-1} \) requires as many flops as one vector addition. \( \hat{C}_s \) has the following sparsity structure

\[ \hat{C}_s = \begin{bmatrix} \hat{M}_0 & \hat{N}_0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \hat{M}_1 & \hat{N}_1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \hat{T} \end{bmatrix}, \] (4.79)

\(^{12}\)The projection on the positive orthant is neglected as it is computationally cheaper than the other operations of the step.
with $\tilde{M}_k, \tilde{N}_k$ and $\tilde{T}$ defined in (4.73) and (4.75). The multiplication requires
\[
2Nn_x(m_s + n_y) + 2Nn_a(m_s + n_y) + 2m tn_x = 2N(m_s + n_y)(n_x + n_u) + 2m tn_x \text{ flops. (4.80)}
\]
However, there is often additional structure that can lower this cost significantly. For instance, in absence of polyhedral constraints, the cost is only $2Nn_y(n_x + n_u)$ flops and if in addition $y = x$, the product requires no flops at all and the complexity of the current step is dominated by Step C1.

**C3. Iterate update**

Similarly to the previous algorithm, this step induces no considerable cost and it is therefore neglected in our analysis for both considered formulations.

### 4.8.2 $u$ formulation

When we choose to condense outputs and states, the inner problem is unconstrained and has a simple analytical solution. The paragraphs that follow, analyze the complexity of initialization and iterative part for the condensed GPAD variant.

**B1. Inversion of $H_c$**

In the condensed formulation, $H_c \in \mathbb{S}^{Nn_u}_{++}$ is in general a dense matrix. If either the dynamics or the weights are changing between QP instances, we need to calculate the inverse of this matrix online. Assuming that we perform the inversion via Cholesky factorization, the complexity of this step is
\[
\frac{4}{3}(Nn_u)^3 \text{ flops, (4.81)}
\]

**Remark 4.8.2.1.** Alternatively, we can avoid calculating the inverse and only store the Cholesky factor. This would require half the memory since $R$ is triangular but the multiplication with $H_c^{-1}$ will not allow for an efficient parallel implementation. The right choice depends on the available hardware, although in most cases the explicit inverse should yield superior performance.

**C1. Inner problem**

We remind, that in order to solve the inner problem, we clip the unconstrained solution
\[
z_{k+1} = -H_c^{-1}(f_c + \tilde{C}_c^T y_k)
\]
(4.82)
to the positive orthant. Therefore, to calculate the complexity of this step we need to determine the number of flops for the two matrix-vector products in the above expression. According to Equations (A.34) and (A.35), the first product can be re-written as
\[
\tilde{C}_c^T y_k = \bar{B}^T y_1 - \bar{B}^T y_2 - \bar{M}^T y_3 - \bar{M}^T y_4 + y_5 - y_6,
\]
(4.83)
where $y_k = [y_1^T \ y_2^T \cdots \ y_6^T]^T$ and $\bar{M} = \mathcal{M}\bar{S} + \mathcal{N}$. Ignoring the vector additions, the first two terms require together $2N^2 n_y n_u + 4N n_x n_u$ flops while the next two require similarly
Finally, a multiplication with the dense Hessian needs another \(2(Nn.u)^2\) operations, yielding a total of

\[
2N^2 n_u + 2N^2 n_u (n_y + m) + 4Nn_u (n_x + m_t) \text{ flops.}
\]  

(4.84)

When the terminal state is not part of the optimization variables, the third term in the expression above vanishes.

**C2. Gradient step**

Finally, for the gradient step in the dual domain, the dominant operation is the multiplication \(\bar{C}_c.z_{k+1}\), or

\[
\begin{bmatrix}
\bar{B} \\
-\bar{B} \\
\bar{M} \\
-\bar{M} \\
I \\
-I
\end{bmatrix}z_{k+1}.
\]  

(4.85)

Due to the special structure of \(\bar{C}_c\) we only need to calculate the products \(\bar{B}z_{k+1}\) and \(\bar{M}z_{k+1}\) which require in total

\[
N^2 n_u (n_y + m) + 2Nn_u (n_x + m_t) \text{ flops.}
\]  

(4.86)

**4.8.3 Lipschitz constants and generalized matrices**

As already described in Sections 3.3.2 and 3.3.3, there are various choices for the scalar Lipschitz constant or the generalized matrix, whose inverse is used as a step in the gradient direction. In the paragraphs that follow, we revisit the available options and establish a common notation.

**Scalar \(L\).** As far as the scalar values of \(L\) are concerned, Equation (3.49) is always at least as tight as (3.48) and therefore the complexity of the latter is skipped in our analysis. Let us define here this first choice of scalar Lipschitz constant as

\[
L_{S1} = ||\tilde{C}_sH_s^{-1}\tilde{C}_s^T||.
\]  

(4.87)

Note that in the xu formulation, matrix \(\tilde{C}_s\) depends on matrices \(\tilde{M}_k, \tilde{N}_k\) which in turn depend on the system matrices \(C_k\) and \(D_k\) (see Equations (4.73) and (4.79)). However, in problems where the dynamics are not constant, we may want to avoid this dependence in order to calculate the step size offline. Since according to Remark 3.3.2.3, inner problem and Lipschitz constant can be calculated using different QP formulations, the data from the uy formulation can be employed to compute the value of \(L_{S1}\).

Equations (3.56) and (3.61) calculate a tighter Lipschitz constant since they are both equal to the spectral norm of the Hessian of the dual, but for different QP formulations. As the employed QP formulation only affects the inner problem and not the dual, the two expressions are equivalent. We refer to this choice of Lipschitz constant as

\[
L_{S2} = ||C_{K11}\tilde{C}_s^T|| = ||\tilde{C}_cH_c^{-1}\tilde{C}_c^T||.
\]  

(4.88)
Note that Equation (4.88) does not require the inversion of $H_s$ and therefore the weights on the outputs do not have to be strictly positive definite as in (4.87). However, the dependence on the dynamics might increase the complexity unnecessarily in time-varying problems where $L_{S1}$ is sufficiently tight and can be calculated offline.

**Remark 4.8.3.1.** For both scalar cases there is no index assigned in the matrix norm. In general, the spectral norm yields the tightest constant but if the step size has to be calculated online, the infinity or Frobenius norm can be also used to upper-bound this value. Recall that since the matrix is symmetric, the 1-norm coincides with the infinity norm.

**Matrix $L$.** When we want to use the generalized variant of GPAD, that yields in general faster convergence without a significant overhead in the complexity per iteration, an SDP must be solved to determine the diagonal matrix $L$. Clearly such an operation is too expensive to be transferred online. In a context where all problem quantities are constant, a matrix $L$ that satisfies condition (3.63) should be chosen which is expected to yield the largest possible step size. In accordance with the previous notation, we call this matrix $L_{M2}$ and it must satisfy the following condition in the sparse or condensed formulation

$$L_{M2} \succeq \tilde{C}_s K_{11} \tilde{C}_s^T \quad \text{or} \quad L_{M2} \succeq \tilde{C}_s H_s^{-1} \tilde{C}_s^T. \quad (4.89)$$

If however the dynamics are time-varying while objective and constraints are constant, the condition (3.62) should be used instead as a constraint in the SDP. The matrices $\tilde{C}_s$ and $H_s$ should be taken again from the uy formulation, to avoid the dependence on the dynamics. Attention must be payed however to keep the order of the constraints between different formulations consistent, so that the computed step sizes are assigned to the proper directions. We define this last option for the generalized matrix as

$$L_{M1} \succeq \tilde{C}_s H_s^{-1} \tilde{C}_s^T. \quad (4.90)$$

Note that the property of GPAD to allow for different formulations in the calculation of $L$ and in the solution of the inner problem, as highlighted in Remark 3.3.2.3, enables the use of the generalized variant in a time-varying setup that appears often in practice.

In Table 4.6 we report the permissible $L$, scalars or matrices, for all possible combinations of varying dynamics, weights and constraints. Note that if any of the last two is not constant, the generalized variant can not be used.

**Remark 4.8.3.2.** In a problem setup where the system dynamics are varying but bounded, a robust value for $L_{M2}$ can be calculated that satisfies Equation (4.89) for any set of dynamics. This modification may improve the convergence of the algorithm significantly compared to an implementation that uses the matrix $L_{M1}$, as we show in Section 6.4.

**Online complexity of $L_{S1}$**

The product $\tilde{C}_s H_s^{-1} \tilde{C}_s^T$ using the matrices of the uy formulation and the ordering of the xu formulation, as indicated in (4.73), has the simple block diagonal structure
\begin{table}[h]
\centering
\begin{tabular}{cccc}
\hline
vd & vw & vc & Permissible L & Comments \\
\hline
0 & 0 & 0 & $L_{M1}^-, L_{M2}^-, L_{S1}^-, L_{S2}^-$ & $L_{M2}$ should be preferred in a linear time-invariant case. \\
0 & 0 & 1 & $L_{S1}^+, L_{S2}^+$ & $L_{S2}^+$ is typically tighter but $L_{S1}^+$ is cheaper. \\
0 & 1 & 0 & $L_{S1}^+, L_{S2}^+$ & $L_{M1}^-$ can be used here using the matrices from the uy formulation that are independent of the dynamics. Whether the convergence is better than using $L_{S2}^+$ depends on the given problem. \\
0 & 1 & 1 & $L_{S1}^+, L_{S2}^+$ & $L_{S2}^+$ is typically tighter but $L_{S1}^+$ is cheaper. \\
1 & 0 & 0 & $L_{M1}^-, L_{S1}^-, L_{S2}^+$ & $L_{S2}^-$ can be calculated offline \\
1 & 0 & 1 & $L_{S1}^+, L_{S2}^+$ & $L_{S1}^+$ must be calculated online \\
1 & 1 & 0 & $L_{S1}^+, L_{S2}^+$ & $L_{S2}^+$ is typically tighter but $L_{S1}^+$ is cheaper. \\
1 & 1 & 1 & $L_{S1}^+, L_{S2}^+$ & \\
\hline
\end{tabular}
\caption{Permissible $L$ depending on time-varying nature of problem data.}
\end{table}

Moreover, the blocks for $k = 0, \ldots, N - 1$ have the following pattern

$$
\tilde{L}_k = \begin{bmatrix}
\tilde{L}_{k,1} & -\tilde{L}_{k,1} & -\tilde{L}_{k,2} & -\tilde{L}_{k,3} & -\tilde{L}_{k,3} \\
-\tilde{L}_{k,1} & \tilde{L}_{k,1} & -\tilde{L}_{k,2} & -\tilde{L}_{k,3} & -\tilde{L}_{k,3} \\
\tilde{L}_{k,2} & -\tilde{L}_{k,2} & Q_k^{-1} & -Q_k^{-1} & 0 \\
-\tilde{L}_{k,2} & \tilde{L}_{k,2} & -Q_k^{-1} & Q_k^{-1} & 0 \\
\tilde{L}_{k,3} & -\tilde{L}_{k,3} & 0 & 0 & R_k^{-1} \\
-\tilde{L}_{k,3} & \tilde{L}_{k,3} & 0 & 0 & -R_k^{-1}
\end{bmatrix},
$$

where the sub-blocks of $\tilde{L}_k$ are in turn:

$$
\tilde{L}_{k,1} = M_k Q_k^{-1} M_k^T + N_k R_k^{-1} R_k^T \\
\tilde{L}_{k,2} = M_k Q_k^{-1} \\
\tilde{L}_{k,3} = N_k R_k^{-1}
$$

The cost of each block $\tilde{L}_k$ is then only $m_s^2(n_y + n_u)$ flops, taking into account that the
weight matrices are diagonal. The total cost for the whole matrix $\tilde{C}_sH_s^{-1}C_s^T$ is

$$Nm_s^2(n_y + n_u) + m_t^2n_x \text{ flops},$$  \hspace{1cm} (4.93)

where the second term is due to the last block $\tilde{L}_N$ that contains only the product $TQ_N^{-1}TT$. Finally, the calculation of the infinity norm\textsuperscript{13} requires only $Nm_s(m_s + n_y + n_u) + m_t^2$ flops and therefore it can be safely neglected.

When weights and constraints are problem-varying, all $\tilde{L}_k$ blocks are identical. In that case, the complexities of the multiplication and infinity norm are not multiplied by the horizon length $N$.

**Online complexity of $L_{S2}$**

In cases where objective or constraints are changing, one of the scalar Lipschitz constants needs to be calculated online. Although $L_{S1}$ induces lower complexity, it is less tight than $L_{S2}$ and therefore the right choice is problem dependent. Depending on which formulation is used to solve the inner problem in GPAD, the corresponding equation for $L_{S2}$ must be employed to avoid unnecessary calculations.

**Sparse formulation.** In the sparse variant, $L_{S2}$ is the norm of the product $\tilde{C}_sK_{11}\tilde{C}_s^T$. To compute the complexity of this operation, we will take into account the sparsity of the matrix $\tilde{C}_s$ (Equation (4.79)) as well as the symmetry of the final result. Multiplying first $\tilde{C}_s$ with the dense matrix $K_{11} \in S_N^{N(n_x+n_u)+n_x}$ needs

$$(2N(n_x + n_u)(m_s + n_y) + 2n_xm_t)(N(n_x + n_u) + n_x) \text{ flops},$$ \hspace{1cm} (4.94)

using the already known complexity of the matrix-vector product in (4.80). This result is independent of whether the blocks $\tilde{C}_k$ are identical or not. Note that the products within the blocks of the inequality constraint matrix are already calculated in a previous step. The second multiplication of the dense result with $\tilde{C}_s^T$, taking again into account its sparsity and special structure while calculating the result row-wise is

$$(2N(n_x + n_u)(m_s + n_y) + 2n_xm_t)(N(m_s + n_y + n_u) + m_t) \text{ flops},$$ \hspace{1cm} (4.95)

where we have divided the result by two due to the symmetry of the result. Finally, calculating the infinity norm of the matrix raises the complexity by

$$4N^2(m_s + n_y + n_u)^2 + 4m_t^2 \text{ flops},$$ \hspace{1cm} (4.96)

assuming that the max operator does not induce any cost.

**Remark 4.8.3.3.** The matrix $K_{11}$ is the upper block of the inverted KKT matrix. Therefore, using $L_{S2}$ requires the explicit inversion and not just the factorization of the matrix.

**Condensed formulation.** When both states and outputs are condensed, the matrices $\tilde{C}_c$ and $H_c$ do not have any structure that we can exploit. Note that $\tilde{C}_c$ has the same number of rows as $\tilde{C}_s$ but fewer columns. The first product yields now

$$4(N(m_s + n_y + n_u) + m_t)N^2n_x^2 \text{ flops},$$ \hspace{1cm} (4.97)

\textsuperscript{13}As highlighted in Remark 4.8.3.1, the infinity norm is used instead of the 2-norm as it is less computationally expensive.
Table 4.7: Problem transformation and initialization complexity of Algorithm 3.

while the second

\[4(N(m_s + n_y + n_u) + m_t)^2 N n_u \text{ flops.} \quad (4.98)\]

The complexity for calculating the norm of the result is equal to the one reported in (4.96) since it is the same matrix.

4.8.4 Conclusions

A complexity summary of all the problem transformation, initialization and iteration steps is presented in Tables 4.7 and 4.8, assuming that \( L_{S1} \) is used in both formulations for simplicity. Once again, the parenthesis next to the step indicates the variables that affect its complexity. Note that the Lipschitz constant also depends on the dynamics (d) when \( L_{S2} \) is used instead of \( L_{S1} \). Note also that the factorization of the KKT matrix might depend on the constraints (c) as well, but in case only the constraints are varying the complexity of the step comprises only the last term of the expression reported in the table.\(^{14}\)

From the expressions we can infer that the condensed formulation might outperform the sparse one in problems with short horizons, large number of states and small number of inputs. This is also depicted in Figures 4.7, 4.8, 4.9 where it is clear that increasing the number of states severely affects the sparse formulation, increasing the number of inputs has the same effect on the condensed formulation and finally the number of outputs has a smaller impact on both. If the horizon is long and the number of states is not prohibitively large, the sparse formulation is the better choice since the complexity scales linearly with \( N \). As a final remark, we remind that the condensed formulation is more sensitive to numerical errors in problems with unstable plant models.

\(^{14}\)We remind that in all other cases, when one of the variables is varying, the whole expression must be taken into account when estimating the complexity.
### Table 4.8: Complexity per iteration of Algorithm 3.

<table>
<thead>
<tr>
<th></th>
<th><strong>xu formulation</strong></th>
<th><strong>u formulation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inner problem</td>
<td>$N(4n_x^2 + 6n_x n_u) + 4N(m_s + n_y + n_u)(n_x + n_u)$</td>
<td>$2N^2 n_u^2 + 2N^2 n_u(n_y + m_s)$</td>
</tr>
<tr>
<td></td>
<td>$+ 2m_t n_x$</td>
<td>$+ 4N n_u(n_x + m_t)$</td>
</tr>
<tr>
<td>Gradient step</td>
<td>$2N(m_s + n_y)(n_x + n_u)$</td>
<td>$N^2 n_u(n_y + m_s)$</td>
</tr>
<tr>
<td></td>
<td>$+ 2m_t n_x$</td>
<td>$+ 2N n_u(n_x + m_t)$</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>$N(4n_x^2 + 6n_x n_u) + 6N(m_s + n_y)(n_x + n_u)$</td>
<td>$N^2 n_u(2n_u + 3n_y + 3m_s)$</td>
</tr>
<tr>
<td></td>
<td>$+ 4N n_u(n_x + n_u) + 4m_t n_x$</td>
<td>$+ 6N n_u(n_x + m_t)$</td>
</tr>
</tbody>
</table>

Figure 4.7: Scaling of complexity when number of states is doubled.

Figure 4.8: Scaling of complexity when number of inputs is doubled.
Remark 4.8.4.1. The xu⋆ formulation is again not listed in the tables as it can be recovered from the standard xu formulation. In Table 4.7, problem transformation and calculation of LS1 have zero online cost due to the assumed special structure. In the factorization of the KKT matrix, only the first four terms have to be taken into account as ms = 0, Dk = 0 and Ck is very sparse. Therefore, the multiplication with matrices Mk and  ̃Nk in Algorithm 6 has negligible cost. Finally, for the cost per iteration in Table 4.8, it suffices to set ms = 0 in the complexity of the inner problem and neglect the cost of the gradient step that is inferior due to the sparsity of the xu⋆ formulation.
<table>
<thead>
<tr>
<th>Varying constraints</th>
<th>no</th>
<th>yes</th>
<th>no</th>
<th>yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Varying objective</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Varying dynamics</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Varying constraints</th>
<th>Varying objective</th>
<th>Varying dynamics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Richter</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Bemporad</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Gisselson-Richter</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Gisselson-Bemporad</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

Calculate $L = A_s H_s^{-1} A_s^T$ and norm $\|L\|_k$ where $k \in \mathcal{K} = \{1, 2, \infty, F\}$.

2. Calculate $L = \tilde{C}_s H_s^{-1} \tilde{C}_s^T$ and norm $\|L\|_k$ where $k \in \mathcal{K} = \{1, 2, \infty, F\}$.

3. Calculate inverse $H_c^{-1}$ or factorize KKT matrix.

4. Calculate $L = A_s H_s^{-1} A_s^T$ and inverse $L^{-1}$ (or factorize $L = R^T R$).

5. Solve an SDP.

Table 4.9: Initialization complexity of FGMs for different combinations of time-varying data.

### 4.9 Summary table

As a tool to quickly assess the suitability of fast gradient methods in a time-varying context, we offer Table 4.9. Computationally valid combinations are marked with ✓ and subscripts indicate the initialization steps that need to be performed online in each case.

The exponent † highlights methods that are expected to have slow convergence when polyhedral constraints are present while subscript ☆ indicates that the selected combination is not even feasible in such a case.

**Remark 4.9.0.2.** The operations of subscript 2, may also refer to the tighter Lipschitz constant $L_{S2}$. In that case, the quantities $H_c^{-1}$ or $K_{11}$ must be calculated first, which are partially calculated in the operations of subscript 3.
Chapter 5

Benchmarking of First-Order Solvers

In this chapter, we test the discussed solvers on numerical examples in order to highlight some of their important properties. As a tool for the simulations, a general MPC benchmarking suite developed within ABB Corporate Research is employed [26, 27]. The software has been significantly enhanced in the course of this Thesis to meet the needs of the designed experiments.

Section 5.1 motivates the development of the simulation tool and discusses its main features. Section 5.2 studies the behavior of first-order solvers on a severely ill-conditioned problem with simple input and output constraints. The example demonstrates that the generalized variants of dual fast gradient methods yield often superior performance in problems involving matrices with high condition number. Section 5.3 discusses the negative effect of polyhedral inequality constraints on standard and generalized variants of Richter’s dual fast gradient method. Finally in Section 5.4, we show that the different sparse variants of the same algorithm, that we introduced in Section 2.4, yield comparable performance in terms of number of iterations and therefore the right choice depends mainly on the dimensions of the problem and the available hardware.

5.1 A general MPC benchmarking suite

A major shortcoming of most publications introducing novel or surveying existing MPC algorithms is the lack of sufficiently detailed numerical assessment. In fact, it is rather typical to illustrate solver performance on just one or two academic examples. While this is perfectly justified when discussing the theoretical properties of a method, it tells little about its practical usefulness.

Often in the industry, practitioners might face either of the two following challenges. On the one hand, one may need to tackle a specific MPC application and has to assess the performance of many different algorithms on this particular problem, preferably exploiting problem-specific structure as much as possible. On the other hand, one may want to employ MPC for a range of problems and need to assess whether there is a single approach that can satisfy performance requirements on all of these problems or whether different problem classes require different solvers. In order to provide a proper assessment environment for both cases, a benchmarking suite needs to be designed in a general way such that it accommodates the use of many different solvers and at the same time offers a large variety of problems. This section introduces a general MATLAB-based benchmarking suite that is currently being developed within ABB Corporate Research.
Collection of problems. The benchmarking suite aims at collecting a large set of MPC problems arising from different application domains and with a great variety of properties. All problems are described in a common way by an appropriate MATLAB data structure, whose fields contain the possibly time-varying quantities of Problem (2.3). The MPC formulation is complemented by a control scenario, e.g., a sequence of initial states for an open-loop or changes in the reference signals for a closed-loop simulation. The benchmark problems originate from three main sources. Academic examples presented in scientific publications, industrial examples or case studies and randomly generated examples. Although the latter are commonly used for benchmarking purposes, there is evidence that real-world system dynamics typically exhibit much more structure such as integrating behavior or partially independent sub-modules that make them behave significantly different from naively generated random examples. Among the goals of the benchmarking suite is to confirm this evidence via more thorough simulations.

Available solvers. The current version of the benchmarking suite offers a wide variety of solvers either by coupling existing publicly available software or by providing prototype implementations in embedded MATLAB. Existing software includes among others qpOASES [17], an open-source implementation of the online active-set strategy proposed in [16], FORCES [12], a code generation tool for convex multistage problems based on the interior-point method presented in [13], quadprog, the native Quadratic Programming solver of MATLAB and FiOrdOs [40], a code generation tool for the primal and dual fast gradient methods of Sections 3.1 and 3.2. In order to assess the generalized variants of the two aforementioned fast gradient methods, prototype implementations have been also coded. Other implemented methods for the needs of the suite and the Thesis are GPAD, the dual fast gradient method of Section 3.3 as proposed in [4,36], and the two variants of ADMM described in Section 3.4, based on [18,23].

Comparing performance. Comparing different QP solvers in a fair way is not a trivial procedure due to the difference nature of the existing algorithms and the dependence on the application. For instance, there are cases where the solver is given a fixed amount of time to solve an optimization problem and others where an early termination is considered advantageous, either for applying faster feedback or allocating resources to less critical tasks. In an effort to cover as many simulation options as possible, the benchmarking suite currently offers the following stopping criteria:

- **Solver’s own termination condition.** A reliable stopping criterion is for many applications an important solver property. However, comparing results from simulations where each solver terminates with its own stopping criterion is not a trivial task. Typically, the criterion depends on one or more tuning nodes, e.g., tolerances, and the different nature of the criteria renders a direct comparison difficult. A fair way to compare solvers with different stopping criteria is proposed in [14]. The authors suggest to first solve the QP with a default tuning and, if the solution is not accurate enough, repeat the procedure with tighter parameters. This procedure should continue until the desired accuracy is reached, or the solver exceeds the maximum computation time.

- **Fixed number of iterations.** In many applications, a fixed amount of time for solving the optimization problem is given to the solver. In this setup, the computational
burden of checking the termination condition at each iteration (or at every \( n \) iterations) is skipped and the solvers perform a constant number of iterations. For a fair comparison, this number must be adjusted according to the cost-per-iteration of each solver.

- **A priori known optimal solution.** Another common way to compare the efficiency of different solvers is to measure how fast they converge to an optimal solution that is known in advance. There are several options on how to implement such a stopping criterion. For strongly convex QPs, where the solution is unique, a commonly used metric is

\[
\frac{||z_k - z^*||_2}{||z^*||_2} \leq \epsilon, \tag{5.1}
\]

where \( \epsilon \) is a chosen tolerance, \( z_k \) is the vector of primal variables at the \( k \)th iteration and \( z^* \) the optimal solution acquired by a solver with high-accuracy. In order to avoid division by zero and take the absolute error into account instead of the relative when the norm of the optimal vector is smaller than one, we define the function:

\[
\Delta_n(a, b) = \frac{||a - b||_n}{\max(||b||_n, 1)} \tag{5.2}
\]

and use it to implement the termination condition in the benchmarking suite. Equation (5.1) is then transformed to

\[
\Delta_2(z_k, z^*) \leq \epsilon. \tag{5.3}
\]

Furthermore, when we want to compare the performance of solvers on the same problem but using different QP formulations, it is reasonable to use only the input variables in the stopping criterion and not the whole vector \( z \). For this purpose, the benchmarking suite offers the termination condition:

\[
\Delta_2(U_k, U^*) \leq \epsilon, \tag{5.4}
\]

where \( U = [u_0^T u_1^T \ldots u_{N-1}^T]^T \) comprises the input trajectory. Finally, if the problem is not strongly convex, it is preferable to measure the error in the objective value:

\[
\Delta_2(p_k, p^*) \leq \epsilon \tag{5.5}
\]

where \( p = \frac{1}{2} z^T H z + f^T z \), together with a feasibility check on the equality and inequality constraints.

**Simulations.** To conclude the description of the benchmarking suite, a discussion of the simulation options and an illustration of the results follows. As far as the simulation options are concerned, we first distinguish between open and closed-loop simulations. In an open-loop simulation, the MPC data are coupled with fixed initial conditions and the derived QPs are solved independently. In a closed-loop simulation on the other hand, a simulation model generates the initial condition for the next time step while a possibly different model is used than inside the MPC algorithm. In the simplest case, simulation and prediction model are both the same linear system. However, when the focus is more on the actual closed-loop performance rather than on the efficiency of the underlying numerical method, the simulation model may comprise more accurate nonlinear dynamics.
and the prediction model can be calculated on the fly by linearization around the current operating point.

When one is interested in studying the behavior of several solvers on a single MPC example, relevant information for assessing the performance is: closed-loop trajectories, number of iterations, execution times, KKT tolerances and number of active inequality constraints in the optimal solution which can be an indicator of how difficult each QP is.

Simulation results with a nonlinear model of a continuously stirred tank reactor [21] are shown in Figure 5.1, where the native QP solver of MATLAB is compared to two prototype implementations of the discussed dual fast gradient methods. The input and output trajectories are not displayed as all solvers are asked to return a high-accuracy solution and consequently the plots coincide.

On the other hand, if a practitioner needs to assess the performance of solvers over a broad variety of benchmark examples, the same information must be presented in a more representative and concise way. For this purpose, we make use of performance profiles as introduced in [11]. The notion of these plots is the following: Assume that we have a set of problems \( P \) and a set of solvers \( S \) and define as \( t_{p,s} > 0 \) the time for solver \( s \) to solve problem \( p \). As a metric, other quantities can also be used instead of time, like for instance number of iterations. The performance ratio is expressed as

\[
    r_{p,s} = \frac{t_{p,s}}{\min_{\hat{s} \in S} t_{p,\hat{s}}},
\]

where the denominator comprises the metric of the most efficient solver for problem \( p \). Plotting the cumulative distribution function of the performance ratio\(^1\)

\[
    P_s(\tau) = \frac{\# \{ p \in P : r_{p,s} \leq \tau \}}{\# \{ P \}}
\]

for each solver, yields the desired performance plot. In Equation (5.7), \( P_s(\tau) \) is the probability for solver \( s \in S \) that a performance ratio \( r_{p,s} \) is within a factor \( \tau \in \mathbb{R} \) of the best possible ratio. As an additional option, the log scale of the performance profiles can be plotted:

\[
    P_s(2^{\tau}) = \frac{\# \{ p \in P : \log_2(r_{p,s}) \leq \tau \}}{\# \{ P \}}
\]

\(^1\)The operator \( \# \{ \cdot \} \) denotes here the cardinality of a set.
in order to reveal all activity for $\tau < r_M$. The parameter $r_M$ is defined as $r_{p,s} \leq r_M$ with $r_{p,s} = r_M$ if and only if solver $s$ cannot solve problem $p$. An example that illustrates the comparison of different first-order methods over a collection of approximately one thousand QPs is shown in Figure 5.2. The employed metric is the number of iterations, assuming that all solvers have comparable cost-per-iteration.

5.2 Ill-conditioning

In this section we demonstrate how the generalized variants of Algorithms 2 and 3 can improve the convergence speed by several orders of magnitude when the optimization problem at hand is ill-conditioned.

As an example we consider the control of an unstable aircraft model [24]. The same plant, but with slightly different tuning is used in [19] as well as in the tutorial of the MPC toolbox in MATLAB. The LTI model has four states, two inputs and two outputs which have to respect the following bounds

\[-25^\circ \leq u_1 \leq 25^\circ \text{ elevator angle}\]
\[-25^\circ \leq u_2 \leq 25^\circ \text{ flaperon angle}\]
\[-0.5^\circ \leq y_1 \leq 0.5^\circ \text{ attack angle}\]
\[-100^\circ \leq y_2 \leq 100^\circ \text{ pitch angle}.\]

The goal is to follow a step change in the pitch angle from $0^\circ$ to $10^\circ$ and then back to $0^\circ$. All five solvers are called to solve a series of eighty QPs in a Receding Horizon Control (RHC) fashion, derived using the xu* formulation. The weights of the objective function
in (2.3) are constant and equal to
\[
Q_k = \begin{bmatrix} 10^2 & 0 \\ 0 & 10^2 \end{bmatrix}, \quad R_k = \begin{bmatrix} 10^{-2} & 0 \\ 0 & 10^{-2} \end{bmatrix}, \quad Q_N = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 10^2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10^2 \end{bmatrix},
\]
(5.9)
while state and output dynamics are given by the following equations:
\[
x_{k+1} = \begin{bmatrix} 0.9993 & -3.0083 & -0.1131 & -1.6081 \\ -0.0000 & 0.9862 & 0.0478 & 0.0000 \\ 0.0000 & 2.0833 & 1.0089 & -0.0000 \\ 0.0000 & 0.0526 & 0.0498 & 1.0000 \end{bmatrix} x_k + \begin{bmatrix} -0.0804 & -0.6347 \\ -0.0291 & -0.0143 \\ -0.8679 & -0.0917 \\ -0.0216 & -0.0022 \end{bmatrix} u_k \quad (5.10a)
\]
\[
y_k = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} x_k. \quad (5.10b)
\]
Note that the simple structure of Equation (5.10b), that essentially maps to the outputs the second and forth state of the system, and the absence of polyhedral constraints enable us to use the xu* formulation for Algorithm 2. However, since matrices \( \tilde{Q}_k = C_k^T Q_k C_k \) and \( Q_N \) are rank deficient, we need to perturb the Hessian with a positive constant \( \delta \) for both algorithms. In our setup this term is chosen as \( \delta = 10^{-4} \), which yields a condition number of \( 10^6 \) for matrix \( H_s \).

As a termination condition in each QP of the closed-loop simulation we use the distance from the optimal solution \( z^* \), i.e. Equation (5.3) with \( \epsilon = 0.5\% \). Figure 5.3 shows the required number of iterations for each QP and solver. Note that the scale of the y-axis is logarithmic, indicating that the performance of the dual fast gradient methods is improved by more than one order of magnitude. In fact, Algorithm 2 benefits more from the use of the generalized matrix than Algorithm 3 and not only for this specific example.\footnote{More extensive simulations with the benchmarking suite have shown that this is common in practice.}

Note that both standard and generalized variants of GPAD solve the problem in one iteration when no constraints are active in the optimal solution.

## 5.3 Polyhedral inequality constraints

In this section, we demonstrate that the performance of Algorithm 2 can be severely affected when polyhedral constraints are entering the problem formulation. This is true for both standard and generalized variants.

As an example, we use an inverted pendulum system [31]. The system has three unconstrained states and one input which is restricted between
\[
-1.25 \leq u \leq 1.25.
\]
(5.11)
The objective is to regulate the system to the origin, starting from the initial condition \( x = [0.6 \ 0.6 \ 0.6]^T \). As a termination condition in each QP of the closed-loop simulation we use again Equation (5.3) but with \( \epsilon = 0.1\% \).
Figure 5.3: Number of iterations for problem with ill-conditioned Hessian and unstable model.

We have seen that Algorithm 2 introduces dual Lagrange multipliers for equalities and polyhedral inequalities. In order to demonstrate that the latter negatively affects the performance, we express the existing bounds on inputs as polyhedral constraints.\(^3\) Although this modification does not change the QP problems as such, we will show that the number of iterations increases significantly.

The number of iterations for a series of hundred forty QPs is presented in Figure 5.4. We note that the generalized variant of Algorithm 2 needs significantly more operations to solve essentially the same optimization problems while the convergence of GPAD is not affected at all. The latter was expected since GPAD handles all inequalities in the same way.

### 5.4 Different sparse formulations of Richter’s Dual FGM

In this section, we study the behavior of Algorithm 2 under the different sparse formulations introduced in Section 2.4. Both standard and generalized variants are considered, showing that the latter is more robust to ill-conditioning due to regularization.

As a benchmark example, we use the linearized quadcopter model provided in the demos of the MPT Toolbox [1]. The system has four control inputs and twelve states, namely the coordinates \(\theta, \phi, \psi, x, y, z\) and their derivatives. Since we do not want to penalize or constrain the derivatives, we chose to define the first six states as outputs of the system. Due to the simple form of the resulting output equation, the \(\text{xu}^*\) formulation is applicable in this example. Moreover, as the outputs are only copies of the states, the \(\text{xuy}\) formulation would introduce an unnecessary overhead.\(^4\) Finally, the \(\text{uy}\) formulation is always a favorable alternative in cases with short horizons and large state space. The first and third options will therefore be compared in terms of number of iterations and

\(^3\)The existing input bounds where either kept in the formulation as redundant constraints or deleted. Both cases yield similar results and therefore we focus on the latter for the rest of this section.

\(^4\)In fact, the \(\text{xuy}\) formulation should be used as a substitute of the \(\text{xu}\) formulation only when the latter cannot preserve the diagonal nature of the Hessian.
Figure 5.4: Effect of polyhedral constraints on convergence of generalized Algorithm 2.

The input and output constraints that need to be satisfied at all times in closed-loop are the following:

\[-0.9916 \leq u_i \leq 2.4084 \quad \text{for} \quad i = 1, 2, 3, 4\]
\[-\frac{\pi}{6} \leq y_1 \leq \frac{\pi}{6} \quad \theta \text{ (pitch)}\]
\[-\frac{\pi}{6} \leq y_2 \leq \frac{\pi}{6} \quad \phi \text{ (roll)}\]
\[-1 \leq y_6 \leq \infty \quad z, \quad (5.12)\]

while the weights on inputs and outputs are chosen as

\[
Q = \begin{bmatrix}
10^{-3} & 0 & 0 & 0 & 0 & 0 \\
0 & 10^{-3} & 0 & 0 & 0 & 0 \\
0 & 0 & 10 & 0 & 0 & 0 \\
0 & 0 & 0 & 10 & 0 & 0 \\
0 & 0 & 0 & 0 & 10 & 0 \\
0 & 0 & 0 & 0 & 0 & 10
\end{bmatrix}, \quad R = \begin{bmatrix}
10^{-1} & 0 & 0 & 0 \\
0 & 10^{-1} & 0 & 0 \\
0 & 0 & 10^{-1} & 0 \\
0 & 0 & 0 & 10^{-1}
\end{bmatrix}. \quad (5.13)
\]

For \( Q_N \) we use the same diagonal elements as in \( Q \) and add a weight of \( 10^{-3} \) to the remaining six states that are not mapped to the outputs. The control objective is to track the set point

\[
y^* = [0 \ 0 \ 1 \ 0 \ 0 \ 0]^T, \quad (5.14)
\]
running in closed-loop and starting from the initial condition \( x = 0 \). Note that in contrast to the original benchmark, we allow for control horizon equal to the prediction horizon (\( N = 10 \)) and we neglect all weights on input and output rates.

\footnote{Actual limits on inputs are 9.6 and 13 but (5.12) takes into account the linearization around the steady state, where all four inputs are equal to 10.5916 and all states are zero.}
Since the product $C^T Q C$ is rank deficient due to missing weights on the last six states, the xu* formulation needs to perturb the QPs and use as Hessian the matrix $H_δ = H + δI$. This is not the case for the uy formulation since $Q, R, Q_N \succ 0$ and the given optimization problems can be solved exactly. The largest perturbation $δ > 0$, such that the optimal solution $z_δ^*$ to the perturbed problems satisfies the condition $||z_δ^* - z^*||/||z^*|| \leq 0.5\%$ for the whole closed-loop simulation is $10^{-3}$.

To compare the two formulations in a fair way, we use as termination condition the distance from the a priori known optimal input trajectory, i.e. Equation (5.4) with $ε = 0.1\%$. As we further decrease the value of $δ$ in the xu* formulation, the returned solution comes even closer to the original one but the condition number of the QP inevitably increases. The maximum number of iterations in a closed-loop scenario as a function of the regularization term is shown in Table 5.1, where the iterations of the uy formulation are only quoted for direct comparison.

It is evident from this table that the generalized variant of Algorithm 2 is less sensitive to ill-conditioning. In fact, the number of iterations does not change for regularization terms up to $8 \times 10^{-14}$. The standard dual fast gradient method on the other hand is severely affected, as the scalar step size is reduced by approximately an order of magnitude in each decrease of $δ$. Note that in this example, the xu* is faster than the uy formulation only when the regularization term is high enough to actually distort the optimal solution.

More extensive simulations using the benchmarking suite, showed that indeed for appropriate values of the regularization constant, that need to be tuned in each experiment, the generalized variant of Algorithm 2 returns the optimal solution in approximately the same number of iterations for all admissible sparse variants. This result favors the complexity analysis of the algorithm in Section 4.7 since the right choice is inherently connected to the dimensions of the problem. For the given benchmark example, according to Figure 5.5, the uy formulation is expected to yield more efficient implementations in this time-invariant setup for horizon length roughly up to ten steps.

---

6This approach was taken from [29] where also solvers that may or may not require regularization are compared.  
7Obtained using qpOASES before solving each QP.  
8For smaller values, the solver fails to return a solution due to finite precision arithmetic.  
9Although for a proper decision, the hardware limitations must be also taken into account.
Figure 5.5: Theoretical complexity for quadcopter benchmark example.
Chapter 6

Nonlinear Case Study

In this chapter we use an industrial example\textsuperscript{1} in the field of power electronics to demonstrate how the previous material of the Thesis can be employed for the design of MPC controllers based on gradient methods. Since the target hardware is an FPGA platform with limited resources that needs to operate at high sampling rates, our objective is to find a suitable algorithm that is efficient, light and highly parallelizable. As we have already discussed, fast gradient methods do exhibit these characteristics and therefore the question we are called to answer is what are the advantages and disadvantages for each feasible combination of algorithm and formulation.

Section 6.1 introduces the linear time-varying problem description and summarizes the properties of the given model. Section 6.2 discusses the suitability of the available algorithms and the favorable QP formulations in each case. Section 6.3 analyzes the complexity of the candidate implementations while Section 6.4 uses the MPC benchmarking suite as a tool to assess their numerical performance. Finally, in Section 6.5 we combine the results of the simulations and the complexity analysis in order to guide the selection of the most appropriate implementation.

6.1 Problem description

The given nonlinear model comprises five states, four inputs and four outputs. Its linearization yields the parameter-dependent LTI system:

\[
\begin{align*}
x_{k+1} &= A(p)x_k + Bu_k + f(p) \\
y_k &= C(p)x_k + e_k(p),
\end{align*}
\]

where the parameter \( p \) lies in a five-dimensional space and it is independent of the control input and the initial condition. There are both input and output constraints in form of simple, potentially varying bounds. Constraints on outputs are softened to prevent infeasibility. The objective is to regulate the system to a point \((y^*, u^*)\) that might also vary over time. For a given value of \( p \), the resulting optimization problem can be written

\textsuperscript{1}Provided by ABB Corporate Research.
as

\[
\begin{align*}
\text{minimize} & \quad \sum_{k=0}^{N-1} \frac{1}{2} \begin{pmatrix} y_k - \bar{y} \end{pmatrix}^T \begin{pmatrix} Q & 0 \\ 0 & R \end{pmatrix} \begin{pmatrix} y_k - \bar{y} \\ u_k - \bar{u} \end{pmatrix} + \frac{1}{2} s_k^T W s_k \\
\text{subject to} & \quad x_0 = x \\
& \quad x_{k+1} = \bar{A} x_k + B u_k + \bar{f} \\
& \quad y_k = \bar{C} x_k + \bar{e} \\
& \quad u^l \leq u_k \leq u^u \\
& \quad y^l - s_k \leq y_k \leq y^u + s_k \\
& \quad s_k \geq 0,
\end{align*}
\]  

(6.2a)

subject to  

\[
\begin{align*}
x_0 = x \\
x_{k+1} = \bar{A} x_k + B u_k + \bar{f} \\
y_k = \bar{C} x_k + \bar{e} \\
u^l \leq u_k \leq u^u \\
y^l - s_k \leq y_k \leq y^u + s_k \\
s_k \geq 0.
\end{align*}
\]

\[\text{subject to} \quad x_0 = x \quad (6.2b) \]

\[x_{k+1} = \bar{A} x_k + B u_k + \bar{f} \quad (6.2c) \]

\[y_k = \bar{C} x_k + \bar{e} \quad (6.2d) \]

\[u^l \leq u_k \leq u^u \quad (6.2e) \]

\[y^l - s_k \leq y_k \leq y^u + s_k \quad (6.2f) \]

\[s_k \geq 0, \quad (6.2g)\]

where the horizon length \(N\) is a design parameter with a value close to 10. Variables on red color and with a bar (e.g. \(\bar{A}\)) are problem-varying. According to our definition in Section 4.3, these variables are constant in each QP but different in successive instances. Problem (6.2) is a special case of the general MPC problem in Section 2.2 that eventually favors some combinations of solvers and formulations more than others. The suitable options are first discussed in Section 6.2 and then compared in the rest of the chapter.

### 6.2 Suitable algorithms and formulations

In this section we assess the suitability of the first-order methods described in Chapter 3 under the different QP formulations of Sections 2.3 and 2.4. First of all, due to the output inequality constraints in (6.2f), the fast gradient method is only applicable in the dual domain and Algorithm 1 is therefore rejected. The dual fast gradient method variants as well as ADMM can all be used to solve the given optimization problem and their properties are briefly discussed in the paragraphs that follow.

**Richter’s dual FGM.** Given that there are no polyhedral inequality constraints and all weight matrices are diagonal, Algorithm 2 can be used in the current framework. As matrix \(\bar{C}\) couples some of the states, we cannot use the xu formulation and hence we are restricted to the remaining two. The soft constraints are tackled with the implementation that we developed in Section 3.2.5, yielding a complexity similar to the hard-constrained problem. Simulations with the benchmarking suite showed that the generalized algorithm yields superior performance and therefore we will only consider this variant in our analysis. Finally, the parameter-dependent nature of the dynamics in combination with the constant weights allow us to transfer the factorization of the generalized matrix offline, by calculating an \(L\) that guarantees convergence for all QP instances.\(^2\) This modification is expected to worsen the performance of the method, due to the conservatism that we introduce, but the algorithmic scheme becomes simpler and thus more suitable for an FPGA implementation.

\(^2\)Assuming that the parameter space is explored in an exhaustive way.
Algorithm 3 is less restrictive with respect to the problem setup and it can be applied directly on Problem (6.2). The presence of soft constraints simply transforms the box output constraints to polyhedral inequalities. We will discuss both u and xu formulations, although the given horizon and problem dimensions seem to favor the latter. The standard and generalized variants are both applicable in the given context, providing several options on the calculation of L. For the generalized case, the matrix L must be calculated beforehand since solving an SDP online is not desirable. This leaves us with two options, either use \( L_{M1} \) that does not depend on the varying dynamics,\(^3\) or calculate a robust value for \( L_{M2} \) that satisfies condition (4.89) for all samples of parameter \( p \). Simulations will show which method is to be preferred, although it seems reasonable to assume that using a matrix that takes into account the dynamics will yield better performance. In terms of complexity, the two options are equivalent since the calculations are performed offline and the derived matrix is diagonal in both cases. Using a scalar value instead, would only make sense if we wanted to calculate a tight Lipschitz constant that takes into account the exact dynamics at each time step, i.e. \( L_{S2} \). However, our simulations showed that the performance compared to the generalized variant is worse in terms of maximum number of iterations and therefore the complexity analysis of this case will not be considered.

For the sparse variant of ADMM, the soft constraints must be tackled with an approach similar to the one reported in Section 3.2.5, as the presence of polyhedral inequality constraints in the problem would require the use of nested loops.\(^4\) This approach has not been implemented as the focus of this Thesis was on fast gradient methods and therefore we use the condensed variant in our simulations that can deal with soft constraints by simply changing the structure of the inequalities.

### 6.3 Complexity of favorable combinations

In this section, we discuss the complexity of the candidate methods based on the analysis of Chapter 4. The structure of the linearized problem in (6.2) allows for a lot of freedom in implementation issues and assessing the advantages and disadvantages of each choice is critical for the design of the MPC controller. We choose to neglect the complexity analysis of the ADMM algorithm since on the one hand, it has not been considered in Chapter 4 and on the other hand, the simulations of Section 6.4 did not show a superior performance that would justify further investigation of the algorithm’s efficiency.

#### 6.3.1 Richter’s Dual FGM

As explained in Section 6.2, Algorithm 2 can be used either under the xuy or uy formulation. The advantages and disadvantages of each case are discussed in the respective paragraphs.

The initialization part for both variants of the algorithm involves the calculation and factorization of the generalized matrix \( L \). However, if we assume that the dynamics do not vary significantly around the operation point of the system, we can also calculate a robust matrix \( L_r \), transferring all the computational effort of the initialization offline. To

---

\(^3\)When the matrices that form it are taken from the uy formulation.

\(^4\)Because the sub-problems will not have an analytical solution any more.
accomplish this, we search for a positive definite matrix that satisfies the condition

$$L_r \succeq A_s(p)H_s^{-1}A_s^T(p),$$

(6.3)

for all values of the parameter $p$ that may arise in the closed-loop simulations. In order to make this problem tractable, we can generate instances $L_i$ of the generalized matrix by varying the parameter $p$, calculate their convex hull and finally form an SDP that guarantees satisfaction of the condition (6.3) for all the samples. The question that the simulations are called to answer is whether the conservatism introduced in the calculation of $L$ will severely affect the performance or not.

**xuy formulation.** Keeping all states, outputs and inputs as optimization variables, leads to sparse but large optimization problems. A main advantage of this sparsity is that no arithmetic operations are needed for the problem transformation and the required memory resources are reduced to the minimum. The initialization part of the algorithm, assuming the exact matrix $L$ is used instead of the robust value $L_r$, consists of the sparse matrix-matrix multiplication $A_sH_s^{-1}A_s^T$ and the factorization of the product with the recursive algorithm described in (4.43). Since system dynamics are problem-varying, all blocks of the tridiagonal matrix $L$ are identical and the cost of the multiplication is independent of the horizon length. The calculation of one block comprises only the products $\bar{C}epsilon^{-1}C^T$, $\bar{C}epsilon^{-1}A^T$ and $\bar{A}epsilon^{-1}A^T$ that require in total

$$(ny + nx)^2nx \text{ flops},$$

(6.4)

taking into account the symmetry of the results. This cost is negligible when compared to the sparse Cholesky factorization of the matrix that requires

$$N(ny + nx)^2 + N(ny + nx)^2nx + \frac{1}{3}N(ny + nx)^3 \text{ flops}.$$  

(6.5)

We remind that $epsilon_x$ is a diagonal matrix that compensates for the missing weights on the states, ensuring that the Hessian $H_s$ is positive definite. The value of the perturbation term is tuned during the simulations, in an effort to balance numerical errors due to ill-conditioning and distance from the optimal solution of the original problem. Confirming the results of Section 5.4, the generalized variant of Algorithm 2 can handle remarkably low values of $epsilon_x$, yielding exactly the same performance. The complexity of the iterative part is dominated by the two matrix-vector multiplications with $A_s$ that require

$$2Nnx(ny + nx + nu) \text{ flops}$$

(6.6)

each and the forward-backward substitutions that add another

$$2N(nx + ny)(3nx + ny) \text{ flops}$$

(6.7)

in the flop count. Note that since the Cholesky factor $R$ is sparse, it requires much less memory than the dense inverse $L^{-1}$. However, the backward and forward solves are less adequate for parallelization, which is an important aspect for an FPGA implementation.

---

5The product $BR^{-1}B^T$ can be calculated offline since both matrices are constant.
**uy formulation.** Excluding the states from the optimization variables yields strictly positive definite QPs of smaller size, since the given weights have full rank and $Nn_x$ dual variables are eliminated. However, the useful structure of the equality constraint matrix is ruined and the memory requirements are significantly higher. The problem transformation consists of an online condensing, needed to form matrix $A_s$. The fact that the dynamics are constant in each QP instance, renders the complexity of the operation linear to the horizon length. More precisely, the cost is approximately

$$2Nn_yn_z(n_x + n_u) \text{ flops,}$$

and as mentioned in Remark 4.5.3.1, it can be significantly reduced by means of parallelization. Note that the equality constraint matrix $A_s$ comprises now several products between system matrices and consequently it requires more memory resources than in the previous formulation. In the initialization part, the calculation of the generalized matrix $L$ requires

$$\frac{1}{3} N^3n_yn_u \text{ flops,}$$

and its inversion via dense Cholesky factorization another

$$\frac{7}{3} N^3n_y^2 \text{ flops.}$$

We recall that the explicit inversion is preferable in this setup, as the Cholesky factor is expected to be dense. This implies that the backward and forward solves have the same cost as the multiplication with $L^{-1}$ but they cannot be as efficiently parallelized. Finally, the iterative part comprises again two multiplications with $A_s$ of

$$N^2n_yn_u \text{ flops}$$

each and the multiplication with the inverse of the generalized matrix that requires another

$$2N^2n_y^2 \text{ flops,}$$

since it is dense. Finally, note the quadratic dependence on the horizon length for the iterative part and the absence of the term $n_x$ in the expressions of the complexity per iteration.

### 6.3.2 Bemporad’s Dual FGM

Algorithm 3 can be applied on either the u or xu formulation without restrictions. Assuming the same step size $L^{-1}$ is used, the number of iterations for convergence is the same for both formulations and therefore, the problem dimensions and hardware specifications should indicate the better choice in our setup. Due to the varying dynamics, the xu formulation has a significant advantage, as the problem transformation is much less computationally expensive. If the generalized matrix or scalar Lipschitz constant $L$ is calculated offline via one of the options discussed in Section 6.2, the initialization of both implementations comprises only the pre-processing for the solution of the inner problem. That is either the factorization of the KKT matrix or the inversion of the dense Hessian $H_c$. This operation has to be performed online, using the current set of system dynamics, as the inner problem of the algorithm must be solved exactly. Based on the previous statements, we infer that there is no approach to make Algorithm 3 factorization-free. This is a significant drawback against Algorithm 2, taking into account that the method will be deployed on an FPGA platform.
**xu formulation.** The problem transformation of the xu formulation is affected mainly by the calculation of the Hessian matrix. However, since the dynamics are problem-varying and the weights are constant, the only quantity that needs to be computed to build all blocks of $H$ is the product $\bar{C}^T Q \bar{C}$ whose cost is only

$$n_x^2 n_y \text{ flops.}$$

The initialization on the other hand involves the factorization of the KKT matrix that requires another

$$N(3n_x^3 + 6n_x^2 n_u + 6n_x^2 n_x + \frac{1}{3}n_u^3) \text{ flops.}$$

In case $L_{S2}$ is used, the KKT matrix must be also inverted in order to calculate the term $K_{11}$, the product $\tilde{C}_s K_{11} \tilde{C}_s^T$ and the norm of this product. The iterative part consists of the backward and forward solves of Algorithm 7, that have a cost of

$$N(4n_x^2 + 4n_u^2 + 10n_x n_u + 4n_y n_u + 4n_y n_x) \text{ flops,}$$

and a less computationally expensive matrix-vector multiplication with $\tilde{C}_s$. The structure of the sparse matrix $\tilde{C}_s$ is the same as in Equation (4.79), with

$$\tilde{M} = \begin{bmatrix} \bar{C} \\ -\bar{C} \\ 0 \\ 0 \end{bmatrix}, \quad \tilde{N} = \begin{bmatrix} 0 \\ 0 \\ I \\ -I \end{bmatrix}.$$

The complexity of the gradient step is therefore only

$$2Nn_x n_y \text{ flops.}$$

Note that according to (6.15), the complexity of Algorithm 7 is linear in $N$ but the operations are less suitable for parallelization.

**u formulation.** When states and outputs are condensed, the complexity of the problem transformation becomes significantly higher as we need to perform an online condensing that according to the previous section yields

$$2N n_y n_x (n_x + n_u) \text{ flops,}$$

and calculate the Hessian matrix through the product $\bar{B}Q \bar{B}^T$, that following the results of Section 4.5.4 requires approximately

$$\frac{1}{3}N^3 n_y^2 n_u \text{ flops.}$$

The complexity for the calculation of the linear weight $f_c$ is neglected here since it introduces only lower order terms (see Equation (4.31) and recall that $n_y \approx n_x$). The derived Hessian must be inverted in the initialization part in order to be used for the solution of the inner problem. This operation is cubic in the horizon length as it needs

$$\frac{7}{3}N^3 n_u^3 \text{ flops.}$$
Finally for the iterative part, the inner problem comprises the multiplication with $\tilde{C}_c$ that costs
\[ 2N^2 n_y n_u \text{ flops} \quad (6.21) \]
and the multiplication with the dense Hessian that adds another
\[ 2N^2 n_u^2 \text{ flops.} \quad (6.22) \]
The gradient step is dominated by the product $\tilde{C}_c z_{k+1}$, that due to the special structure of the inequality constraint matrix requires
\[ N^2 n_u n_y \text{ flops.} \quad (6.23) \]

### 6.3.3 Comparison of complexities

In this section, we compare the complexity of the discussed choices. The number of floating point operations is summarized in Table 6.1, where the symbol † denotes parts that can be efficiently parallelized. A quick look on the first two columns shows that even though the initialization of the uy formulation is significantly more expensive, its impact on the total complexity will be negligible if the algorithm is executed for a few hundreds iterations. Taking into account that the iterative part can be also more efficiently parallelized, the state elimination might yield faster performance at the expense of increased memory resources. The superiority of the uy formulation becomes more clear if a robust $L$ is used since the initialization part can be transferred offline. As far as GPAD is concerned, we observe that the high cost for the inversion of the dense Hessian together with the larger cost per iteration favors the xu formulation against the condensed variant. However, we remind that the solution of the inner problem for the sparse case is found via an iterative procedure that cannot be as efficiently parallelized.

Since the horizon length is a more flexible parameter in the project than the problem dimensions, we also study the scaling of the complexity with respect to $N$. In Figure 6.1 we plot the theoretical complexity of the four discussed choices. As the curves suggest, for values of the horizon length around ten, the sparse formulations have significantly smaller problem transformation and initialization cost while the cost per iteration is in all cases comparable. To decide on the final design, we need to combine the above a priori knowledge with simulations. This is the content of the next section.

### 6.4 Simulations

The purpose of this section is to compare the performance of the different algorithms on the given problem. The test data are acquired by running qpOASES in closed-loop
and logging all the initial conditions and linearized instants of the model. The derived series of QPs is then solved in an open-loop fashion by all candidate implementations and the number of iterations for convergence is compared. For a fair comparison, we use as termination condition the distance between the current and the optimal input trajectory. More precisely, the method terminates when:

$$\Delta_2(U,U^*) < 0.1\%,$$  \hspace{1cm} (6.24)

with function $\Delta$ as defined in Equation 5.2 and $U^*$ the optimal solution as provided by qpOASES. We deliberately avoid using the whole primal vector $z$ in the termination condition since different formulations comprise different number of optimization variables and therefore the same solution will yield different Euclidean norms. Ultimately, what we are interested in is the optimal input trajectory, that uniquely determines states and outputs given an initial condition $x$.

The given closed-loop input and output trajectories are shown in Figure 6.2. The reference signals were chosen in an effort to sufficiently excite the system and yield as much variation in the dynamics as possible. Therefore, when the robust values for the generalized matrices $L$ are tested, we will be able to observe the response of the system in a sufficient range around the desired operating point. The performance of the solvers in terms of number of iterations is shown in Figure 6.3, were we have merged implementations that yield exactly the same number of iterations. Table 6.2 summarizes the main quantities of the plot, where (gen) means that the generalized variant of the algorithm is used and the terms (r) or (s) indicate whether a robust calculation is employed via sampling of parameter $p$. Note that the standard variant of GPAD, that uses $L_{S2}$ as Lipschitz constant, has roughly the same performance as the robust generalized implementation. However, we choose not to consider it in the subsequent analysis because it is more computationally expensive. More precisely, the algorithm requires the explicit inversion of the KKT matrix, the calculation of the product $\hat{C}_sK_{11}\hat{C}_s^T$ and finally the computation of its norm.

In order to combine information on complexity and performance, we calculate the total

---

6We have not considered warm-starting with the previous dual solution since we are mainly interested in the worse-case scenario.

7r stands for robust and s for standard implementation.
Figure 6.2: Closed-loop simulation of nonlinear model.

Figure 6.3: Number of iterations for convergence to $U^*$. 
number of flops that the discussed implementations need to solve each QP. To compute this number, we simply multiply the number of iterations with the cost per iteration of the algorithm and add the corresponding cost of problem transformation and initialization to the sum. The results for the two dual fast gradient method variants are shown in Figures 6.4 and 6.5. Each plot comprises four lines as we have two options for the formulation and the calculation of $L$ in each method. The several conclusions that can be drawn from the acquired results are discussed in detail in the following section.

### 6.5 Interpretation of results

Examining Figures 6.3, 6.4 and 6.5 carefully, gives us some useful information regarding the design of the controller. Most of the conclusions we can draw, were already expected from the previous discussion but through the simulations we are able to quantify these arguments and decide on the suitability of each option. The main messages from the plots are summarized below:
• The condensed ADMM algorithm does not exhibit a superior performance that would justify further investigation of its computational complexity. In fact, a quick look on Algorithm 5 will convince the reader that the initialization operations are equivalent to the condensed variant of GPAD and therefore prohibitively expensive.

• The xuy and xu formulations of Richter’s dual fast gradient method need exactly the same number of iterations in Figure 6.3. This result is in agreement with the example of Section 5.4. Note that if a stopping condition based on the norm of the primal vector was used instead, it would not be clear whether the two formulations are equivalent because we would compare vectors of different size and content.

• Using a robust value for matrix $L$ in Richter’s dual fast gradient method deteriorates the performance of the method since we introduce conservatism in the selected step sizes. However, according to the available data, the required number of iterations is in the worst case approximately doubled. Taking into account that the implementation will require no online factorizations or matrix inversions, the price of increased iterations might be worth to pay.

• In Figure 6.4, the offset between xuy and uy formulation is smaller for the robust implementations since the initialization part is transferred offline. The small difference in required floating point operations indicates that the decision on the formulation should be rather based on whether there are available memory resources to support condensing and on how simple and parallelizable should the final implementation be. We remind that the xuy formulation contains a backward and forward solve in the gradient step that is replaced by a single matrix-vector multiplication in the uy formulation.

• A robust implementation of the Bemporad’s dual fast gradient method, improves
the convergence of the algorithm by almost a factor of three, making it the most efficient among the considered methods. In fact, Figure 6.5 indicates that the sparse variant requires only half of the floating point operations when compared to the most efficient implementation of Richter’s method. However, the online factorization of the KKT matrix cannot be avoided and although it is not significantly expensive, it does affect the simplicity of the algorithmic scheme.

- Figure 6.5 also shows that the condensed implementation of Algorithm 3 is not able to compete with the sparse variant since the computational overhead yields an increase of approximately 50% in floating point operations.

To conclude, although the above analysis provides valuable information regarding the numerical performance and complexity of the solvers, the decision on the final design can be made only once the exact hardware specifications are known. Given these specifications, one can derive the maximum allowed number of iterations, the available memory resources and the significance of a simple and factorization-free implementation. Moreover, extensive closed-loop simulations should validate the robust calculations and warm-started versions of the algorithms should also be considered.

Given that the illustrated project is still at an early stage of its development, the purpose of this chapter was not to decide which implementation should be deployed on the FPGA platform but rather provide a useful guideline on how the content of the Thesis can be used to assist on the design of gradient-based MPC controllers.
Chapter 7

Conclusion

In this last chapter, we briefly summarize the content of the Thesis and propose some relevant future topics.

7.1 Summary

In the first part of the Thesis, we provided an overview of recently proposed first-order methods for MPC and explained their algorithmic schemes. Then we assessed the online complexity of the algorithms with focus on time-varying setups and simulated their performance in different MPC problems to highlight some of their properties. Finally, a nonlinear MPC example from an ongoing project of ABB Corporate Research was studied in order to illustrate how the complexity analysis together with the developed benchmarking tools can be used to design efficient MPC controllers based on first-order solvers.

7.2 Future work

Since first-order methods are currently an active field of research in the control community, new algorithms are often proposed in scientific papers and keeping the overview and the complexity analysis up-to-date can be an ongoing process. Some further aspects that have not been investigated and could complement the content of the Thesis are listed below:

- Efficient implementations of the algorithms on embedded hardware, that fully take into account the structure of each formulation, would serve in validating the theoretical results and study for what range of problem dimensions is the flop count a meaningful metric of the algorithm’s complexity.

- A more thorough study of the effect of parallelization on the algorithmic steps could make the complexity results more meaningful for FPGA implementations and render the comparison between different formulations easier.

- Similarly to the complexity analysis, a detailed calculation of the memory requirements of each implementation would also be valuable when assessing their suitability for embedded applications.

Finally, the results of Chapter 4 can be potentially incorporated in the MPC benchmarking suite that we presented in Section 5.1, to assess both complexity and performance of first-order solvers in the same environment.
Appendix A

Problem Formulations

In this chapter, we derive the objective function and the constraints of the QP for all possible formulations. The ordering of variables and the adopted notation is used throughout this Thesis.

A.1 xuy formulation

For the fully sparse formulation, where all states, outputs and inputs belong to the optimization variables, we chose the following classification, so as to preserve the structure that is exploited in the complexity analysis,

\[ z = [x_0^T \ u_0^T \ y_0^T \ \ldots \ \ y_{N-1}^T \ x_N^T]^T. \]  

(A.1)

A.1.1 Objective function

For the quadratic and linear terms of the objective function of (2.7), we have

\[ H_s = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & \ldots & 0 & 0 \\ 0 & R_0 & S_0^T & 0 & 0 & \ldots & 0 & 0 \\ 0 & S_0 & Q_0 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \ldots & Q_{N-1} & 0 \\ 0 & 0 & 0 & 0 & 0 & \ldots & 0 & Q_N \end{bmatrix}, \quad f_s = \begin{bmatrix} 0 \\ r_0 \\ q_0 \\ 0 \\ r_1 \\ \vdots \\ q_{N-1} \\ 0 \end{bmatrix} - 2H_s \begin{bmatrix} 0 \\ u_0^r \\ u_0^r \\ 0 \\ u_1^r \\ \vdots \\ y_{N-1}^r \\ x_N^r \end{bmatrix}. \]  

(A.2)

The zero blocks on the diagonal correspond to the weights on the state variables. Therefore they are square matrices of dimension \( n_x \). In case the solver needs a positive definite hessian matrix, a small diagonal weight should be used instead or the Hessian matrix should be perturbed in the sense: \( H_s = H_s^\delta + \delta I \).
A.1.2 Constraints

The equality constraints, using always the order of variables in (A.1), are

\[
A_s = \begin{bmatrix}
I & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\
-C_0 & -D_0 & I & 0 & 0 & 0 & \cdots & 0 & 0 \\
-A_0 & -B_0 & 0 & I & 0 & 0 & \cdots & 0 & 0 \\
0 & 0 & 0 & -C_1 & -D_1 & I & 0 & \cdots & 0 \\
0 & 0 & 0 & -A_1 & -B_1 & 0 & I & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\
\end{bmatrix}, \quad b_s = \begin{bmatrix}
x \\
e_0 \\
f_0 \\
e_1 \\
f_1 \\
e_2 \\
\vdots \\
f_{N-1} \\
\end{bmatrix}, \quad (A.3)
\]

while the polyhedral inequality constraints have the following simple structure:

\[
C_s = \begin{bmatrix}
0 & N_0 & M_0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
0 & -N_0 & -M_0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & N_1 & M_1 & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -N_1 & -M_1 & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & T \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & -T \\
\end{bmatrix}, \quad d_s = \begin{bmatrix}
d_0^u \\
d_0^l \\
d_1^u \\
d_1^l \\
\vdots \\
d_{N-1}^u \\
d_{N-1}^l \\
\end{bmatrix}, \quad (A.4)
\]

A.2 xu formulation

When the output variables are eliminated, we chose to order the variables as

\[
z = [x_0^T \quad u_0^T \quad \cdots \quad u_{N-1}^T \quad x_N^T]^T. \quad (A.5)
\]

Note that in the special case where \( y = x \), the xu formulation boils down to the standard sparse formulation.

A.2.1 Objective function

The quadratic and linear weights of this formulation have the following structure:

\[
H_s = \begin{bmatrix}
H_{0,0} & 0 & 0 & \cdots & 0 & 0 \\
0 & H_{1,1} & 0 & \cdots & 0 & 0 \\
0 & 0 & H_{2,2} & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & H_{N-1,N-1} & 0 \\
0 & 0 & 0 & \cdots & 0 & Q_N \\
\end{bmatrix}, \quad f_s = \begin{bmatrix}
f_{0,0} \\
f_{1,1} \\
f_{2,2} \\
\vdots \\
f_{N-1,N-1} \\
0 \\
\end{bmatrix}, \quad (A.6)
\]
with

\[H_{k,k} = \begin{bmatrix} C_k^T Q_k C_k & C_k^T Q_k D_k + C_k^T S_k \\ D_k^T Q_k C_k + S_k^T C_k & R_k + D_k^T Q_k D_k + D_k^T S_k + S_k^T D_k\end{bmatrix}\]  \hspace{1cm} (A.7)

\[f_{k,k} = \begin{bmatrix} C_k^T (2Q_k e_k - 2Q_k y_k^r - 2S_k u_k^r + q_k) \\ D_k^T (2Q_k e_k - 2Q_k y_k^r - 2S_k u_k^r + q_k) + 2S_k (e_k - y_k^r) - 2R_k u_k^r + r_k\end{bmatrix}\]  \hspace{1cm} (A.8)

**Remark A.2.1.1.** The structure of \(H_s\) in this setup is block diagonal. We remind at this point that its inverse will preserve this structure.

### A.2.2 Constraints

The equality constraints are similar to the xuy formulation,

\[A_s = \begin{bmatrix} I & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ -A_0 & -B_0 & I & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & -A_1 & -B_1 & I & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & -B_{N-1} & I\end{bmatrix}, \quad b_s = \begin{bmatrix} x \\ f_0 \\ f_1 \\ \vdots \\ f_{N-1}\end{bmatrix}\]  \hspace{1cm} (A.9)

and the polyhedral inequality constraints become:

\[C_s = \begin{bmatrix} M_0 C_0 & M_0 D_0 + N_0 & 0 & 0 & \cdots & 0 \\ -M_0 C_0 & -M_0 D_0 - N_0 & 0 & 0 & \cdots & 0 \\ C_0 & D_0 & 0 & 0 & \cdots & 0 \\ -C_0 & -D_0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & M_1 C_1 & M_1 D_1 + N_1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & T \\ 0 & 0 & 0 & 0 & \cdots & -T\end{bmatrix}, \quad d_s = \begin{bmatrix} d_0^s - M_0 e_0 \\ -d_0^s + M_0 e_0 \\ y_0^s - e_0 \\ -y_0^s + e_0 \\ d_1^s - M_1 e_1 \\ \vdots \\ d_N^s \\ -d_N^s \end{bmatrix}\]  \hspace{1cm} (A.10)

**Remark A.2.2.1.** In the xu* formulation, the Hessian matrix \(H_s\) is diagonal and all polyhedral inequalities in \(C_s\) can be transformed to simple state constraints.
A.3 uy formulation

For the last sparse formulation, let us first define the state, input and output vectors as:

\[ X = [x_0^T \ x_1^T \ldots x_N^T]^T \]
\[ U = [u_0^T \ u_1^T \ldots u_{N-1}^T]^T \]
\[ Y = [y_0^T \ y_1^T \ldots y_{N-1}^T \ x_N^T]^T. \]

(A.11)

Note that the terminal state is included in the output vector to ease notation later on. We can choose any ordering for the optimization variables here since in any case there is no structure to exploit. The vector of primal variables is conveniently defined as:

\[ z = [Y^T \ U^T]^T. \]

(A.12)

A.3.1 Dynamic equations

Expressing first the states as a function of the inputs yields:

\[ X = Ax + BU + F, \]

(A.13)

with

\[ A = \begin{bmatrix} I \\ A_0 \\ A_1A_0 \\ \vdots \\ A_{N-1}A_{N-2} \ldots A_0 \end{bmatrix}, \]

(A.14)

\[ B = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ B_0 & 0 & 0 & \cdots & 0 \\ A_1B_0 & B_1 & 0 & \cdots & 0 \\ A_2A_1B_0 & A_2B_1 & B_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{N-1} \ldots A_1B_0 & A_{N-1} \ldots A_2B_1 & A_{N-1} \ldots A_3B_2 & \ldots & B_{N-1} \end{bmatrix}, \]

(A.15)

\[ F = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ I & 0 & 0 & \cdots & 0 \\ A_1 & I & 0 & \cdots & 0 \\ A_{2A_1} & A_2 & I & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{N-1} \ldots A_1 & A_{N-1} \ldots A_2 & A_{N-1} \ldots A_3 & \cdots & I \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \end{bmatrix}. \]

(A.16)
For the LTI case, the matrices are simplified to:

\[ A = \begin{bmatrix} I & A & A^2 & A^3 & A^4 & \cdots & A^N \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 & 0 & 0 & \cdots & 0 \\ I & A & A^2 & A^3 & A^4 & \cdots & A^N \end{bmatrix} \]

(A.17)

\[ F = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ I & I & A & A^2 & A^3 & \cdots & A^N \end{bmatrix} \]

(A.18)

Now the output vector can be written in vectorized notation as

\[ Y = C X + D U + E, \quad \text{(A.19)} \]

where

\[ C = \begin{bmatrix} C_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & C_1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & C_{N-1} & 0 \\ 0 & 0 & 0 & \cdots & 0 & I \end{bmatrix}, \quad \text{(A.20)} \]

\[ D = \begin{bmatrix} D_0 & 0 & 0 & \cdots & 0 \\ 0 & D_1 & 0 & \cdots & 0 \\ 0 & 0 & D_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & D_{N-1} \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}, \quad E = \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ \vdots \\ e_{N-1} \\ 0 \end{bmatrix}, \quad \text{(A.21)} \]

The LTI matrices are in principle the same, without the time index. Finally, writing the output vector as a function of the inputs yields:

\[ Y = C A x + (C B + D) U + C F + E = \tilde{A} x + \tilde{B} U + \tilde{F} + \tilde{E}, \quad \text{(A.22)} \]
where the structure of matrices $\bar{A}, \bar{B}, \bar{F}$ is explicitly shown in the following equation.

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
\vdots \\
y_{N-1} \\
x_N
\end{bmatrix} = \begin{bmatrix}
C_0 \\
C_1 A_0 \\
C_2 A_1 A_0 \\
\vdots \\
C_{N-1} A_{N-2} \cdots A_0 \\
A_{N-1} A_{N-2} \cdots A_0
\end{bmatrix} \begin{bmatrix}
D_0 & 0 & 0 & \cdots & 0 \\
C_1 B_0 & D_1 & 0 & \cdots & 0 \\
C_2 A_1 B_0 & C_2 B_1 & D_2 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
C_{N-1} A_{N-2} B_0 & \cdots & C_{N-1} A_{N-2} B_1 & \cdots & D_{N-1} \\
A_{N-1} A_{N-2} B_0 & \cdots & A_{N-1} A_{N-2} B_1 & \cdots & B_{N-1}
\end{bmatrix} \begin{bmatrix}
u_0 \\
u_1 \\
u_2 \\
\vdots \\
u_{N-2} \\
u_{N-1}
\end{bmatrix} + \begin{bmatrix}
f_0 \\
f_1 \\
f_2 \\
\vdots \\
f_{N-2} \\
f_{N-1}
\end{bmatrix}
\]

(A.23)

The above results will be used to formally express the equality constraints of the QP.

### A.3.2 Objective function

The quadratic and linear weights of the objective function are simply

\[
H_s = \begin{bmatrix} Q & S \end{bmatrix} S^T \begin{bmatrix} R \end{bmatrix}, \quad f_s = \begin{bmatrix} G_Y & G_U \end{bmatrix} - 2H_s \begin{bmatrix} Y_R & U_R \end{bmatrix},
\]

(A.24)

with matrices

\[
Q = \begin{bmatrix}
Q_0 & 0 & 0 & \cdots & 0 \\
0 & Q_1 & 0 & \cdots & 0 \\
0 & 0 & Q_2 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & Q_N
\end{bmatrix}, \quad R = \begin{bmatrix}
R_0 & 0 & 0 & \cdots & 0 \\
0 & R_1 & 0 & \cdots & 0 \\
0 & 0 & R_2 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & R_{N-1}
\end{bmatrix},
\]

(A.25)

\[
S = \begin{bmatrix}
S_0 & 0 & 0 & \cdots & 0 \\
0 & S_1 & 0 & \cdots & 0 \\
0 & 0 & S_2 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & S_{N-1} \\
0 & 0 & 0 & \cdots & 0
\end{bmatrix}
\]

(A.26)
and vectors

\[
Y_R = \begin{bmatrix} y_0^r \\ y_1^r \\ y_2^r \\ \vdots \\ y_{N-1}^r \\ x_N^r \end{bmatrix}, \quad G_Y = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ \vdots \\ q_{N-1} \\ 0 \end{bmatrix}, \quad U_R = \begin{bmatrix} u_0^r \\ u_1^r \\ u_2^r \\ \vdots \\ u_{N-1}^r \end{bmatrix}, \quad G_U = \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ \vdots \\ r_{N-1} \end{bmatrix}.
\] (A.27)

**A.3.3 Constraints**

The equality constraint matrix and equality constraint vector, based on Equation (A.22) are:

\[
A_s = \begin{bmatrix} I & -\bar{B} \end{bmatrix}, \quad b_s = \bar{A}x + \bar{F} + \bar{E},
\] (A.28)

while the inequality constraints:

\[
C_s = \begin{bmatrix} \mathcal{M} & \mathcal{N} \\ -\mathcal{M} & -\mathcal{N} \end{bmatrix}, \quad d_s = \begin{bmatrix} D_U \\ -D_L \end{bmatrix},
\] (A.29)

with

\[
\mathcal{M} = \begin{bmatrix} M_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & M_1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & M_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & M_{N-1} & 0 \\ 0 & 0 & 0 & \cdots & 0 & T \end{bmatrix}, \quad \mathcal{N} = \begin{bmatrix} N_0 & 0 & 0 & \cdots & 0 \\ 0 & N_1 & 0 & \cdots & 0 \\ 0 & 0 & N_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & N_{N-1} \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix},
\] (A.30)

\[
D_U = \begin{bmatrix} d_0^u \\ d_1^u \\ \vdots \\ d_N^u \end{bmatrix}, \quad D_L = \begin{bmatrix} d_0^l \\ d_1^l \\ \vdots \\ d_N^l \end{bmatrix}.
\] (A.31)
A.4 u formulation

Finally, for the condensed case, we use the same notation as in Section A.3 due to the underlying similarities of the two formulations.

A.4.1 Objective function

The elements of the quadratic objective function in (2.4) are

\[ H_c = \bar{B}^T Q \bar{B} + R + S^T \bar{B} + \bar{B}^T S, \]  

(A.32a)

\[ f_c = 2 (\bar{B}^T \bar{A} + S \bar{A}) \bar{x} + \bar{B}^T (2Q F + GY - 2QY_R - 2SU_R) + 2S^T \bar{F} + GU - 2RU_R - 2SY_R. \]  

(A.32b)

Remark A.4.1.1. Note that the dependence of the linear term \( f_c \) on the initial condition \( \bar{x} \) and the reference trajectories is linear.

A.4.2 Constraints

To derive the matrix \( C_c \) and the vector \( d_c \) used in (2.4), we first define the vectors:

\[
Y_U = \begin{bmatrix}
y_0^u \\
y_1^u \\
y_2^u \\
\vdots \\
y_{N-1}^u
\end{bmatrix}, \quad Y_L = \begin{bmatrix}
y_0^l \\
y_1^l \\
y_2^l \\
\vdots \\
y_{N-1}^l
\end{bmatrix}, \quad U_U = \begin{bmatrix}
u_0^u \\
u_1^u \\
u_2^u \\
\vdots \\
u_{N-1}^u
\end{bmatrix}, \quad U_L = \begin{bmatrix}
u_0^l \\
u_1^l \\
u_2^l \\
\vdots \\
u_{N-1}^l
\end{bmatrix}.
\]  

(A.33)

In the condensed formulation, \( C_c \) comprises all polyhedral inequalities and simple output constraints. Namely,

\[
C_c = \begin{bmatrix}
\bar{B} & -\bar{A} \\
-\bar{B} & \bar{A} \\
M\bar{B} + N & -M\bar{A} \\
-M\bar{B} - N & M\bar{A}
\end{bmatrix}, \quad d_c = \begin{bmatrix}
Y_U - F - E \\
Y_L + F + E \\
D_U - M\bar{F} - M\bar{E} \\
-D_L + M\bar{F} + M\bar{E}
\end{bmatrix}.
\]  

(A.34)

Finally, for the inequality constraints of the QP in (2.5), where simple and polyhedral inequalities are merged, we have:

\[
\tilde{C}_c = \begin{bmatrix}
C_c \\
I \\
-I
\end{bmatrix}, \quad \tilde{d}_c = \begin{bmatrix}
d_c \\
U_U \\
-U_L
\end{bmatrix}.
\]  

(A.35)
Bibliography


