Using ADMM for Hybrid System MPC

Mattias Fält

Lucas Jimbergsson

Department of Automatic Control

MSc Thesis
ISRN LUTFD2/TFRT--5981--SE
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Abstract

Model Predictive control (MPC) has been studied extensively because of its ability to handle constraints and its great properties in terms of stability and performance [Mayne et al., 2000]. We have in this thesis focused on MPC of Hybrid Systems, i.e. systems with both continuous and discrete dynamics. More specifically, we look at problems that can be cast as Mixed Integer Quadratic Programming (MIQP) problems which we are solving using a Branch and Bound technique. The problem is in this way reduced to solving a large number of constrained quadratic problems. However, the use in real time systems puts a requirement on the speed and efficiency of the optimization methods used. Because of its low computational cost, there have recently been a rising interest in the Alternating Direction Method of Multiplies (ADMM) for solving constrained optimization problems. We are in this thesis looking at how the different properties of ADMM can be used and improved for these problems, as well as how the Branch and Bound solver can be tailored to accompany ADMM. We have two main contributions to ADMM that mitigate some of the downsides with the often ill-conditioned problems that arise from Hybrid Systems. Firstly, a technique for greatly improving the conditioning of the problems, and secondly, a method to perform fast line search within the solver. We show that these methods are very efficient and can be used to solve problems that are otherwise hard or impossible to precondition properly.

Acknowledgements

We want to thank our advisor Pontus Giselsson at the department of Automatic Control, Lund University for all his help during this project. Not only did he guide us through this project, but his ability and willingness to help us with any question we had was truly valuable.

We also want to thank our families for supporting us in our studies.

Contents

1

Introduction

1.1 Background

Model Predictive Control is the idea of controlling a system by predicting its behavior and selecting an optimal control signal based on this prediction. The control signal is usually chosen to minimize some cost over the system states, inputs and outputs.

The most common case is unconstrained linear systems with quadratic costs. This problem can be efficiently solved for time-invariant and time varying problems, in the discrete as well as the continuous case. It is also possible to solve constrained problems (as we do in this thesis), but it requires more computation.

Stability can often be achieved by choosing the prediction horizon long enough, preferably considerably longer than the time constants in the system. Model Predictive Control (MPC) is therefore often chosen for its ability to handle constraints, as well as its good properties in terms of performance and robustness [Mayne et al., 2000].

Hybrid Systems are able to model a much wider range of systems than ordinary Linear Systems. We are in this thesis looking at a subset of hybrid systems that can be modeled as linear systems with a mix of continuous and binary variables. This class can not only approximate many non-linear systems as piece-wise affine, but also systems with different modes and logics.

However, these problems can also be considerably harder to solve and the complexity can grow exponentially with the number of binary variables. This is a problem since the use in real time systems requires that the full problem can be solved between each time sample, preferably even faster. This is therefore putting a bound on how fast systems we can control.

Explicit MPC is one way to push this bound farther. By explicitly computing the solutions for the whole state space in advance, it is possible to control much faster systems. However, this method is subject to the curse of dimensionality and is only feasible for relatively small systems, even more so for problems with quadratic cost.

The Branch and Bound method is an alternative to the exponentially growing brute-forcing approach of testing all combinations of the binary variables. By solving a series of relaxed problems, the idea is to successively find bounds on the solution, and thus hopefully drastically reducing the range of solutions that needs to be considered.

The problem of solving these individual relaxed (Quadratic Programming (QP)) problems persists. There are many properties of a solver that can be beneficial in this setting. The ability to warm start the solver (i.e. to use a previous solution or guess as initial point) can be very valuable since little is expected to change between time steps. Many of the relaxed problems that are being solved are also very similar which makes it compelling to use a method that can exploit this. Previous work has for example considered dual active set methods for these reasons [Axehill and Hansson, 2006].

We have chosen to look at the Alternating Direction Method of Multipliers (ADMM). It is a first-order method with very cheap iterations. It also has the ability to initialize the solver with an infeasible solution. This is valuable since old solutions might not be feasible at a later time, when warm-starting the solver. Lastly, a parametric implementation of ADMM exists which further exploits the similarities between the problems that need to be solved (QPgen [Giselsson, 2015]).

1.2 Purpose and Restrictions

The purpose of this thesis is to investigate how ADMM can be used for Model Predictive Control of hybrid systems. In particular, we will consider using ADMM for solving subproblems in a Branch and Bound solver. Our goal is not to implement a state-of-the-art Branch and Bound solver – instead we want to investigate how ADMM can be improved to work better in this setting, and also how the properties of ADMM can be exploited in the Branch and Bound solver.

1.3 Thesis Outline

In Section 2 we present some optimization theory relevant to this thesis. Section 3 then gives a brief overview of MPC for linear systems, followed by a more thorough presentation of MPC for hybrid systems. Together, Sections 2 and 3 give some background to our thesis, while the following sections describe our own ideas and solutions (unless stated otherwise).

In Section 4 we outline the two examples that we work with. Section 5 describes some modifications or extensions to ADMM of ours, to improve the performance when applied to MPC for hybrid systems in a Branch and Bound solver. Section 6 instead presents some ways in which the Branch and

Bound solver may exploit properties of ADMM. Section 7 deals with how to exploit the fact that we will solve similar optimization problems again and again.

The thesis is then concluded by our results in Section 8, and some concluding remarks in Section 9.

2

Optimization Theory

In this section we will introduce some optimization theory relevant to the thesis.

2.1 Quadratic Programming

The basic problem that we will be working with in this thesis is the constrained Quadratic Programming (QP) problem:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & \frac{1}{2} x^T H x + f^T x + c \\ \text{s.t} \quad & A_{\mathcal{E}} x = b_{\mathcal{E}}, \\ & b_{\mathcal{I}l} \leq C_{\mathcal{I}} x \leq b_{\mathcal{I}u}, \end{aligned} \tag{2.1}$$

where H is a positive semidefinite matrix.

All of the problems we are looking at in this thesis will be reduced to solving a series of these QP problems. There are many different ways to solve QP problems, with different properties in terms of convergence and scalability. Common algorithms include active set and interior point methods.

2.1.1 Splitting

The ADMM algorithm (see Section 2.3) can be applied on a split problem on the form

$$\begin{aligned} \min_{x,y} \quad & g(x) + h(y) \\ \text{s.t.} \quad & Cx = y, \end{aligned} \tag{2.2}$$

where g and h are convex functions. We let x in (2.2) correspond exactly to x in (2.1), while y is a new variable introduced with the splitting.

For (2.2) to be equivalent to (2.1), the functions g and h should together model the cost and constraints in (2.1). This could be achieved in many ways, but we will let (2.1) be split into (2.3):

$$\begin{aligned}\min_{x,y} \quad & \underbrace{\frac{1}{2}x^T Hx + f^T x + c + \mathbb{I}(A_{\mathcal{E}}x = b_{\mathcal{E}})}_{g(x)} + \underbrace{\mathbb{I}(y \in \mathcal{Y})}_{h(y)} \\ \text{s.t} \quad & Cx = y,\end{aligned} \tag{2.3}$$

where

$$\mathcal{Y} = \{y : b_{\mathcal{I}l} \leq Ky \leq y_{\mathcal{I}u}\} \tag{2.4}$$

$$C_{\mathcal{I}} = KC, \tag{2.5}$$

for some matrices K and C and where the indicator function $\mathbb{I}(x)$ is simply defined as

$$\mathbb{I}(x) = \begin{cases} 0 & \text{if } x \\ \infty & \text{else.} \end{cases}$$

In the splitting (2.3), the inequality constraints modeled by $h(y)$ are separated from the equality constraints and cost function modeled by $g(x)$. As minimization of g and h will be done separately in ADMM, the absence of inequality constraints in g will be beneficial[1].

The factorization of $C_{\mathcal{I}}$ into K and C may also be done in many ways, and is further discussed in Section 2.4. Specifically, consider the choice $C = C_{\mathcal{I}},\ K = I$, as it makes the constraints $y \in \mathcal{Y}$ element-wise.

2.2 Mixed Integer Quadratic Programming

We define a Mixed Integer Quadratic Program as in (2.6), the only difference to a QP (2.1) being that some elements in x now have to be binary. The number of real and binary elements in x are denoted by n_r and n_b, respectively.

$$\begin{aligned}\min_{x \in \mathbb{R}^{n_r} \times \{0,1\}^{n_b}} \quad & \frac{1}{2}x^T Hx + f^T x + c \\ \text{s.t} \quad & A_{\mathcal{E}}x = b_{\mathcal{E}}, \\ & b_{\mathcal{I}l} \leq C_{\mathcal{I}}x \leq b_{\mathcal{I}u}.\end{aligned} \tag{2.6}$$

Let $x_r \in \mathbb{R}^{n_r}$ denote the real elements, and $x_b \in \{0,1\}^{n_b}$ denote the binary elements in x. This problem is much harder to solve than the QP problem (2.1) and is in general NP hard.

[1] If there were no inequality constraints in the original QP (2.1), it could be solved trivially by solving a KKT system.

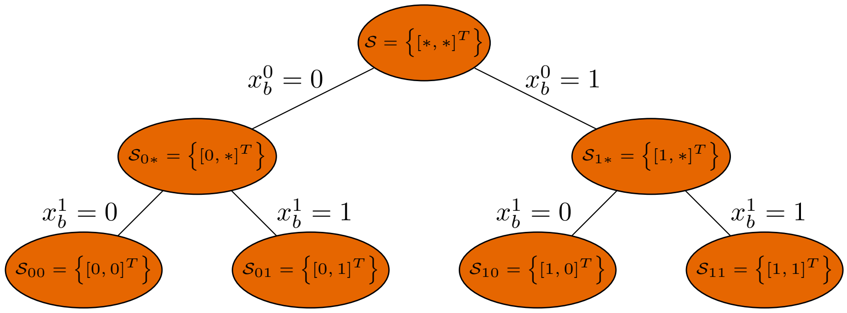

$\mathcal{S} = \left\{[*,*]^T\right\}$
$x_b^0 = 0$
$x_b^0 = 1$
$\mathcal{S}_{0*} = \left\{[0,*]^T\right\}$
$\mathcal{S}_{1*} = \left\{[1,*]^T\right\}$
$x_b^1 = 0$
$x_b^1 = 1$
$x_b^1 = 0$
$x_b^1 = 1$
$\mathcal{S}_{00} = \left\{[0,0]^T\right\}$
$\mathcal{S}_{01} = \left\{[0,1]^T\right\}$
$\mathcal{S}_{10} = \left\{[1,0]^T\right\}$
$\mathcal{S}_{11} = \left\{[1,1]^T\right\}$

Figure 2.1 Illustration of the Branch and Bound tree structure.

2.2.1 Branch and Bound

If we knew the solution of x_b, we could formulate and solve an ordinary QP[2] and also find the solution of x_r. The simplest and most straight-forward approach to solving an MIQP is by brute-force, trying out every possible combination of x_b, and solving the corresponding QPs. Then the solution is determined by which of the QPs give the best optimal function value. The drawback of brute-force however, is that 2^{n_b} QPs have to be solved, giving exponential time complexity.

The main idea of Branch and Bound is to decrease the number of QPs that have to be solved, using a hierarchical tree structure and evaluating the prospect of entire groups of possible x_b, potentially discarding some of these groups.

Let $\mathcal{S} = \{0,1\}^{n_b}$ denote the set of possible x_b in the MIQP (2.6). In Branch and Bound, $\mathcal{S}$ is hierarchically divided into subsets, as illustrated in an example in Figure 2.1. Each of the tree nodes (which we denote N_{XX}) correspond to MIQPs like (2.6), but with $x_b \in \mathcal{S}$ substituted for $x_b \in \mathcal{S}_{XX}$, where $\mathcal{S}_{XX}$ are the corresponding subsets of $\mathcal{S}$.

At the leaf[3] nodes all binary variables are fixed, and every leaf corresponds to a combination of binary variables in the brute-force approach. These are the only nodes that can be solved as QPs directly. All the other nodes are MIQPs, although with fewer unknown binary variables as compared to the root node.

Although only the leaves can be solved as QPs directly, we can still compute lower bounds of the optimal function value for the other nodes, by solving a relaxation of the corresponding MIQP. The solution of such a relaxed problem may very well not be feasible in the non-relaxed problem, but has

[2] This could be formulated in many ways, for instance by adding equality constraints that fix x_b to their known optimal values.

[3] By leaves or leaf nodes we mean the childless nodes at the bottom of the tree, where all binary variables are fixed.

the advantage of being more straight-forward to solve. Different relaxations are discussed later on.

Let J_{ub} denote an upper bound of the optimal objective function value J^* of (2.6). The upper bound J_{ub} is initiated to $J_u = +\infty$, but will be updated with tighter and tighter bounds as corresponding points x_{ub} satisfying the constraints of (2.6) are found. The relation

$$J_{ub} \geq J^* \tag{2.7}$$

will always hold, as we only update J_{ub} with points x_{ub} satisfying the constraints of (2.6).

The Branch and Bound algorithm traverses the tree, and solves relaxations of the MIQPs at the tree nodes. Let x_{N_i} denote the solution to the relaxation of node N_i, with corresponding objective function value $J^*_{N_i}$. In the following three cases we can prune the entire subtree, for which N_i is the root node:

- If the relaxation is infeasible, since then the non-relaxed MIQP of N_i is also infeasible.
- If $J^*_{N_i} \geq J_{ub}$, since in that case no feasible point x in the subtree can give lower objective function value than x_{ub} does[4].
- If x_{N_i} is binary feasible, because then it also solves the non-relaxed MIQP of N_i. If $J^*_{N_i} < J_{ub}$, we also update $x_{ub} \leftarrow x_{N_i}$ and $J_{ub} \leftarrow J^*_{N_i}$.

In any other case, we do not prune the subtree.

Unfortunately Branch and Bound also has exponential time complexity, but typically reduces the number of subproblems to be solved drastically as compared to the brute-force approach.

Implementation We have implemented Branch and Bound with a priority list of tree nodes. The list is initiated by holding a single element – the root node. When another node is to be solved, the node N_i with highest priority is always chosen (slightly confusing – high priorities are represented by small numbers). If the subtree for which N_i is the root node cannot be pruned (and N_i is not a leaf), the children of N_i are added to the priority list. On the contrary, pruning is simply done by not adding the children to the list. When the priority list is empty, the algorithm is done. Then (x_{ub}, J_{ub}) minimize the MIQP (2.6) (or $J_{ub} = +\infty$ if (2.6) is infeasible).

There are different ways of assigning priorities to the nodes, and some choice of priorities may require many more nodes to be traversed than another choice of priorities. This is further discussed in Section 6.1.

[4] If we store the optimal objective function value $J^*_{N_j}$ of the relaxation at the parent N_j of N_i, we may prune if $J^*_{N_j} \geq J_{ub}$ without having to solve the current relaxation, due to the relation $J^*_{N_i} \geq J^*_{N_j}$.

Relaxations As explained in Section 2.2.1, all nodes in the Branch and Bound tree correspond to MIQP problems, except for the leaf nodes which correspond to QP problems since all binary variables are fixed. To be able to apply convex optimization methods, convex relaxations of the non-convex MIQPs need to be considered. There are different ways of relaxing an MIQP, and while considering which one to use there is typically a trade-off between tightness of the resulting bounds, and performance when solving the relaxations.

The QP relaxation is probably the most straight-forward example, as the constraints $x_b \in \{0,1\}^{n_b}$ are simply exchanged for the inequality constraints

$$0 \leq x_b \leq 1. \tag{2.8}$$

This turns the MIQP into a QP, which can be solved by interior point methods, active set methods or first-order methods, among others. In Section 7.1.1, more details on how we formulate a QP relaxation for parametric ADMM will be given.

Other examples of relaxations are Semi-Definite Programming (SDP) and Second-Order Cone Programming (SOCP) relaxations. For the purposes of this thesis however, we have restricted ourselves to the standard QP relaxation.

2.3 ADMM

2.3.1 Lagrange Dual

To explain ADMM in an intuitive way, we first introduce the Lagrange Dual. The optimization problem that we will be solving using ADMM is the QP problem (2.3):

$$\begin{aligned} \min_{x,y} \quad & \underbrace{\frac{1}{2}x^T H x + f^T x + c + \mathbb{I}(A_{\mathcal{E}} x = b_{\mathcal{E}})}_{g(x)} + \underbrace{\mathbb{I}(y \in \mathcal{Y})}_{h(y)} \\ \text{s.t} \quad & Cx = y, \end{aligned}$$

and the Lagrangian for this problem can be written as

$$L(x,y,\lambda) = g(x) + h(y) + \lambda^T (Cx - y). \tag{2.9}$$

From this function it is possible to define the dual problem

$$\max_{\lambda} \theta(\lambda) = \max_{\lambda} \inf_{x,y} L(x,y,\lambda). \tag{2.10}$$

The dual problem (2.10) has the nice property that its optimal value is the same as the one for the primal problem (2.3) and that (x^*, y^*) is a solution to the primal (2.3) if and only if (x^*, y^*, λ^*) is the solution to dual. This opens up the possibility of solving the dual problem instead of the primal, which is exactly what ADMM is doing.

However, before we attempt to solve the dual problem, we will first introduce the *augmented Lagrangian*

$$L_\rho(x, y, \lambda) = g(x) + h(y) + \lambda^T (Cx - y) + \frac{\rho}{2} \|Cx - y\|^2 , \tag{2.11}$$

where ρ is a scaling factor. The associated dual problem to this augmented Lagrangian is

$$\max_\lambda \theta_\rho(\lambda) = \max_\lambda \inf_{x,y} L_\rho(x, y, \lambda). \tag{2.12}$$

This extra term will not change the optimal value or solution, since $Cx = y$ at any feasible point. However, it has better robustness properties and will let the solver converge under fewer assumptions [Boyd et al., 2011]. We will thus be solving this problem instead.

2.3.2 Method of Multipliers

It is now tempting to apply a simple gradient ascent method to the function $\theta_\rho(\lambda)$. Assume that for a fix λ^k we can compute $(x^k, y^k) = \operatorname{argmin}_{x,y} L_\rho(x, y, \lambda^k)$. It is then possible to calculate the gradient $\nabla\theta_\rho(\lambda) = Cx^k - y^k$, which inspires us to the *method of multipliers* algorithm

$$\begin{aligned} (x^k, y^k) &= \operatorname*{argmin}_{x,y} L_\rho(x, y, \lambda^{k-1}) \\ \lambda^k &= \lambda^{k-1} + \rho(Cx^k - y^k). \end{aligned} \tag{2.13}$$

It is not a coincidence that the step length here is chosen to ρ. It can be shown (for differentiable functions) that this choice will result in dual feasibility of (x^k, y^k, λ^k), i.e. $\nabla_x L_\rho(x, y^k, \lambda^k) = 0$ and $\nabla_y L_\rho(x^k, y, \lambda^k) = 0$. We refer the reader to [Boyd et al., 2011] for a slightly more thorough introduction.

2.3.3 ADMM

Although the *method of multipliers* has good convergence properties, it is hard to implement in practice. It requires that we are able to minimize L_ρ over x and y simultaneously, which is difficult in general. We thus introduce ADMM[5], where the minimization is done in x and y separately:

$$x^k = \operatorname*{argmin}_x L_\rho(x, y^{k-1}, \lambda^{k-1}) \tag{2.14a}$$

$$y^k = \operatorname*{argmin}_y L_\rho(x^k, y, \lambda^{k-1}) \tag{2.14b}$$

$$\lambda^k = \lambda^{k-1} + \rho(Cx^k - y^k). \tag{2.14c}$$

[5] The "alternating directions" in the Alternating Direction Method of Multipliers refer to the alternating updates (2.14a) and (2.14b) of x and y.

There are several convergence results for ADMM, and we refer the reader once again to [Boyd et al., 2011] for a more rigorous explanation of ADMM with references to many of the relevant articles on this method.

We will now focus on how ADMM can be used to solve our specific problem. We note that the first step is independent of $h(y)$ and the second of $g(x)$, a very important property for it to be possible to do the minimizations efficiently. We can express the steps in the algorithm as

$$\begin{aligned}
x^k &= \underset{x}{\operatorname{argmin}} \left\{ g(x) + {\lambda^{k-1}}^T Cx + \frac{\rho}{2} \left\| Cx - y^{k-1} \right\|^2 \right\} \\
&= \underset{x}{\operatorname{argmin}} \left\{ g(x) + \frac{\rho}{2} \left\| Cx - y^{k-1} + \frac{1}{\rho}\lambda^{k-1} \right\|^2 \right\} && \text{(2.15a)} \\
y^k &= \underset{y}{\operatorname{argmin}} \left\{ h(y) - {\lambda^{k-1}}^T y + \frac{\rho}{2} \left\| Cx^k - y \right\|^2 \right\} \\
&= \underset{y}{\operatorname{argmin}} \left\{ h(y) + \frac{\rho}{2} \left\| Cx^k - y + \frac{1}{\rho}\lambda^{k-1} \right\|^2 \right\} && \text{(2.15b)} \\
\lambda^k &= \lambda^{k-1} + \rho(Cx^k - y_k). && \text{(2.15c)}
\end{aligned}$$

If we look at the functions we chose as $g(x)$ and $h(x)$:

$$\begin{aligned}
g(x) &= \frac{1}{2}x^T Hx + f^T x + c + \mathbb{I}(A_{\mathcal{E}}x = b_{\mathcal{E}}) \\
h(y) &= \mathbb{I}(y \in \mathcal{Y}),
\end{aligned} \tag{2.16}$$

we see that the first minimization is a convex quadratic problem under equality but no inequality constraints, and that the second can be seen as minimizing the distance of y to $Cx^k + \frac{1}{\rho}{\lambda^{k-1}}^T$, while keeping $y \in \mathcal{Y}$.

The first minimization (2.15a) can thus be done by solving the KKT system

$$\begin{bmatrix} H + \rho C^T C & A_{\mathcal{E}}^T \\ A_{\mathcal{E}} & 0 \end{bmatrix} \begin{bmatrix} x^k \\ \mu \end{bmatrix} = \begin{bmatrix} -f - C^T\left(\lambda^{k-1} - \rho y^{k-1}\right) \\ b_{\mathcal{E}} \end{bmatrix}. \tag{2.17}$$

Solving this can be done by first computing a LDL factorization of the matrix, and then a relatively cheap backsolve. It is important to note that this matrix is completely independent on the current iterates (x^k, y^k, λ^k), and the factorization can thus be done once per problem, and be used for all iterations! Moreover, the factorization is independent on the inequality constraints $\mathcal{Y}$ and the right hand side of the equality constraints $b_{\mathcal{E}}$. This makes it possible to use the same factorization for almost all the problems we are solving, which is one of the main reasons why we are considering ADMM in this thesis.

The second minimization (2.15b) can be expressed as

$$\begin{aligned} \min_{y} \quad & \left\| Cx^k - y + \frac{1}{\rho}\lambda^{k-1} \right\|^2 \\ \text{s.t} \quad & b_{\mathcal{I}l} \leq Ky \leq b_{\mathcal{I}u}, \end{aligned} \tag{2.18}$$

or equivalently as a projection of $Cx^k + \frac{1}{\rho}{\lambda^{k-1}}^T$ onto $\mathcal{Y}$. This is in general a non-trivial problem. However, in the common case of $K = I, C = C_{\mathcal{I}}$, it is much easier and (2.18) can be explicitly solved by

$$y^k = \underset{b_{\mathcal{I}l}, b_{\mathcal{I}u}}{\text{clip}} \left\{ C_{\mathcal{I}} x^k + \frac{1}{\rho}\lambda^{k-1} \right\}, \tag{2.19}$$

where $y = \text{clip}_{a,b}\{z\}$ is defined as

$$y_i = \begin{cases} a_i & \text{if} \quad z_i < a_i \\ b_i & \text{if} \quad z_i > b_i \\ z_i & \text{else} \end{cases}, \tag{2.20}$$

for each index i in z. A more thorough discussion of the choice of K and C follows in Section 2.4.

2.3.4 Optimality condition

In this section we will present a brief description of the optimality criterion that is used both as a stopping criterion for ADMM as well as a basis for the cost function in the line search in Section 5.5. If the Lagrangian (2.9) was differentiable, optimality of (x^*, y^*, λ^*) for the problem (2.3) would be equivalent to it being saddle point of $L(x, y, \lambda) = g(x) + h(y) + \lambda^T(Cx - y)$, i.e.

$$\begin{aligned} \nabla_x L(x^*, y^*, \lambda^*) &= 0 \\ \nabla_y L(x^*, y^*, \lambda^*) &= 0 \\ \nabla_\lambda L(x^*, y^*, \lambda^*) &= 0. \end{aligned}$$

However, since the gradient is not defined everywhere for $g(x)$ and $h(x)$, we must use *subgradients* instead. Subgradients can be seen as an extension of gradients to non-differentiable convex functions. We say that g is a subgradient to f at x if $f(y) \geq f(x) + g^T(y - x)$ for all y, and denote the *subdifferential*, the set of all subgradients, as ∂f. In the case where f is differentiable, the gradient will be the only subgradient and $\partial f = \{\nabla f\}$. The optimality conditions can now be shown to be

$$\begin{aligned} 0 &\in \partial_x L(x^*, y^*, \lambda^*) \\ 0 &\in \partial_y L(x^*, y^*, \lambda^*) \\ 0 &= \nabla_\lambda L(x^*, y^*, \lambda^*). \end{aligned} \tag{2.21}$$

The two first are referred to as dual feasibility, and require that x^* and y^* are minimizers for a fix λ, and the last is called primal feasibility, which requires $Cx = y$.

2.4 Choosing K and C

Recall the factorization $C_{\mathcal{I}} = KC$ in Section 2.1.1. A straight-forward choice of this factorization is to let

$$K = C_{\mathcal{I}}, \quad C = I. \tag{2.22}$$

The update step (2.18) for y however, is not trivial to solve. We can instead let

$$K = I, \quad C = C_{\mathcal{I}} \tag{2.23}$$

which, as mentioned previously, separates (2.18) into each of the coordinates, and may be solved by the simple clip operation (2.19).

In Section 2.4.1, some drawbacks with (2.23) are discussed. As it is quite involved, we want to point out that while it may give some intuition of the relation between the x and y variables, it is not crucial to grasp for the report as a whole. The main point is that while (2.23) has computational benefits, it also has drawbacks when compared to (2.22).

2.4.1 Drawbacks of $K = I$

While (2.23) has a great computational advantage, the approach also has several drawbacks. One drawback is related to preconditioning, and will be discussed further in Section 5.4.1, where a compromise between (2.22) and (2.23) will also be suggested. Some other drawbacks of (2.23) will be illustrated in the following.

Let $c_i^T, \quad i = 1, \ldots, m$ denote the rows of $C \in \mathbb{R}^{m\times n}$. Then each constraint $y_i = c_i^T x$ corresponds to a hyperplane in the domain of x (i.e. $\mathbb{R}^n$) with c_i being the corresponding normal vectors. Now let x_y (loosely) denote the point(s) fulfilling $y = Cx_y$ for a particular y update of (2.15b). The point(s) x_y may then be seen as a (never carried out) x update suggested by the y update.

The mapping $y = Cx$ is bijective if and only if C is invertible (which of course is the case for (2.22), but not necessarily for (2.23)). If not, there may be several points x_y, or none at all, that fulfill $y = Cx_y$ for a particular y. These scenarios are illustrated in Figures 2.2 and 2.3.

When C is invertible there is no ambiguity in x_y, but the y update may represent a questionable point x_y, as illustrated in Figure 2.4. As the clip operation will be applied to y_2, the infeasible point x_1 will be shifted along the line $y_1 = c_1^T x$, to a feasible point $x_f = x_2$. However, we would prefer

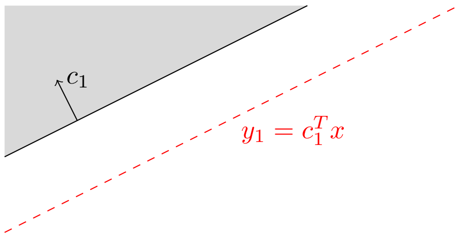


Figure 2.2 $C \in \mathbb{R}^{1\times 2}$, and is thus not invertible. One element y_1 in y, corresponds to the entire dashed line, and hence to many possible points x_y. The infeasible region is marked with gray background.

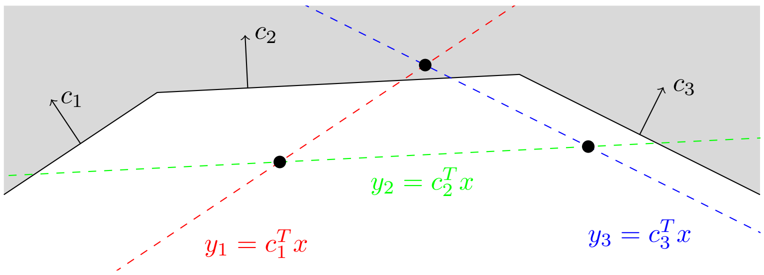


Figure 2.3 $C \in \mathbb{R}^{3\times 2}$, and is thus not invertible. Each element y_i in y, corresponds to one of the dashed lines, but apparently there is no point x_y at which all of the lines intersect. The infeasible region is marked with gray background.

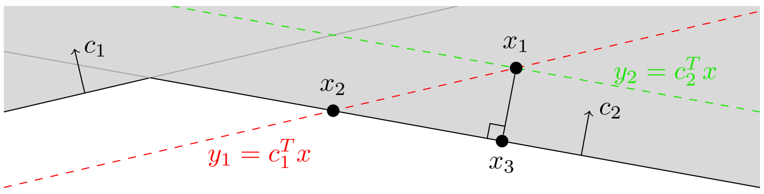


Figure 2.4 $C \in \mathbb{R}^{2\times 2}$, and is invertible. For every point y, there is a unique point x_y determined by the intersection of the lines $y_1 = c_1^T x_b$ and $y_2 = c_2^T x_b$. The infeasible region is marked with gray background.

a point x_f close to x_1, as x_1 was in fact selected due to its low objective function value in (2.15a). Considering this, $x_f = x_3$ would in fact be a better choice than $x_f = x_2$, which points out a drawback of the approach (2.23). If instead the rows c_1 and c_2 of C were orthogonal, this would not be an issue.

3

Model Predictive Control

3.1 Unconstrained MPC for Linear Systems

Considering MPC of linear systems with unconstrained states and inputs, we can write the optimization problem as in (3.1).

$$\begin{aligned} \min_{x} \quad & \sum_{k=0}^{N-1} \left(e_k^T Q e_k\right) + e_N^T Q_N e_N \\ \text{s.t} \quad & x_{k+1} = A x_k + B u_k, \end{aligned} \tag{3.1}$$

where e is the state and input error from some reference trajectories r_x, r_u (possibly zero):

$$e_k = \begin{bmatrix} x_k - r_x^k \\ u_k - r_u^k \end{bmatrix}, \quad k = 1, \ldots, N \tag{3.2}$$

Typically the cost matrix is on the form

$$Q = \begin{bmatrix} Q_x & 0 \\ 0 & Q_u \end{bmatrix}$$

and the state and input cost matrices Q_x and Q_u are often diagonal. It is also usually required that Q_x is positive semi-definite and Q_u positive definite. This is the common Linear Quadratic Regulator problem (LQR) and the optimal (linear feedback) regulator is found by solving the discrete time algebraic Riccati equation.

3.2 Constrained MPC for Linear Systems

A more general case of MPC, is when we have added variable constraints as in (3.3).

$$\begin{aligned} \min_{x} \quad & \sum_{k=0}^{N-1} \left(e_k^T Q e_k\right) + e_N^T Q_N e_N \\ \text{s.t} \quad & x_{k+1} = A x_k + B u_k. \\ & l_x \leq x_k \leq u_x \\ & l_u \leq u_k \leq u_u \end{aligned} \tag{3.3}$$

This problem however, gives no explicit feedback rule. Instead a Quadratic Program on the form (2.1) has to be solved at each sample.

3.3 Mixed Integer Predictive Control

When applying MPC to hybrid systems, the term Mixed Integer Predictive Control (MIPC) is often used. MIPC is however just a special case of MPC.

The way of modeling hybrid systems is not obvious. Several possible representations exist, modeling different flavors of hybrid systems but in many cases overlapping. One of these representations – Mixed Logical Dynamical Systems (MLD), will be considered in the following.

3.3.1 Mixed Logical Dynamical Systems

MLD systems can be used to model several types of hybrid systems[1], and MPC of such systems have the benefit of being easily converted to MIQPs. This procedure is illustrated in Sections 3.3.3 and Appendix A. We define an MLD system as

$$\begin{aligned} x_{k+1} &= A x_k + B_u u_k + B_{aux} w_k + B_{aff} \\ &\begin{cases} E_x^{eq} x_k + E_u^{eq} u_k + E_{aux}^{eq} w_k = E_{aff}^{eq} \\ E_x^{ineq} x_k + E_u^{ineq} u_k + E_{aux}^{ineq} w_k \leq E_{aff}^{ineq} \end{cases} \\ &\begin{cases} l_x \leq x_k \leq u_x \\ l_u \leq u_k \leq u_u \\ l_w \leq w_k \leq u_w \end{cases} \end{aligned} \tag{3.4}$$

where the vectors x_k, u_k and w_k denote states, inputs and auxiliary variables, respectively. All of these may have some real elements, and some binary.

[1] Not all hybrid systems can be written as MLD systems. For example the state update equation needs to be linear.

3.3.2 Dynamic Modes

One kind of hybrid systems may switch between different dynamics, or so called modes. An example of this is *piece-wise affine systems*, for which different modes are enabled in different regions of the state space. Another example is *switched affine systems*, where the mode is selected by control input(s).

Both *piece-wise affine systems* and *switched affine systems* can be modeled as MLD systems, and we will illustrate the procedutre for the latter. Consider a system with one real state x and one real input u_r. In addition, a binary input u_b toggles which of two possible dynamics will be active. This is formalized in (3.5).

$$x(k+1) = A_{u_b(k)}x(k) + B_{u_b(k)}u_r(k) \tag{3.5}$$

If we now introduce some auxiliary variables (z_x, z_u) fulfilling (3.6), the system (3.5) can be reformulated according to (3.7).

$$\begin{cases} z_x(k) = u_b(k)x(k) \\ z_u(k) = u_b(k)u_r(k) \end{cases} \tag{3.6}$$

$$\begin{aligned} x(k+1) = {} & A_0x(k) + B_0u_r(k) \\ & + (A_1 - A_0)\, z_x(k) + (B_1 - B_0)\, z_u(k) \end{aligned} \tag{3.7}$$

Consider the QP relaxation described in Section 2.2.1. As u_b then belongs to the convex set $[0,1]$, we may speak of convexity of constraints on u_b. Unfortunately, (3.6) will then be a non-convex constraint. However, the so called "big M" formulation (3.8)-(3.9) may be used instead of the constraint $z_x(k) = u_b(k)x(k)$.

$$m_x u_b(k) \leq z_x(k) \leq M_x u_b(k) \tag{3.8}$$

$$m_x(1 - u_b(k)) \leq x(k) - z_x(k) \leq M_x(1 - u_b(k)) \tag{3.9}$$

Let the constants m_x and M_x be selected such that $m_x \leq x(k) \leq M_x$ is always true. When $u_b(k) = 0$, (3.8) will be equivalent to $z_x(k) = 0$ and (3.9) will be redundant. When instead $u_b(k) = 1$, (3.9) will be equivalent to $z_x(k) = x(k)$ and (3.8) will be redundant. Thus, $z_x(k) = u_b(k)x(k)$ if $u_b(k) \in \{0,1\}$. Of course, similar (convex) inequalities can replace the constraint $z_u(k) = u_b(k)u_r(k)$, and (3.6) will be fulfilled for binary $u_b(k)$. However, if $u_b(k) \notin \{0,1\}$, (3.6) may not hold.

The entire *switched affine system* can now be modeled on the MLD form

(3.4) according to (3.10).

$$x = x, \quad u = [u_r, u_b]^T, \quad w = [z_x, z_u]^T \tag{3.10a}$$

$$A = A_0, \quad B_u = \begin{bmatrix} B_0 & 0 \end{bmatrix}, \quad B_{aux} = \begin{bmatrix} A_1 - A_0 & B_1 - B_0 \end{bmatrix}, \quad B_{aff} = 0 \tag{3.10b}$$

$$\left[\, E_x^{ineq} \;\middle|\; E_u^{ineq} \;\middle|\; E_w^{ineq} \,\right] = \left[\begin{array}{c|c|c|cc} 1 & 0 & M_x & -1 & 0 \\ -1 & 0 & -m_x & 1 & 0 \\ 0 & 0 & m_x & -1 & 0 \\ 0 & 0 & -M_x & 1 & 0 \\ 0 & 1 & M_u & 0 & -1 \\ 0 & -1 & -m_u & 0 & 1 \\ 0 & 0 & m_u & 0 & -1 \\ 0 & 0 & -M_u & 0 & 1 \end{array}\right] \tag{3.10c}$$

$$E_{aff}^{ineq} = \begin{bmatrix} M_x \\ -m_x \\ 0 \\ 0 \\ M_u \\ -m_u \\ 0 \\ 0 \end{bmatrix} \tag{3.10d}$$

The equality constraints E_*^{eq} are not present in this example, and the lower/upper bounds in (3.4) may be selected as desired. In (3.10c)-(3.10d), the green elements constitute the "big M" formulation for z_x, while the red elements correspond to z_u. This coloring will be helpful when referring from Section 5.4.1. If we would consider a larger system, additional such blocks of four lines would be incorporated for the corresponding additional states or inputs.

If we would instead let the modes correspond to different parts of the state space, (3.5) would turn into a *piece-wise affine system*. However, the MLD modeling would be almost the same. The binary input u_b could be replaced by a binary auxiliary variable δ, and inequality constraints could let the state determine if $\delta = 0$ or $\delta = 1$. How to formulate such constraints[2] is described in [Mignone et al., 1999].

If we desire more than two modes, this could be achieved by introducing more binary variables. However, we will not go through the details of this procedure.

[2] Many other logical relationships are also converted into linear (in)equality constraints in this article.

3.3.3 MIPC Formulation

While the MLD system (3.4) can describe hybrid system dynamics and variable constraints, we also need a cost function to perform mixed integer predictive control. Let the final MIPC problem be defined as

$$\begin{aligned} \min_{x} \quad & \sum_{k=0}^{N-1} \left(e_k^T Q e_k\right) + e_N^T Q_N e_N \\ \text{s.t} \quad & (3.4), \end{aligned} \tag{3.11}$$

where e is the error for states, inputs and auxiliary variables, from some reference trajectories r_x, r_u, r_w (possibly zero):

$$e_k = \begin{bmatrix} x_k - r_x^k \\ u_k - r_u^k \\ w_k - r_w^k \end{bmatrix}, \quad k = 1, \ldots, N \tag{3.12}$$

Similarly to the non-hybrid case, the cost matrix is typically on the form

$$Q = \begin{bmatrix} Q_x & 0 & 0 \\ 0 & Q_u & 0 \\ 0 & 0 & Q_w \end{bmatrix}$$

where Q_x, Q_u and Q_w are often diagonal.

The MIPC problem (3.11) can now be formulated as an MIQP of the form (2.6) (see Section A).

4

Test Cases

4.1 Spring-Damper

Our first example is inspired by a classic linear spring-damper system, characterized by

$$x(k+1) = A_0 x(k) + B_0 u_1(k). \tag{4.1}$$

The system matrices A_0 and B_0 are given in Section C.1, together with other details for this section. The states x_1 and x_3 are positions of two masses m_1 and m_2, along a direction in which they are both free to move. Their corresponding velocities are x_2 and x_4. The masses are interconnected by a spring with spring constant $k = 400\,\mathrm{N/m}$, and they are also damped with corresponding damping coefficients d_1 and d_2. There is one input u_1, which is a force on the first mass.

Inspired by the linear system (4.1), we define our hybrid spring-damper system as

$$x(k+1) = A_{u_2(k)} x(k) + B_{u_2(k)} u_1(k) + B_b u_3(k) \tag{4.2a}$$

$$-10 \leq x_1, x_3 \leq 10 \tag{4.2b}$$

$$-100 \leq x_2, x_4 \leq 100 \tag{4.2c}$$

$$-1 \leq \quad u_1 \quad \leq 1 \tag{4.2d}$$

where the states $x(k)$ and input force $u_1(k) \in \mathbb{R}$ are the same as before. The binary input $u_2 \in \{0, 1\}$ is a mode switching variable, switching between the dynamics given by A_0, B_0 (same as before) and A_1, B_1 (see Section (C.1)). The only difference between the modes is that the spring constant is decreased to $k = 10\,\mathrm{N/m}$ for A_1, B_1. A binary input force $u_3(k)$ is also exerted on the first mass, by the additional matrix B_b, which is the same for both modes.

Except for the presence of B_b and $u_3(k)$, (4.2) is on the *switched affine system* form (3.5). It is straightforward to write the system on MLD form, by following (3.10) and simply extending B_u with B_b in (3.10b). The "big M" values m and M are taken from the corresponding lower and upper bounds in (4.2).

The cost matrices when applying MIPC to this system are stated in Section C.1.

4.2 Turbo Car

The following example is provided with the Hysdel modeling language [Torrisi and Bemporad, 2004], which lets the user specify hybrid systems with an intuitive high-level language. An MLD system representation is then generated automatically.

Consider a simple (one-dimensional) model of a turbo car. There is one real input a which accelerates the car, but there is also one binary input t which if enabled turns on the turbo and doubles the acceleration a. The states consist of the position p, velocity v and a turbo count c. The turbo count is initiated to 5, but decreases by one every time the turbo is used. Due to a constraint $0 \leq c \leq 5$, we may not use the turbo at more than 5 samples. The dynamics can be formalized by

$$\begin{cases} p(k+1) = p(k) + v(k) + z(k) \\ v(k+1) = v(k) + 0.5 \cdot z(k) \\ c(k+1) = c(k) - t(k), \end{cases} \tag{4.3a}$$

where z is an auxiliary variable defined by

$$z(k) = \begin{cases} a(k) & \text{if } t(k) = 0 \\ 2a(k) & \text{if } t(k) = 1. \end{cases} \tag{4.3b}$$

In addition we have the constraints

$$\begin{bmatrix} -50 \\ -10 \\ 0 \end{bmatrix} \leq \begin{bmatrix} p \\ v \\ c \end{bmatrix} \leq \begin{bmatrix} 50 \\ 10 \\ 5 \end{bmatrix}. \tag{4.3c}$$

An MLD system (4.4) is then generated by Hysdel (the matrices are defined in Section C.2):

$$\begin{aligned} x_{k+1} &= Ax_k + B_u u_k + B_{aux} w_k + B_{aff} \\ &\begin{cases} E_x^{eq} x_k + E_u^{eq} u_k + E_{aux}^{eq} w_k = E_{aff}^{eq} \\ E_x^{ineq} x_k + E_u^{ineq} u_k + E_{aux}^{ineq} w_k \leq E_{aff}^{ineq} \end{cases} \\ &\begin{cases} l_x \leq x_k \leq u_x \\ l_u \leq u_k \leq u_u \\ l_w \leq w_k \leq u_w \end{cases} \end{aligned} \tag{4.4}$$

Although (4.3b) is not identical to (3.6), it results in inequalities (C.1b) similar to our "big M" formulation (3.10c)-(3.10d)

The cost matrices when applying MIPC to this system are stated in Section C.2.

5

Improving ADMM

The purpose of this Section is to point out ideas and implementations of ours, which improve ADMM or makes it fit for our purposes better. The main contributions of the entire thesis are given in Sections 5.4.1 and 5.5.

5.1 ADMM Implementation

ADMM is implemented in QPgen[Giselsson, 2015], a Matlab[MATLAB, 2014] toolbox which generates C code for solving optimization problems on the form (2.3) and similar forms. QPgen is based on the following articles: [Giselsson and Boyd, 2014a; Giselsson, 2014; Giselsson and Boyd, 2014c; Giselsson and Boyd, 2014b].

The QPgen implementation does however differ slightly from the presentation in Section (2.14). First of all Cx^k in (2.15b) is exchanged for an auxiliary variable

$$z^k = \alpha C x^k + (1-\alpha) y^{k-1}, \tag{5.1}$$

where $\alpha \in (0,2)$ is called a *relaxation parameter*. According to [Boyd et al., 2011], experiments have indicated that $\alpha \in [1.5, 1.8]$ can improve convergence. We have selected the parameter to $\alpha = 1.75$.

In addition to this, the implementation includes a diagonal preconditioning matrix E (with elements $E_{ii} > 0$), which may be used to scale an ill-conditioned problem for improved convergence.

In QPgen the iterations (5.2) replace (2.14).

$$x^k = \underset{x}{\operatorname{argmin}} \left\{ g(x) + \frac{\rho}{2} \left\| ECx - Ey^{k-1} + \frac{1}{\sqrt{\rho}} \lambda_E^{k-1} \right\|^2 \right\} \tag{5.2a}$$

$$y^k = \underset{y}{\operatorname{argmin}} \left\{ h(y) + \frac{\rho}{2} \left\| Ez^k - Ey + \frac{1}{\sqrt{\rho}} \lambda_E^{k-1} \right\|^2 \right\} \tag{5.2b}$$

$$\lambda_E^k = \lambda_E^{k-1} + \sqrt{\rho}(ECx^k - Ey^k). \tag{5.2c}$$

Note that the dual variable λ (which is the multiplier for the constraint $Cx = y$) is exchanged for a new variable λ_E. The latter can be interpreted as the multiplier for $\sqrt{\rho}ECx = \sqrt{\rho}Ey$, and the two are related by[1] $\lambda = \sqrt{\rho}E\lambda_E$.

The KKT system (2.17) for minimization w.r.t x is exchanged for (5.3):

$$\begin{bmatrix} H + \rho C^T E^T E C & A_{\mathcal{E}}^T \\ A_{\mathcal{E}} & 0 \end{bmatrix} \begin{bmatrix} x^k \\ \mu \end{bmatrix} = \begin{bmatrix} -f - \sqrt{\rho} C^T E^T \left(\lambda_E^{k-1} - \sqrt{\rho} E y^{k-1}\right) \\ b_{\mathcal{E}} \end{bmatrix}. \tag{5.3}$$

Recalling the auxiliary variable (5.1), the minimization (2.18) w.r.t. y is replaced by (5.4).

$$\begin{aligned} \min_{y} \quad & \left\| \sqrt{\rho} E (z^k - y) + \lambda_E^{k-1} \right\|^2 \\ \text{s.t} \quad & b_{\mathcal{I}l} \leq K y \leq b_{\mathcal{I}u} \end{aligned} \tag{5.4}$$

The minimization can still be carried out by a simple clip operation if $K = I$, but (2.19) is exchanged for (5.5).

$$y^k = (\sqrt{\rho}E)^{-1} \underset{\sqrt{\rho}Eb_{\mathcal{I}l}, \sqrt{\rho}Eb_{\mathcal{I}u}}{\text{clip}} \left\{ \sqrt{\rho} E z^k + \lambda_E^{k-1} \right\}, \tag{5.5}$$

For debugging, we have also implemented the QPgen version of ADMM in Matlab. The two implementations are meant to be identical, but have diverged to some degree when either one or the other has been extended with additional features of ours.

5.2 Infeasibility Check

Ability to detect infeasibility of a QP relaxation is crucial in Branch and Bound, since it allows for pruning of the entire subtree of the corresponding node. Since QPgen lacked this feature at the start of this thesis, we set out for implementing an infeasibility check based on [Raghunathan and Di Cairano, 2014]. The article proposed the following four conditions for detection of

[1] The relation holds for the optimal variables λ^* and λ_E^*, but not for every iteration (the point with preconditioning is to search for the minimum in another, hopefully faster way).

infeasibility[2]:

$$\max(\|\sqrt{\rho}E(y^k - y^{k-1})\|, \|(\sqrt{\rho}E)^{-1}(\lambda_E^k - \lambda_E^{k-1})\|) > \varepsilon_0 \tag{5.6a}$$

$$\frac{\max(\|Cx^k - Cx^{k-1}\|, \|\sqrt{\rho}E(y^k - y^{k-1})\|)}{\max(\|\sqrt{\rho}E(y^k - y^{k-1})\|, \|(\sqrt{\rho}E)^{-1}(\lambda_E^k - \lambda_E^{k-1})\|)} \leq \varepsilon_r \tag{5.6b}$$

$$\frac{\left((\sqrt{\rho}E)^{-1}\lambda_E^k\right)^T (Cx^k - y^k)}{\|(\sqrt{\rho}E)^{-1}\lambda_E^k\| \, \|Cx^k - y^k\|} \geq 1 - \varepsilon_a \tag{5.6c}$$

$$(\sqrt{\rho}E)^{-1}\lambda_E^k \circ (Cx^k - y^k) \geq 0 \text{ or } \frac{v^k - 2v^{k-1} + v^{k-2}}{\|v^k\|} \leq \varepsilon_v \tag{5.6d}$$

Here $\circ$ denotes element-wise product of vectors or matrices (the Hadamard product). Only one of the conditions in (5.6d), but all of the others need to be fulfilled for triggering infeasibility. The numerator in the second condition in (5.6d) is a finite difference approximation of the second derivative of v, where $v^k = \sqrt{\rho}Ez^k + \lambda_E^{k-1}$ is the argument to the clip operator, in the update (5.5) of y. The tolerances are selected to the proposed values in [Raghunathan and Di Cairano, 2014]: $\varepsilon_0 = 10^{-6}, \varepsilon_r = 10^{-3}, \varepsilon_a = 10^{-3}, \varepsilon_v = 10^{-4}$.

While the conditions (5.6a)-(5.6d) have indeed proven able to detect infeasible problems[3], they sometimes triggered infeasibility even for feasible problems. In an attempt to decrease the risk of false infeasibility detection, we extended conditions (5.6) with conditions (5.7).

$$\frac{\|Cx^k - Cx^{k-1}\|}{\|Cx^k\|} < \varepsilon_{Cx} \tag{5.7a}$$

$$\frac{\|y^k - y^{k-1}\|}{\|y^k\|} < \varepsilon_y \tag{5.7b}$$

The tolerances are selected to $\varepsilon_{Cx} = \varepsilon_y = 10^{-8}$. After adding the extra conditions, we do not detect infeasibility mistakenly as often as before. However, the infeasibility check is not entirely satisfactory as we do still observe false alarms occasionally. Due to this, we have been forced to avoid certain problems.

5.3 Early Termination

Imagine a scenario where ADMM is working on a relaxation of a node in the Branch and Bound tree. If the relaxation has an optimal value[4] which is

[2] We have slightly modified the conditions to account for preconditioning, and accounted for the possibility of $C \neq I$.

[3] For almost feasible problems, more iterations are needed before infeasibility is detected.

higher than the current global upper bound of the MIQP, it should be pruned as described in Section 2.2.1. Therefore, we have no interest in finding a solution to the relaxation, but solely to detecting that it should be pruned and in such case terminate ADMM instead of waiting for convergence.

This section describes two methods for computing lower bounds on the optimal objective function value. If one of them is larger than the global upper bound J_{ub} of the MIQP, ADMM will simply be stopped.

Of course this method relies on having found a good enough global upper bound. The exact same requirement will be needed for the Suboptimal Branch and Bound in Section 6.1.1, where more details about how to achieve this are given.

5.3.1 Evaluating the Dual

One way to compute a lower bound on the optimal primal function value J^*, is to evaluate the dual function $\theta(\lambda) = \inf_{x,y} L(x, y, \lambda)$, where $L(x, y, \lambda)$ is the non-regularized Lagrangian function, defined in (2.9). The dual function will fulfill $\theta(\lambda) \leq J^*$ for any λ, with $\theta(\lambda^*) = J^*$ for an optimizing dual variable λ^*.

Since ADMM continuously updates λ while searching for λ^*, the iterates λ^k will probably give tighter and tighter bounds $\theta(\lambda^k)$. Therefore evaluating $\theta(\lambda^k)$ for the current iterate seems to be a reasonable way of computing a lower bound on J^*.

Since $L(x, y, \lambda)$ is separable in x and y, we define the parts

$$\begin{cases} L_x(x, \lambda) = g(x) + \lambda^T Cx = \frac{1}{2}x^T Hx + (f + C^T\lambda)^T x + \mathbb{I}(A_{\mathcal{E}}x = b_{\mathcal{E}}) \\ L_y(y, \lambda) = h(y) - \lambda^T y = -\lambda^T y + \mathbb{I}(y \in \mathcal{Y}) \end{cases}$$

fulfilling $L(x, y, \lambda) = L_x(x, \lambda) + L_y(y, \lambda)$. Then $\theta(\lambda) = \theta_x(\lambda) + \theta_y(\lambda)$, where

$$\begin{cases} \theta_x(\lambda) = \inf\limits_x L_x(x, \lambda) \\ \theta_y(\lambda) = \inf\limits_y L_y(y, \lambda). \end{cases}$$

Considering evaluation of $\theta_x(\lambda^k)$, we want to minimize a QP in x, with equality constraints but without inequality constraints. This can be done by finding a KKT point, which is a solution to the system of equations (2.17) (with $\rho = 0$)[5]. However the left-hand-side matrix in (2.17) will not be invertible when H is positive semi-definite but not positive definite on $\mathrm{Ker}(A_{\mathcal{E}})$. Since we are only interested in the value of $\theta_x(\lambda^k)$, any KKT point[6] x suffice since they will all give the same value of $L_x(x, \lambda^k)$. Although any

[4] We consider the optimal value of an infeasible relaxation to be ∞, and want to prune subtrees of such nodes as well.

[5] Of course, we insert the latest λ iterate we have.

[6] If H is only semi-definite, there may be several points x minimizing the QP (2.1).

solution fits our purpose, for some λ^k there may actually not be any points x fulfilling the KKT conditions (2.17). In this situation $L_x(x,\lambda)$ and thus $L(x,y,\lambda)$ is unbounded from below w.r.t x for $\lambda = \lambda^k$.

In case we do find a KKT point x, we also want to evaluate $\theta_y(\lambda^k)$, which corresponds to solving the linear program (5.8):

$$\begin{aligned} \min_{y} \quad & -\lambda^T y \\ \text{s.t} \quad & b_{\mathcal{I}l} \leq Ky \leq b_{\mathcal{I}u} \end{aligned} \tag{5.8}$$

In the common case $K = I$ this can be reduced to minimization of each coordinate separately, giving three cases:

$$\begin{cases} \text{if } \lambda_i^k < 0, & y_i = b_{\mathcal{I}l,i} & \implies L_y(y,\lambda^k)_i = -\lambda_i^k b_{\mathcal{I}l,i} \\ \text{if } \lambda_i^k > 0, & y_i = b_{\mathcal{I}u,i} & \implies L_y(y,\lambda^k)_i = -\lambda_i^k b_{\mathcal{I}u,i} \\ \text{if } \lambda_i^k = 0, & y_i \in [b_{\mathcal{I}l,i}, b_{\mathcal{I}u,i}] & \implies L_y(y,\lambda^k)_i = 0 \end{cases} \tag{5.9}$$

If K is block diagonal, as will be proposed in Section 5.4.1, (5.8) will instead separate into one (smaller) LP for each block. If the blocks are small, the LPs can be solved efficiently using explicit optimization software such as Multi-Parametric Toolbox (MPT) [Kvasnica et al., 2004]. This is however not implemented.

As for $L_x(x,\lambda)$, $L_y(y,\lambda)$ will also be unbounded from below if the bounds are not finite. If both L_x and L_y turn out to be bounded for the current λ^k iterate, we can simply evaluate the dual as $\theta(\lambda^k) = L(x,y,\lambda^k)$ for the x minimizing $L_x(x,\lambda^k)$ and y minimizing $L_y(y,\lambda^k)$.

5.3.2 Evaluating a Modified Dual

Whenever the dual function $\theta(\lambda)$ is unbounded for the current iterate $\lambda = \lambda^k$, it is of no use for us. We can instead modify the Lagrangian by adding a regularization term, very similar to (2.11):

$$L_1(x,y,\lambda) = g(x) + h(y) + \lambda^T(Cx - y) + \frac{1}{2}\left\|E_m(Cx-y)\right\|^2. \tag{5.10}$$

By the same argument, this term will not change the optimal value J^* for any matrix E_m since $Cx = y$ at any feasible point. If we would evaluate the corresponding dual function $\theta_1(\lambda) = \inf_{x,y} L_1(x,y,\lambda)$, it would in fact give a tighter bound than $\theta(\lambda)$ for any λ. Unfortunately, evaluation of θ_1 is much harder since it's not separable in x and y.

Let us ignore $h(y)$, and consider the even more modified dual

$$\theta_2(\lambda) = \inf_{x,y} L_2(x,y,\lambda), \tag{5.11}$$

where

$$L_2(x,y,\lambda) = g(x) + \lambda^T(Cx - y) + \frac{1}{2}\left\|E_m(Cx-y)\right\|^2. \tag{5.12}$$

Since $h(y) \geq 0 \ \forall y$, $\theta_2(\lambda)$ will also give valid bounds, i.e. $\theta_2(\lambda) \leq J^* \ \forall \lambda$. While neither L_1 nor L_2 is separable in x and y, at least L_2 contains no inequality cosntraints. Introducing dual variables for the equality constraints in the term $A_{\mathcal{E}}^T \mu$, and taking the derivative with respect to x, μ and y yields the KKT system

$$\begin{bmatrix} H + C^T E_m^T E_m C & A_{\mathcal{E}}^T & -C^T E_m^T E_m \\ A_{\mathcal{E}} & 0 & 0 \\ -E_m^T E_m C & 0 & E_m^T E_m \end{bmatrix} \begin{bmatrix} x \\ \mu \\ y \end{bmatrix} = \begin{bmatrix} -f - C^T \lambda \\ b_{\mathcal{E}} \\ \lambda \end{bmatrix}. \tag{5.13}$$

If x and y are part of a solution to (5.13), we also have a lower bound on J^* by evaluating $L_2(x, y, \lambda^k)$.

Since we have completely left out the inequality constraints in $h(y)$, evaluation of θ_2 may give a poor bound on J^*. The bound is however tightened by the regularization term, and E_m can be chosen as any matrix of suitable size. The larger[7] we pick E_m, the tighter the bound will be. In practice though, there is a limit as to how large it can be before encountering numerical issues when solving (5.13).

5.4 Preconditioning

The diagonal preconditioning matrix E defined in Section 5.1 may be used to scale the optimization problem for better conditioning[8]. While QPgen implements several methods for finding a good preconditioning matrix, none of them have given satisfactory results for our QP relaxations of the Spring-Damper example. As we have discovered that the "big M" formulation (see Section 2.3) might be causing problems, this Section illustrates how to overcome some of these by an alternative way of preconditioning.

Due to the regularization term in (5.2a) and (5.2b), the primal variables y and Cx are prohibited from taking very large steps away from each other. The extent of this effect can however be tuned by preconditioning with the scale matrix E. If E_{ii} is small, the elements y_i and $(Cx)_i$ will take larger steps. The choice of E will however also influence the dual variables λ, as can be seen in (5.2c). If E_{ii} is small, λ_i will instead take smaller steps. The effect of the step size parameter ρ can be seen as scaling the entire preconditioning matrix, and thus determines whether we will take larger steps in λ or x and y.

If we have some knowledge about the scale of $y = Cx$ and λ, we can use this to set the preconditioning to something like $E_{ii} = |\lambda_i / y_i|$. Of course we have to avoid zero or infinite elements on the diagonal of E, e.g. by the modification $E_{ii} = \max(|\lambda_i|, \varepsilon) / \max(|y_i|, \varepsilon)$.

[7] For example in the sense of large eigenvalues.

[8] Loosely speaking, a well-conditioned problem has the same curvature in all directions.

5.4.1 Preconditioning on x

In situations where variables have very different scaling, a proper preconditioning becomes even more important. Such a case can occur due to the "big M" formulation, described in Section 3.3.2. The auxiliary variables introduced will indeed behave properly as long as the mode switching variables are binary constrained. However, a relaxed switching variable can cause the corresponding optimal auxiliary variables to be very large, if the constants m and M are of great magnitude. This scenario is an example of when we have knowledge of the scaling of x. However, we can only scale elements of Cx, not x, by the preconditioning matrix E.

As mentioned in Section 2.4, the approach (2.22) makes the update step of y intractable. On the other hand, it gives us full flexibility in preconditioning based on $Cx = x$, as we desire for the big M case. However, if exploiting the structure of (A.2) and (3.10c), it may be possible to do something in between (2.22) and (2.23).

Let $P_{C_\mathcal{I}}$ be a permutation matrix, permuting the rows of $C_\mathcal{I}$. We then split the permuted matrix into blocks $C_\mathcal{I}^i$ such that

$$P_{C_\mathcal{I}} C_\mathcal{I} = \begin{bmatrix} C_\mathcal{I}^0 \\ C_\mathcal{I}^1 \\ \vdots \\ C_\mathcal{I}^m \end{bmatrix}, \tag{5.14}$$

and define matrices

$$K_P = P_{C_\mathcal{I}} K = \begin{bmatrix} I & & & \\ & K_1 & & \\ & & \ddots & \\ & & & K_m \end{bmatrix}, \quad C = \begin{bmatrix} C_\mathcal{I}^0 \\ C_1 \\ \vdots \\ C_m \end{bmatrix}, \tag{5.15}$$

with blocks fulfilling $C_\mathcal{I}^i = K_i C_i,\ i = 1, \dots, m \implies P_{C_\mathcal{I}} C_\mathcal{I} = K_P C$.

The block structure of K_P, splits the optimization problem (5.4) into one clip operation for the I block, and one non-trivial optimization problem for each block K_i. However, if the blocks are small we can apply explicit optimization methods for solving these subproblems fast. Due to the "block diagonal"[9] structure (A.2) of $C_\mathcal{I}$, the desirable approach of (2.22) would in fact split (5.4) into one subproblem per time instant. Despite this, the subproblems will still be large if the MLD system is large[10], and should preferably be split up even more.

[9] $C_\mathcal{I}$ is not block diagonal strictly speaking, but by a permutation of rows it is.

[10] In the sense of many states, inputs, auxiliary variables or constraints.

Consider the "big M" inequalities defined by (3.10c). If the red and green elements would (with a permutation of columns) separate into different blocks, we could split the subproblems further. As only the 3rd column (corresponding to u_b) holds both red and green elements, this would almost be possible. We will now go through how C_1, C_2 and K_1, K_2 can be chosen to split up the "big M" inequalities for the first time step, but the procedure will be the same for the remaining time steps.

Let e_i denote the vector such that its i:th coordinate is 1, while all the others are 0. Also let $I(x(k))$, $I(u_b(k))$, etc. denote the indices of the corresponding variables $x(k)$, $u_b(k)$, etc. in (A.1). Now let C_1 pick out the variables corresponding to the green elements, and C_2 the variables corresponding to the red elements:

$$C_1 = \begin{bmatrix} e_{I(u_b(1))}{}^T \\ e_{I(x(1))}{}^T \\ e_{I(z_x(1))}{}^T \end{bmatrix}, \quad C_2 = \begin{bmatrix} e_{I(u_b(1))}{}^T \\ e_{I(u_r(1))}{}^T \\ e_{I(z_u(1))}{}^T \end{bmatrix} \tag{5.16}$$

Then we can construct the blocks K_1 and K_2 to hold the "big M" formulation for z_x and z_u, respectively:

$$K_1 = \left[\begin{array}{ccc} M_x & 1 & -1 \\ -m_x & -1 & 1 \\ m_x & 0 & -1 \\ -M_x & 0 & 1 \\ \hline 0 & 0 & 1 \end{array}\right], \quad K_2 = \left[\begin{array}{ccc} M_u & 1 & -1 \\ -m_u & -1 & 1 \\ m_u & 0 & -1 \\ -M_u & 0 & 1 \\ \hline 0 & 0 & 1 \end{array}\right] \tag{5.17}$$

By the last row in K_1 and K_2, additional constraints on the auxiliary variable are incorporated, as these may be present in the lower (identity) part of $C_{\mathcal{I}}$ in (A.2).

By defining C_1 and C_2 as in (5.16), we do indeed map $x(1)$, $z_x(1)$, $u_r(1)$ and $z_u(1)$ to individual elements in y, according to $y = Cx$. Thus, we may for example choose a preconditioning which allows for large steps in the possibly large auxiliary variables $z_x(1)$ and $z_u(1)$. However, C_1 maps $u_b(1)$ to one element in y while C_2 maps it to another. Though (if desired) we can apply preconditioning to both of these elements in y, there is still an ambiguity effect as illustrated by Figure 2.3 in Section 2.4.

The additional blocks C_i and K_i for $i > 2$ are chosen accordingly, for the remaining time steps. If we then have rows left in $C_{\mathcal{I}}$ which have not been handled by these blocks, we simply let $C_{\mathcal{I}}^0$ hold these rows[11], and the corresponding inequalities will be subject to the trivial clip operation.

As the K_i blocks are small independently of the size of the MLD system, we can apply explicit optimization for solving the corresponding subproblems

[11] As we are free to select the permutation matrix $P_{C_{\mathcal{I}}}$, we put the remaining rows of $C_{\mathcal{I}}$ to the top of $P_{C_{\mathcal{I}}} C_{\mathcal{I}}$.

fast. Our implementation makes use of Multi-Parametric Toolbox (MPT) [Kvasnica et al., 2004], for this purpose.

If we would have more states x or inputs u_r, each of them would introduce an additional auxiliary variable and an additional "big M" block in (3.10c), giving additional blocks C_i and K_i, and some increased ambiguity for u_b.

5.5 Line Search

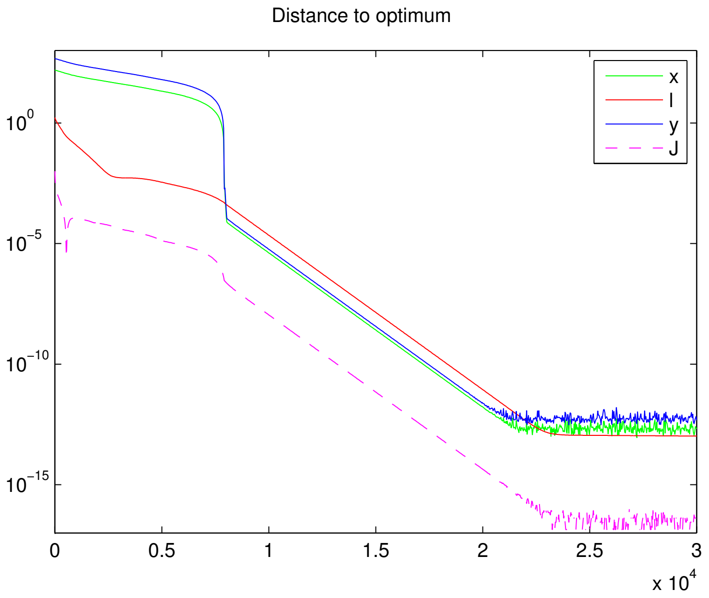
One of the big drawbacks with ADMM is the difficulty in finding a good preconditioning which results in good convergence. Even if a good preconditioning is found for some of the nodes in the Branch and Bound routine, there is no guarantee that it will work for all of them. The results of this can be devastating when it comes to the number of iterations (and thus time) it takes to converge. One illustration of the convergence for a problem with bad preconditioning is shown in Figure 5.1. In this section we develop an efficient method to mitigate some of these problems.

If we study the convergence in Figure 5.1 closer, we notice an interesting feature. For a long period of time when the step size is short, the iterations seem to be going in a straight line. Furthermore, not only do they go in the same direction most of the time, they even go almost straight towards the optimum. For the first 8000 iterations, when the convergence is as slowest for this problem, we do not see this behavior too clearly. In many other examples however, it was very apparent that the solver was taking very short steps in a seemingly straight line.

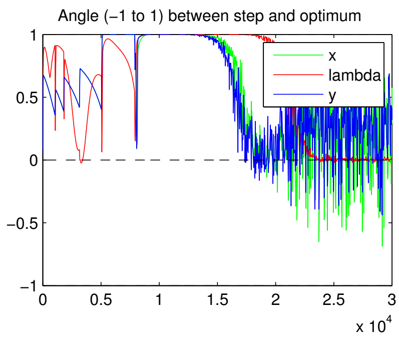
Since this phenomenon was observed for several problems, it spurred the idea of taking large steps in this direction when that kind of stagnation occurred. The obvious choice for taking this step is using a line search. This however requires that it can be implemented efficiently.

Our first attempt was to fix λ and minimize the regularized Lagrangian (2.11) over x and y in the detected direction. This method resulted in some improvement, but it wasn't satisfactory. We instead developed a method that performs a line search in all the variables x, y, λ by minimizing the distance to a saddle point. This can also be interpreted as minimizing the stopping criterion (2.21).

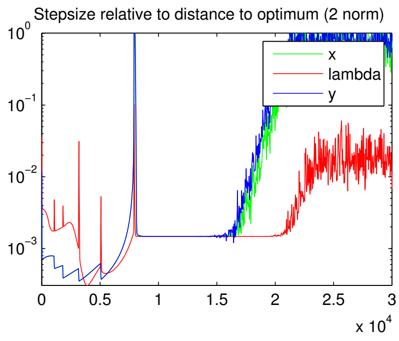
Although we can no longer guarantee convergence when applying the line search, our experiments indicate that this is not a problem. Even the line search is not always effective, it never seems to inhibit convergence. It may be possible to prove convergence for this method, but that is outside the scope of this thesis.



(a) Distances to the optimum at each iteration for the variables x, λ, y and the total cost J.



(b) Plot of the direction of each iteration compared to the optimum. 1 means a step straight towards optimum and -1 a step in the opposite direction.



(c) Plot of each stepsize as compared to the total distance to the optimum. It is noteworthy that the algorithm takes steps between iteration 10000 and 15000 that are of a constant length relative to the remaining distance.

Figure 5.1 Plots illustrating the convergence properties of ADMM for a problem with slow convergence.

5.5.1 Algorithm

For detecting that the algorithm is moving in a constant direction we introduce the following criterion

$$\left\| \frac{v^k - v^{k-N}}{\|v^k - v^{k-N}\|} - \frac{v^{k-N} - v^{k-2N}}{\|v^{k-N} - v^{k-2N}\|} \right\|^2 \leq \epsilon \cdot N_e \tag{5.18}$$

where N_e is the number of elements in the vector. We apply this criteria for all variables $v = x$, $v = y$ and $v = \lambda$. This can be interpreted as requiring the angle between the last N steps and the N steps before that to be small, for both x, y and λ. In our implementation we used $\epsilon = 10^{-7}$ for x and y, and $\epsilon = 10^{-5}$ for λ.

For the horizon N we chose values between 10 and a few hundred. This value can be tweaked to change how the line search performs. For efficiency reasons we only do this check every N iterations. This means that we only have to save the state v^k for two previous iterations. It is thus important to note that N will not only change how far back in the history we look, but also how often we might trigger the line search. It might be interesting to tweak these properties independently, but it is not something that we studied in this thesis.

Let the detected direction $(x^k - x^{k-N})/\left\|x^k - x^{k-N}\right\|$ be denoted as $\tilde{x}$, the current iteration as x^0, and for the purpose of the line search let $x = x^0 + t\tilde{x}$ and define the variables $y^0, \lambda^0, \tilde{y}, \tilde{\lambda}$ analogously. Given the Lagrangian (2.9)

$$L(x, y, \lambda) = g(x) + h(y) + \lambda^T (Cx - y),$$

where

$$\begin{aligned} g(x) &= \frac{1}{2} x^T H x + f^T x + c + \mathbb{I}(A_{\mathcal{E}} x = b_{\mathcal{E}}) \\ h(x) &= \mathbb{I}(y \in \mathcal{Y}) \end{aligned},$$

the optimality conditions are, as introduced in Section 2.3.4:

$$\begin{aligned} 0 &= \nabla_\lambda L(x, y, \lambda) = Cx - y \\ 0 &\in \partial_x L(x, y, \lambda) = \partial g(x) + C^T \lambda \\ 0 &\in \partial_y L(x, y, \lambda) = \partial h(y) - \lambda. \end{aligned} \tag{5.19}$$

We are here using addition of sets, defined as $A + B = \{a + b \,|\, a \in A, b \in B\}$. It can also be noted here that we require the sub-differential to be non-empty which means that both indicator functions must be finite. This follow automatically for the equality constraint from feasibility of x^k and x^{k-N}, and will be handled explicitly for the inequality below. We now choose to minimize the distance to an optimal point through the norm

$$\mathcal{J}(x,y,\lambda) = \underbrace{\|Cx-y\|^2}_{\mathcal{J}_1(x,y)} + \underbrace{\left\|\partial g(x) + C^T\lambda\right\|^2_{\min}}_{\mathcal{J}_2(x,\lambda)} + \underbrace{\|\partial h(y) - \lambda\|^2_{\min}}_{\mathcal{J}_3(y,\lambda)}, \tag{5.20}$$

where

$$\|\mathcal{S}\|_{\min} \doteq \min_{s\in S} \|s\|\,.$$

For the term $\mathcal{J}_2$ we look at $\partial g(x)$ separately in the direction $x = x_{\mathrm{K}} + x_{\mathrm{I}}$, where $x_{\mathrm{K}} \in \operatorname{Ker} A_{\mathcal{E}}$ and $x_{\mathrm{I}} \in \operatorname{Im} A_{\mathcal{E}}^T \perp \operatorname{Ker} A_{\mathcal{E}}$. For any point $A_{\mathcal{E}}x = b_{\mathcal{E}}$, the sub derivatives can be calculated to

$$\begin{aligned} \frac{\partial \mathbb{I}_{\mathcal{E}}}{\partial x_{\mathrm{K}}}(x) &= 0, && \forall x_{\mathrm{K}} \in \operatorname{Ker} A_{\mathcal{E}} \\ \frac{\partial \mathbb{I}_{\mathcal{E}}}{\partial x_{\mathrm{I}}}(x) &= [-\infty,\infty], && \forall x_{\mathrm{I}} \perp \operatorname{Ker} A_{\mathcal{E}}. \end{aligned} \tag{5.21}$$

The sub gradient at x is therefore any vector perpendicular to the null space of $A_{\mathcal{E}}$

$$\partial \mathbb{I}_{\mathcal{E}}(x) = \{v : v \perp \operatorname{Ker} A_{\mathcal{E}}\} = \operatorname{Im} A_{\mathcal{E}}^T.$$

It is thus possible to simplify the second term $\mathcal{J}_2$ to

$$\begin{aligned} \left\|\partial g(x^0) + C^T\lambda\right\|^2_{\min} &= \left\| \underbrace{Hx + f - C^T\lambda}_{u} + \partial\mathbb{I}_{\mathcal{E}}(x)\right\|^2_{\min} = \\ &= \left\|P_{\mathrm{K}}u + P_{\mathrm{I}}u + \operatorname{Im} A_{\mathcal{E}}^T\right\|^2_{\min} = \\ &= \|P_{\mathrm{K}}u\|^2_{\min} + \left\|P_{\mathrm{I}}u + \operatorname{Im} A_{\mathcal{E}}^T\right\|^2_{\min} = \\ &= \left\|P_{\mathrm{K}}(Hx + f + C^T\lambda)\right\|^2 + 0, \end{aligned} \tag{5.22}$$

where P_{K} and P_{I} are orthogonal projections onto the orthogonal subspaces $\operatorname{Ker} A_{\mathcal{E}}$ and $\operatorname{Im} A_{\mathcal{E}}^T$ respectively. That the norm-min can be split follows from the fact that all the vectors in the first part are orthogonal to all the vectors in the second.

The evaluation of $\mathcal{J}_3 = \|\partial h(y) + \lambda\|^2_{\min}$ requires more care. We have

$$h(y) \quad = \quad \mathbb{I}(b_{\mathcal{I}l} \le Ky \le b_{\mathcal{I}u}) = \sum_{i=1..n} \mathbb{I}(b_{\mathcal{I}l,i} \le k_i y \le b_{\mathcal{I}u,i}),$$

where k_i is the i:th row in K. The sub-differential can then be written as

$$\partial h(y) = \sum_{i=1..n} \partial\mathbb{I}(b_{\mathcal{I}l,i} \le k_i y \le b_{\mathcal{I}u,i}) = \sum_{i=1..n} c_i(y) k_i^T$$

where

$$c_i(y) = \begin{cases} 0 & \text{if } b_{\mathcal{I}l,i} < k_i y < b_{\mathcal{I}u,i} \\ [-\infty, 0] & \text{if } b_{\mathcal{I}l,i} = k_i y < b_{\mathcal{I}u,i} \\ [0, \infty] & \text{if } b_{\mathcal{I}l,i} < k_i y = b_{\mathcal{I}u,i} \\ [-\infty, \infty] & \text{if } b_{\mathcal{I}l,i} = k_i y = b_{\mathcal{I}u,i} \end{cases}.$$

It is in general not trivial to calculate the norm $\left\|\sum c_i(y)k_i^T - \lambda\right\|_{\min}$. It requires a projection of λ onto $\sum c_i(y)k_i^T$, which means solving an inequality constrained QP. This makes the line search expensive in the general case. However, with $K = I$, the sum can be simplified to

$$\begin{aligned} \partial h(y) &= \sum_{i=1..n} \partial \mathbb{I}(b_{\mathcal{I}l,i} \le e_i^T y \le b_{\mathcal{I}u,i}) = \\ &= \sum_{i=1..n} c_i(y)e_i = \begin{bmatrix} c_1(y) & \dots & c_n(y) \end{bmatrix}^T \doteq c(y). \end{aligned} \tag{5.23}$$

The cost $\mathcal{J}_3$ can in this case be calculated coordinate wise as

$$\begin{aligned} \left\|\sum c(y) - \lambda\right\|_{\min}^2 &= \sum_{i=1..n} \|c_i(y) - \lambda_i\|_{\min}^2 = \sum_{i=1..n} \left(\text{clip}_{c_i(y)}(\lambda_i) - \lambda_i\right)^2 = \\ &= \sum_{i=1..n} \left(\text{clip}_{\tilde{c}_i(y_i)}(\lambda_i)\right)^2 \end{aligned}$$

where

$$\tilde{c_i}(y) = \begin{cases} [-\infty, \infty] & \text{if } b_{\mathcal{I}l,i} < y < b_{\mathcal{I}u,i} \\ [0, \infty] & \text{if } b_{\mathcal{I}l,i} = y < b_{\mathcal{I}u,i} \\ [-\infty, 0] & \text{if } b_{\mathcal{I}l,i} < y = b_{\mathcal{I}u,i} \\ 0 & \text{if } b_{\mathcal{I}l,i} = y = b_{\mathcal{I}u,i} \end{cases}.$$

The total cost function for the line search can now be written

$$\mathcal{J}(x, y, \lambda) = \|Cx - y\|^2 + \left\|P_{\text{K}}(Hx + f + C^T\lambda)\right\|^2 + \sum_{i=1..n} \left(\text{clip}_{\tilde{c}_i(y)}(\lambda_i)\right)^2. \tag{5.24}$$

We now have a way to evaluate the cost $\mathcal{J}$ which is a requirement to do the minimization. We want to minimize this function, from the current position (x^0, y^0, λ^0), in the detected direction $(\tilde{x}, \tilde{y}, \tilde{\lambda})$, while keeping the solution feasible ($y \in \mathcal{Y}$). We refer the reader to Appendix B for the details on how this can be implemented efficiently.

5.6 Line search with preconditioning on x

We have now introduced two methods, the line search and the preconditioning on x, both with the hope of improving bad convergence properties in ADMM on some of our problems. As is shown in the results in Section 8.2 and 8.3 this is also the case for many problems. It is thus tempting to consider how these methods can work together. It is quickly apparent that it is not trivial to make this happen.

The main reason that we are able to add preconditioning on x instead of y is that we are no longer using $K = I$ for the inequality $b_{\mathcal{I}l} \leq Ky \leq b_{\mathcal{I}u}$. On the other hand, the only reason that we are able to both compute and minimize the cost $\mathcal{J}_3(y, \lambda)$ so efficiently is that $K = I$. It may be possible to come up with a relatively cheap way to evaluate the cost $\mathcal{J}_3$, using for example the Multi-Parametric Toolbox (which we already use at each iteration when doing preconditioning on x). It seems however, that there would be no explicit way to do the line search, thus forcing us to do many of these relatively expensive evaluations. We therefore didn't implement a complete combination of the two ideas.

On the other hand, neither of the costs $\mathcal{J}_1$ and $\mathcal{J}_2$ are affected by the use of $K \neq I$. We have therefore implemented a combination of the two methods where the line search is only minimizing the cost of the first two functions $\mathcal{J}_1$, $\mathcal{J}_2$. There is still no guarantee that this method will converge. However, we have seen in the testing that the two methods work well together in many cases.

6

Improving Branch and Bound

6.1 Tree Traversal Strategies

There are different ways of traversing the Branch and Bound tree, and they may differ a lot in the number of nodes having to be solved. Reminding the priority list implementation in Section (2.2.1), different traversal strategies can be implemented by different choices of priority for the tree nodes.

Let the priority $p_i = p(N_i)$ be the priority assigned to node N_i. If $p_i < p_j$, N_i is prioritized over N_j. A basic example of priority selection is $p_i = 1/(d_i + 1)$, where d_i is the depth of N_i (the number of fixed binary variables). This corresponds to the traversal strategy *depth-first*, ultimately prioritizing leaf nodes. Another basic example is $p_i = d_i$, which corresponds to *breadth-first.* Our experience is that both *depth-first* and *breadth-first* search give poor results (too many nodes need to be solved).

Another alternative is *best-first* search, which assigns priorities $p_i = J^*_{N_j}$, where the lower bound $J^*_{N_j}$ was computed when solving the relaxed problem at the parent node N_j of N_i.

Let the function $s_{fix}(x^*_{N_j}, N_i)$ output a slight modification to the solution $x^*_{N_j}$ of the relaxed problem at N_j, with an additional binary variable fixed, according to branching from N_j to its child node N_i. Then we assign priorities according to (6.1) below:

$$p_i = J(s_{fix}(x^*_{N_j}, N_i)). \tag{6.1}$$

The traversal strategy (6.1) is simply *best-first* search, but with the objective function being evaluated when fixing the binary variables[1] according to N_i. If the binary variable to be branched on has a cost assigned to it, the child with lowest cost will be prioritized over its sibling. In Section 6.1.1, (6.1) will be further modified.

[1] Note that $s_{fix}(x^*_{N_j}, N_i)$ may be infeasible.

6.1.1 Suboptimal Branch and Bound

As proposed in [Axehill et al., 2014], the number of nodes to be solved may be reduced if we do not require absolute optimality. In Section (2.2.1), we allowed pruning of a subtree if $J^*_{N_i} \geq J_{ub}$. If we also allow pruning in the case

$$(1 + \varepsilon_s) \cdot J^*_{N_i} \geq J_{ub}, \tag{6.2}$$

where $\varepsilon_s \geq 0$ is a suboptimality tolerance, we may of course prune more often, and thus reduce the number of QP relaxations to be solved. Due to (2.7) we also ensure that $(1+\varepsilon_s) \cdot J^*_{N_i} \geq J^*$, meaning that ε_s (which we may pick arbitrarily) really is a bound on the suboptimality of our final "solution" $\hat{x}^*$ of (2.6).

While the suboptimal approach is appealing, it relies on having found a good enough point x_{ub} with objective function value $J_{ub} \leq (1 + \varepsilon_s) \cdot J^*_{N_i}$ to be able to prune the subtree of N_i. However, if we stick with *best-first* search as defined by (6.1), we have observed that x_{ub} and J_{ub} will typically not be updated for a long time – possibly not until we have found an optimal leaf.

To understand this, first consider Theorem 1 concerning ideal[2] *best-first* search, as defined by (6.3).

$$p_i = J^*_{N_i} \tag{6.3}$$

Theorem 1

If Branch and Bound is applied to a feasible MIQP (2.6) with ideal *best-first* search (6.3), x_{ub} and J_{ub} will not be updated until the solution to (2.6) is found. This is the case both with or without suboptimal Branch and Bound, i.e. whether pruning requires the condition (2.7) or (6.2). □

Proof Let A denote the set of all nodes N in the priority list, such that N is an optimizing leaf, or an ancestor to an optimizing leaf. Also let F denote the set of all nodes in the priority list, whose relaxations have binary feasible solutions. Since (2.6) is feasible, $F \neq \emptyset$.

Assume that x_{ub} and J_{ub} have not yet been updated, i.e. $J_{ub} = +\infty$ and x_{ub} is undefined. Then (suboptimal) pruning has never taken place, so $A \neq \emptyset$. According to (6.3), $p_{N_A} = J^*_{N_A} \leq J^*$ (for some arbitrary node $N_A \in A$), with equality only if also $N_A \in F$, which implies $p_{N_B} < J^*$ for any node $N_B \in B = A \cap F^*$.

Now consider any node $N_F \in F$, with solution $x^*_{N_F}$ and objective function value $J^*_{N_F}$. Because $x^*_{N_F}$ is feasible in (2.6), of course $p_F = J^*_{N_F} \geq J^*$. Then nodes $N_B \in B$ will be prioritized over any $N_F \in F$, since apparently $p_B < J^* \leq p_F$.

[2] In practice, (6.3) cannot be used since it requires solving the relaxation at N_i prior to assigning a priority to it, while we actually decide which nodes to solve based on their priority.

As mentioned in Section (2.2.1), x_{ub} and J_{ub} may only be updated with binary feasible solutions, i.e. solutions to relaxations of $N_F \in F$. Therefore nodes $N_F \in F$ will not be prioritized, as long as $B \neq \emptyset$. If indeed $B = \emptyset$ and a node $N_F \in F$ has the highest priority, then $N_F \in A \cap F$ (since $A \neq \emptyset, B = \emptyset \implies A \cap F \neq \emptyset$). Then x_{ub} will be updated to $x^*_{N_F}$, which will be a solution to (2.6). □

When assigning priorities according to the non-ideal *best-first* search (6.1), we cannot prove conclusions like Theorem 1. However, it is not surprising if the non-ideal priority assignment would in many cases give priorities similar to (6.3), and would hence not promote investigating leaves or other binary feasible nodes.

As mentioned previously, the suboptimal approach relies on having updated (x_{ub}, J_{ub}) to something good enough for suboptimal pruning. Thus a priority assignment not promoting binary feasible solutions is very problematic. Inspired by *depth-first* (which promotes leaves to an extreme level), we could modify (6.1) into something like (6.4).

$$p_i = \frac{J(s_{fix}(x^*_{N_j}, N_i))}{d_i + 1} \tag{6.4}$$

When trying this, we did however observe the *depth-first* drawback of traversing a large number of nodes. Reasoning that we do only want to explore the leaves occasionally, we settled on the priority assignment (6.5), also incorporates randomness. This helps avoid the algorithm getting stuck at some part deep down in the tree.

$$\begin{gathered} p_i = \frac{J(s_{fix}(x^*_{N_j}, N_i))}{e^Z}, \quad Z \sim \mathcal{N}\left(\mu_Z, \sigma_Z^2\right) \\ \mu_Z = \varepsilon_s \cdot \log(d_i + 1), \quad \sigma_Z = 0.5 \cdot \varepsilon_s. \end{gathered} \tag{6.5}$$

Still, d_i denotes the number of fixed binary variables at N_i, and ε_s is the suboptimality tolerance. Note that if Z is exchanged for its mean $\varepsilon_s \cdot \log(d_i + 1)$, (6.4) and (6.5) only differs by a factor e^{ε_s}. Also, (6.5) will coincide with (6.1) if no suboptimality is allowed, i.e. if $\varepsilon_s = 0$.

Apart from the priority assignment (6.5), warmstarting of the Branch and Bound tree could also help finding a (hopefully good) binary feasible node at an early stage (see Section 7.2.2). Note that both the priority assignment and warmstarting are also crucial for the Early Termination feature (see Section 5.3).

6.2 Branch Ordering

When dividing $\mathcal{S}$ into subsets according to Section 2.2.1, there is a freedom to choose in which order to branch on the variables. For example - in Figure

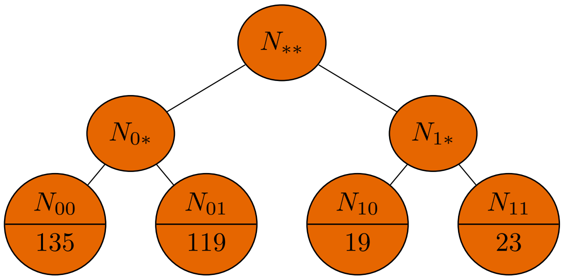


Figure 6.1 Illustrative example, where x_b^0 is branched on before x_b^1. Optimal costs at the leaves are also shown.

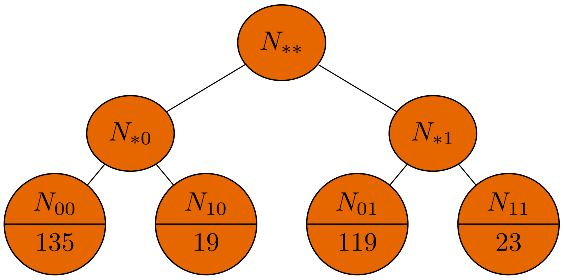


Figure 6.2 Illustrative example, where x_b^1 is branched on before x_b^0. Optimal costs at the leaves are also shown.

6.1 we branch first on x_b^0 and last on x_b^1, while in Figure 6.2 we branch first on x_b^1 and last on x_b^0.

Figures 6.1 and 6.2 illustrate different ways to branch for the same MIQP. The leaves are the same, with identical costs, but ordered differently. Apparently there is a tendency for high cost when the first binary variable $x_b^0 = 0$, and low cost when $x_b^0 = 1$. The second binary variable x_b^1 does not seem to affect the cost as much.

In Figure 6.1 we would expect the relaxation at node N_{0*} to have high optimal cost as that is the case for both its children. If we have traversed some of the low cost leaves N_{10} and N_{11} and the cost at N_{0*} is high enough, we will prune the subtree of N_{0*}.

Now consider Figure 6.2. As both N_{11} and thus N_{*1} has low optimal cost, it is unlikely that we can prune the subtree of N_{*1} even if we have traversed the optimal leaf N_{10}. Thus, we will have to branch N_{*1} into its children, and more nodes will have to be traversed.

We have concluded that the order in which we branch on binary variables can affect the number of nodes to be traversed. Also – if some binary variable x_b^i has greater influence[3] on the cost than some other variable x_b^j, x_b^i should

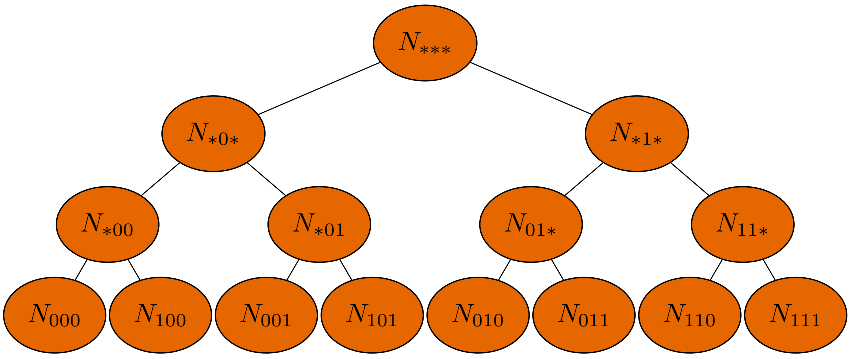


Figure 6.3 Illustration of dynamic branch ordering.

be branched on before x_b^j.

For causal systems (which are the only ones we consider), a binary variable at time k will not influence the cost at times $< k$, but may affect the cost at times $\geq k$. Thus, binary variables at early times may affect more of the terms in (3.11), and we argue that binary variables at early times should generally be branched on before variables at the end of the horizon. Therefore, we have always used this branch ordering in our experiments.

6.2.1 Dynamic Branch Ordering

Until now, we have only considered what we will call static branch ordering. With a static branch ordering, the variables are ordered in the same way in every part of the tree (if x_b^i is branched on before x_b^j in some part of the tree – the same will hold for the entire tree).

However, Branch and Bound does not require a static branch ordering, and how to branch from a node N_i may even be decided at runtime. Figure 6.3 shows an example of dynamic branch ordering.

Though it would be interesting to investigate how to pick a good branch ordering at runtime, we have not considered this due to limitations described in Section 7.1.1.

[3] Loosely speaking. x_b^i could have very different influence on the cost at different parts of the tree, and in such cases we have nothing to say about how to pick a good branch ordering.

7

Exploiting Repeated Optimization

7.1 Parametric Optimization

For an inequality constrained QP (2.1), a great advantage of first-order methods like ADMM is that the iterations are inexpensive to carry out. In addition, a great deal of the work may be done once for all iterations[1], making every iteration even cheaper.

Consider the KKT system of equations (5.3), and denote the left-hand-side matrix M. M depends on H, $A_{\mathcal{E}}$, C, E and ρ, but assume that all of these are fixed. Then one possibility is to compute M^{-1} beforehand, so that (5.3) can be solved by matrix multiplication. M^{-1} may however be dense even if M is sparse, causing a dense matrix multiplication which does not exploit the sparsity of M. In this case a sparsity preserving decomposition of M, such as the permuted LDL decomposition $P^TMP = LDL^T$, may be a better choice although still systems of equations (but now triangular) have to be solved.

The same inversion or decomposition of M may be used not only for every iteration, but also if solving several similar QPs. Although H, $A_{\mathcal{E}}$, C, E and ρ need to be fixed, the vectors f, $b_{\mathcal{E}}$, $b_{\mathcal{I}l}$ and $b_{\mathcal{I}u}$ may very well be altered from one problem instantiation to another. The resulting solver will solve a parametric QP, where the latter vectors are possible parameters.

7.1.1 Parametric Formulation of QP Relaxations

When applying MIPC to an MLD system we derive an MIQP on the form (2.6) according to Appendix A, where only $b_{\mathcal{E}}$ will vary from one time to another. However, we also want to express the QP relaxations parametrically.

[1] Second-order methods like the interior point method, need to solve a linear system of equations $Ax = b$ at each iteration, where A depends on x. This dependence makes it impossible to invert A or applying some decomposition to it once for all iterations.

At each node some binary variables are fixed while others are binary constrained (according to the corresponding set $\mathcal{S}_{XX}$). The binary constrained variables are then relaxed according to (2.8). Both fixation and relaxation may be modeled by varying the upper and lower bounds $b_{\mathcal{I}l}$ and $b_{\mathcal{I}u}$:

$$\begin{cases} x_b^i = 0 & \implies 0 \leq x_b^i \leq 0 \\ x_b^i = 1 & \implies 1 \leq x_b^i \leq 1 \\ x_b^i \in \{0,1\} & \implies 0 \leq x_b^i \leq 1 \end{cases} \tag{7.1}$$

As the upper and lower bounds can be parameters, (7.1) allows us to apply parametric ADMM to all QP relaxations. Nevertheless, we have experienced convergence issues when sticking with (7.1). Typically the nodes with fixed binary variables required many more iterations than the root node. Therefore we have added equality constraints for the fixed binary variables, in addition to (7.1). Although we observe much more stability in the number of iterations required from one node to another, it comes with a drawback – $A_{\mathcal{E}}$ is now node dependent.

Since $b_{\mathcal{E}}$ can be parametric, the first two cases in (7.1) can be handled by the very same row, appended to $A_{\mathcal{E}}$. The third case however, requires that the row is not appended.

If the static branch ordering (see Section 6.2) is used, $A_{\mathcal{E}}$ will be constant w.r.t. nodes at the same level in the tree. Thus, M may be inverted/factorized only once for each level. Of course the offline computational effort will increase, but only linearly in the number of binary variables.

We have also implemented a possibility for sticking with the inequality constraints in (7.1), and thus have the same $A_{\mathcal{E}}$ for all nodes.

7.2 Warmstarting

7.2.1 Warmstarting Node Relaxations

If we have a good guess $\hat{x}$ of the solution x^* of a QP relaxation, the number of iterations can be reduced by initiating the solver at $x_0 = \hat{x}$ rather than at some arbitrary point like $x_0 = 0$. There are of course different ways of choosing a point $\hat{x}$.

When speaking of points or solutions "x" in the following, we could mean the optimal primal variables, dual variables or both, depending on how the actual algorithm is initiated. For example, ADMM is initiated by primal variables y_0 and dual variables λ_0, and these may here be referenced together as x_0.

For ADMM, one way of warmstarting is to choose $\hat{x}$ as the solution x^*_{par} of the QP relaxation at the parent node. This may be done without considering feasibility, since ADMM can be started at an infeasible point. As we do only add one more constraint on one binary variable, we expect x^* to be close to x^*_{par} (at least at some times), and we have thus picked our initial guesses as $\hat{x} = x^*_{par}$.

Another way of warmstarting a QP relaxation is possible if that particular node has already been solved at another, but recent time. That solution x^*_{old} may then (shifted) be chosen as an initial guess $\hat{x}$ of the current solution, similarly to Section 7.2.2. Although this approach could be promising, we have not explored it due to lack of time.

7.2.2 Warmstarting the Branch and Bound Tree

We can use information from previous times not only for warmstarting individual nodes, but also for choosing which nodes to prioritize.

We have the structure (A.1), where the optimizer x is obtained by stacking time by time. Similarly the binary variables x_b (picked out from x) can be obtained by stacking the binary variables time after time:

$$x_b = [x_b^0, u_b^0, w_b^0, x_b^1, u_b^1, w_b^1, \ldots, x_b^N, u_b^N, w_b^N]^T \tag{7.2}$$

If (7.2) represents the optimal binary variables (the optimal leaf) at the previous sample, we argue that for the current sample some of the leaves represented by

$$x_b^{shifted} = [x_b^1, u_b^1, w_b^1, \ldots, x_b^N, u_b^N, w_b^N, *, *, *]^T, \tag{7.3}$$

will probably have almost-optimal objective function values[2]. The last variables denoted by $*$ are free/unknown.

We force the Branch and Bound algorithm to explore the leaves represented by (7.3), by assigning very high priorities to the leaves themselves as well as to any ancestor to any of the leaves.

Note that this feature serves the same goal as the priority assignment (6.5) in Section 6.1.1 – to find a (hopefully good) binary feasible node at an early stage. Thus, warmstarting of the Branch and Bound tree may improve the features of Suboptimal Branch and Bound (Section 6.1.1) as well as Early Termination (Section 5.3).

7.3 Precomputation of Cutting Planes

The purpose of the method that we present here is to find parts of the Branch and Bound tree that will always be infeasible and exclude them from the search. The goal is also to use this information to improve the relaxations

[2] Assuming that the horizon is long enough for the last costs to be negligible or if the estimated cost Q_N accurately represents the real cost, this result should follow from Bellman's Principle of Optimality [Bellman, 1957].

used in the rest of the tree. It is an unfortunate limitation in ADMM that there is no way to do reliable and fast infeasibility detection. It is therefore even more interesting in our case to detect and avoid infeasible nodes, before the actual real time solver is run.
The simplest way to find infeasible nodes is to remove the equality constraint for the initial point. It is then possible to traverse the nodes in the tree to find any nodes that will necessarily be infeasible, no matter what the initial point is. It is not hard to make up cases where it is actually expected to have a large amount of infeasible nodes. For example, in the turbo car example, as described in Section 4.2, the system is limited to setting the binary variable to 1 only a few times. Several of the nodes (and sub trees) will therefore be infeasible for all initial points.

The most basic way to handle this knowledge is to avoid the nodes in the Branch and Bound routine. This can simply be done by automatically setting them to infeasible as soon as they are reached, without doing any computation.

7.3.1 Extra constraints in precomputation

Every inequality constraint that is added to the problem will increase the computational cost when using ADMM. It could therefore sometimes be beneficial to not include all the known constraints into the problem formulation. For example, if it is known that the system will remain within some invariant set, it may not be efficient to add this knowledge as a constraint since the system will automatically satisfy it anyway. However, if this constraint (or a conservative approximation of it) is added in the precomputation, it can be possible to identify nodes that the system will never be able to reach in practice, as infeasible.

7.3.2 Adding constraints to the problem

Although it can be useful to exclude specific nodes in the Branch and Bound routine, a lot more can be done when knowing that some nodes are infeasible. The idea is to not only exclude these nodes, but also to restrict other relaxations, whenever possible. Study for example the problem with a Branch and Bound tree as shown in Figure 7.1. In this tree, we assume that nodes $\mathcal{S}_{0,1,1}$ and $\mathcal{S}_{1,1,1}$ are infeasible. In the best case, if we are simply skipping these nodes, we are reducing the number of nodes that need to be solved by 4 (solving $\mathcal{S}_{01}, \mathcal{S}_{011}, \mathcal{S}_{11}$ or $\mathcal{S}_{111}$ is redundant). However, we would still be solving exactly the same problems in the rest of the tree. If we instead restrict the space of feasible solutions in all nodes by excluding the known infeasible solutions, we can expect better bounds.

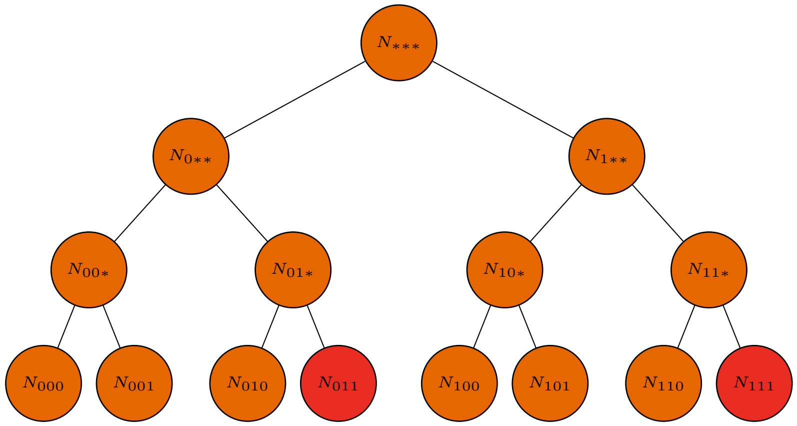


Figure 7.1 Illustration of tree with two infeasible nodes, as described in Section 7.3.2. Observe that these nodes could represent a subtree if the branching order was different.

The simplest restrictions given the infeasible nodes would be

$$\begin{aligned} \text{not } [1,1,1] &\longrightarrow & x_b^1 + x_b^2 + x_b^3 &\leq 2 \\ \text{not } [0,1,1] &\longrightarrow & (1 - x_b^1) + x_b^2 + x_b^3 &\leq 2. \end{aligned} \tag{7.4}$$

This restriction to the relaxed binary variables is shown in Figure 7.2. It is apparent that this is not the maximal restriction possible. For example, the solution $[0.5, 0.5, 1]$ is still possible in the root node, even though it is not in the convex hull of the feasible points.

By reformulating the exclusion of $[1, 1, 1]$ and $[0, 1, 1]$ to the exclusion of $[*, 1, 1]$, we can get the maximal restriction, which is depicted in Figure 7.3,

$$\text{not } [*, 1, 1] \longrightarrow x_b^2 + x_b^3 \leq 1. \tag{7.5}$$

We have not looked extensively into how this conversion can be done in general. We believe this to be a non-trivial problem and have only considered a few special cases.

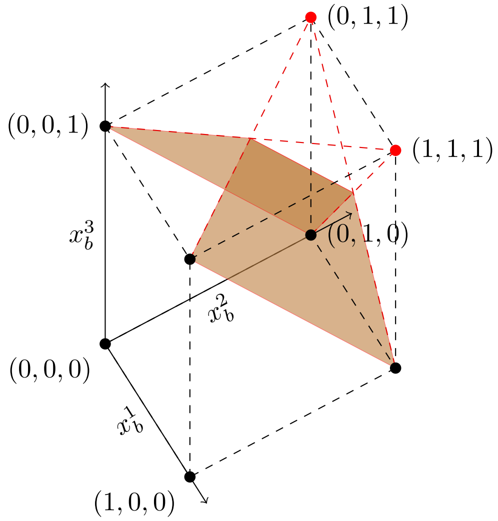


Figure 7.2 Illustration of restrictions that makes certain nodes unfeasible, but that does not restrict the relaxed space maximally. The planes correspond to the restrictions $x_b^1 + x_b^2 + x_b^3 \leq 2$ and $(1 - x_b^1) + x_b^2 + x_b^3 \leq 2$.

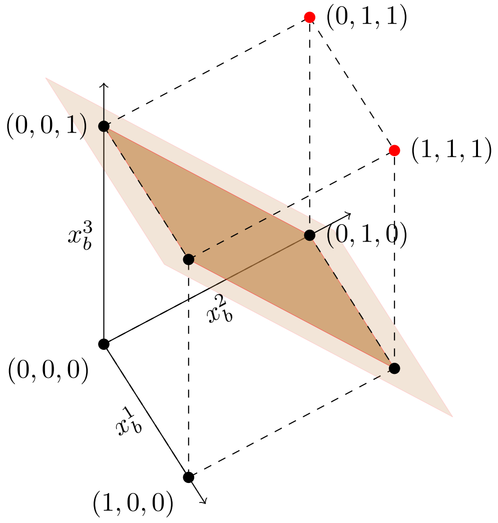


Figure 7.3 An illustration of the optimal restriction of relaxed binary variables. Only the convex hull of the binary feasible points is feasible in the relaxed problem. The plane corresponds to the restriction $x_b^2 + x_b^3 \leq 1$ which is equivalent to the restrictions in Figure 7.2 for binary x_b.

8

Results

8.1 Default Options

In this section we define some default options, which we will use for our results unless stated otherwise.

8.1.1 Branch and Bound Options

For the Branch and Bound solver, we do not allow for any suboptimality by default. Furthermore we do not compute any cutting planes in advance, and the tree is not warmstarted. If however ADMM is used as the QP solver, warmstarting from parent nodes will be enabled[1]. Even if ADMM is used, the early termination feature is not activated by default.

8.1.2 ADMM Options

By default we do not use any line search with ADMM, and we do not use the "preconditioning on x" feature. Neither do we use any preconditioning at all ($E = I$). We do however let ρ be computed automatically according to [Giselsson and Boyd, 2014a], but with an additional scaling of 120 for the Turbo Car example[2]. At every node the known binary variables are fixed with both inequality and equality constraints.

8.2 Line Search

In this section we will evaluate the line search algorithm on a few different problems. We apply the method on three different nodes in the Spring Damper example as well as the Turbo Car example. We try the line search with two different settings. When we set the interval between line search

[1] To be clear – if individual nodes are run, they will not be warmstarted from the corresponding parent.
[2] The additional scaling was simply chosen due to empirically improved convergence.

checks to $N = 100$, we call this "LS100", and when we set the interval to $N = 10$, we call it "LS10". A list of the results is shown in Table 8.2. These results were run with the default settings outlined above with the maximum number of iterations limited to 100 000. The step size ρ was set according to Table 8.1.

It is clear from the results that ADMM has a hard time solving the Spring Damper problem. The improvement in the root node is quite impressive. Not only is the problem solved very fast, the time cost is merely 1% when considering line search each 100th iteration and 10% at each 10th. The gain in terms of iterations on the other hand, is a reduction by more than 95%. The other nodes seemed even harder to solve, and it is hard to draw any direct conclusions more than that the line search does not increase the time per iteration significantly.

The Turbo Car was easier to solve and the results clearer. In every test it was far better to use any of the variations of the line search compared to the standard ADMM. There is no clear result on which of the variations of the line search is better. This is consistent with the other testing we have done. For some problems it seems better to line search often with relatively small steps in order to converge as fast as possible. On others it is more valuable to wait longer, which often results in larger steps in the line search.

The properties of the convergence for the root node of the Turbo Car example is illustrated in Figure 8.1. On the first row we show how the solver works without any line search. In the left column we show how far away the iterates x, y and λ are from the optimum at each iteration, as well as how far the cost J is from the optimal value. In the right column we show the direction of each iteration for x, y and λ compared to the location of the optimum. This angle is calculated as

$$\frac{2}{\pi} \arcsin\left(\frac{(v^k - v^{k-1}) \cdot (v^* - v^{k-1})}{\|v^k - v^{k-1}\| \, \|v^* - v^{k-1}\|}\right),$$

and will be 1 if the iteration is going straight towards the optimum, and -1 if it is going in the opposite direction. It is quite striking to see how quickly the solver starts to iterate very straight towards the optimum, without reaching it for a very long time.

When the line search is applied with $N = 100$ the solver starts very similarly. The solutions look identical up to iteration 1000. But just as all the angles seem to stagnate at 1, the line search kicks in and removes a large part of the remaining distance to the optimum. Around iteration 1300 the same thing happens again, this time bringing the solution almost all the way to a point where the stopping criterion is fulfilled, roughly 20 times faster than without line search.

This behavior is very typical for what we have observed in several examples. We have also observed that the convergence rate often is greatly

	Spring-Damper	Turbo Car
Root	$1.8 \cdot 10^{-4}$	0.39
N_{110}	$7.5 \cdot 10^{-5}$	0.39
N_{1010}	$5.0 \cdot 10^{-5}$	0.39

Table 8.1 Step size ρ for the different nodes and problems, when evaluating line search.

improved, directly after a line search. Our theory is that while ADMM might get stuck minimizing the function in an ill-conditioned direction, the line search quickly minimizes the function completely in this direction, allowing ADMM to start working in a new direction.

If we look at the last row where we allow the line search each 10th iteration, we see that the performance is not as good. It is hard to say exactly why this is the case. One idea is that ADMM is not given enough time to find a good direction, another is that the two methods are working slightly against each other from time to time. It should however be noted that a solution is found roughly twice as fast as compared to not using line search.

8.3 Preconditioning on x

In this section it is investigated to what extent the performance of ADMM may be increased, when preconditioning according to Section 5.4.1. The method is evaluated for both the Spring-Damper and the Turbo Car example, with horizon length $N = 10$ and step sizes according to Table 8.3.

All "big M" blocks[3] have been treated separately according to Section 5.4.1, and the diagonal preconditioning matrix E is set to:

$$E_{ii} = \begin{cases} 1 & \text{if } i \in \mathcal{I}_{aux} \\ 1/L_i & \text{if } i \notin \mathcal{I}_{aux} \end{cases}, \tag{8.1}$$

where $\mathcal{I}_{aux}$ is the set of all x indices such that x_i is a "big M" auxiliary variable, but whose corresponding binary variable is not fixed. Thus $\mathcal{I}_{aux}$, and therefore the preconditioning (8.1) differs between nodes. The values of L_i should correspond to the magnitude of the "big M" auxiliary variables,

[3] Although the Turbo Car does not have exactly the same "big M" formulation as (3.10c)-(3.10d), the inequalities (C.1b) are quite similar.

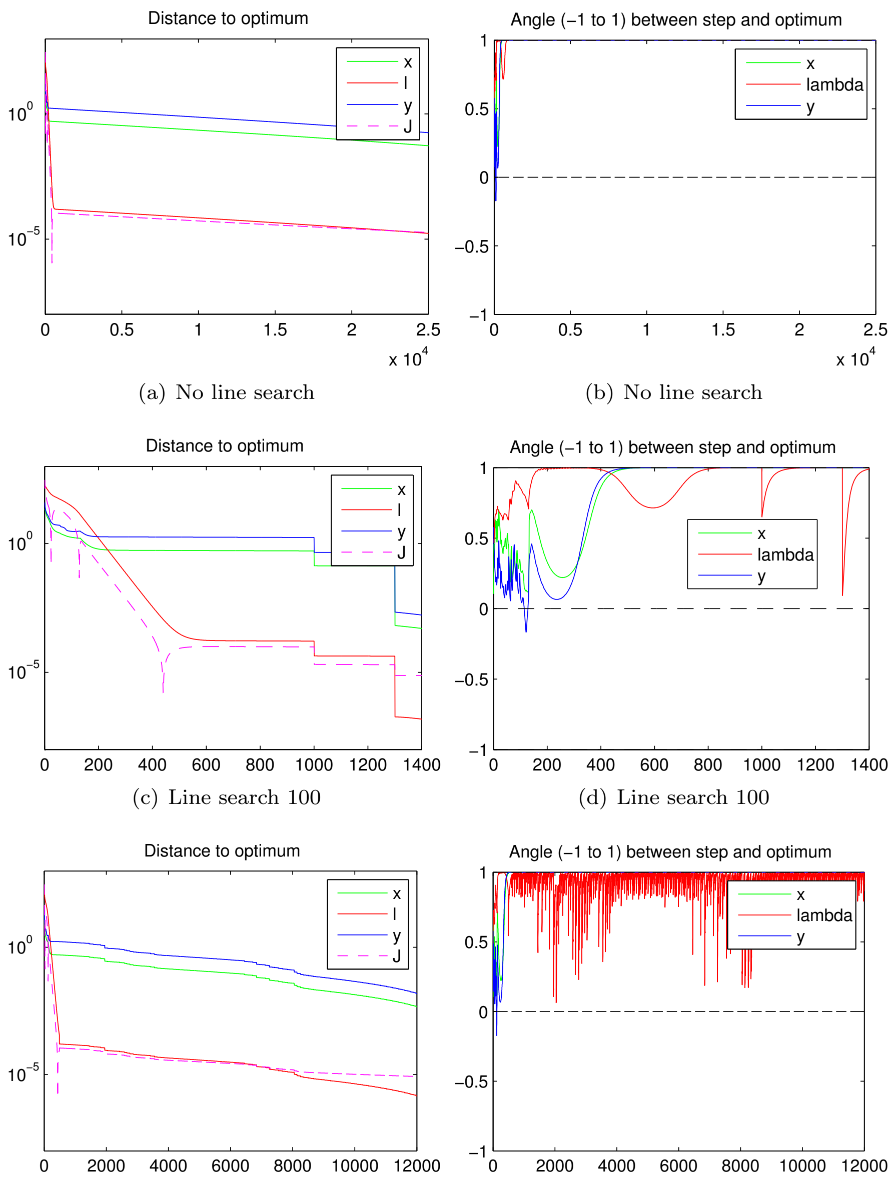
LS	**Node**	**Rel. error**	**Itrs.**	**Time / itr.** (ms)
Off	N	$2.20 \cdot 10^{-4}$	100 000	0.182
LS100		$4.51 \cdot 10^{-12}$	4 370	0.184
LS10		$4.90 \cdot 10^{-11}$	5 700	0.235
Off	N_{110}	$2.11 \cdot 10^{-1}$	100 000	0.179
LS100		$3.31 \cdot 10^{-4}$	64 310	0.181
LS10		$1.52 \cdot 10^{-1}$	100 000	0.193
Off	N_{1010}	$2.07 \cdot 10^{-1}$	100 000	0.182
LS100		$1.34 \cdot 10^{-1}$	100 000	0.183
LS10		$1.48 \cdot 10^{-1}$	100 000	0.196

(a) Spring-Damper

LS	**Node**	**Rel. error**	**Itrs.**	**Time / itr.** (ms)
Off	N	$1.96 \cdot 10^{-12}$	41 970	0.091
LS100		$5.72 \cdot 10^{-11}$	1740	0.095
LS10		$6.08 \cdot 10^{-12}$	13 360	0.104
Off	N_{110}	$3.27 \cdot 10^{-8}$	2 620	0.097
LS100		$2.71 \cdot 10^{-8}$	1 540	0.094
LS10		$2.71 \cdot 10^{-8}$	630	0.105
Off	N_{1010}	$5.55 \cdot 10^{-8}$	3 500	0.094
LS100		$5.09 \cdot 10^{-8}$	1 470	0.097
LS10		$5.09 \cdot 10^{-8}$	1 060	0.104

(b) Turbo Car

Table 8.2 Results of the line search on three different nodes, with line search interval 10 and 100 on both the Turbo Car example and the Spring Damper.



(a) No line search

(b) No line search

(c) Line search 100

(d) Line search 100

(e) Line search 10

(f) Line search 10

Figure 8.1 Performance of line search on Turbo Car example in root node. Notice the very different scales on the number of iterations.

	SPRING-DAMPER	TURBO CAR
Root	$1.8 \cdot 10^{-4}$	0.39
N_{111}	$8.4 \cdot 10^{-5}$	0.39
$N_{0001101}$	$3.7 \cdot 10^{-5}$	0.39

Table 8.3 Step size ρ for the different nodes and problems, when evaluating preconditioning on x.

and were for the Spring-Damper example selected[4] to 1, 10 or 100. For the Turbo Car example the L_i were selected[5] to 2.

Table 8.4 shows the performance of preconditioning on x, at different nodes. When the feature is turned off, no preconditioning is applied. Relative objective function errors[6] $|(\hat{J}^* - J^*)/J^*|$ are presented, as well as the number of iterations carried out and the average execution time per iteration.

Consider the Spring-Damper results in Table 8.4(a). Apparently preconditioning on x performs very well at the root node (and is well worth the extra execution time), but gives no improvement of the extremely slow convergence at nodes N_{111} and $N_{0001101}$. In fact, we observed a general bad performance except for the root node.

Now consider the Turbo Car results in Table 8.4(b). As the bounds $-2 \leq z(k) \leq 2$ are quite tight, we should not expect a vast improvement from the modest preconditioning $1/L_i = 1/2$. There may however be other benefits of having elements in y directly corresponding to elements in x (see Sections 2.4 and 5.4.1). All the nodes converge both with or without preconditioning on x, but the number of iterations differs quite much. At the root node and at node N_{111}, we get a considerable decrease in the number of iterations, but the extra execution time for each iteration makes the different approaches similar in performance. At node $N_{0001101}$ however, the number of iterations is instead increased, and we do not know why this is the case.

8.3.1 Different Performance at Different Nodes

Again consider the Spring-Damper example. The vast reduction of the number of iterations at the root node seems very promising, and the poor results at the other nodes are thus disappointing. In this section we do some effort in understanding why the performance can differ so much.

[4] According to the magnitude of the bounds of the corresponding states and inputs specified in (4.2).

[5] According to the magnitude of the bounds (C.1c)- (C.1d) of the "big M" auxiliary variables $z(k)$.

[6] Matlab's interior point solver is used with objective function tolerance 10^{-15}, to produce reference values J^*.

PC x	**Node**	**Rel. error**	**Itrs.**	**Time / itr.** (ms)
Off	N	$2.20 \cdot 10^{-4}$	100 000	0.18
On		$8.82 \cdot 10^{-8}$	380	0.90
Off	N_{111}	$1.95 \cdot 10^{-1}$	100 000	0.22
On		$2.84 \cdot 10^{-1}$	100 000	0.81
Off	$N_{0001101}$	$5.58 \cdot 10^{-1}$	100 000	0.23
On		$5.75 \cdot 10^{-1}$	100 000	0.82

(a) Spring-Damper

PC x	**Node**	**Rel. error**	**Itrs.**	**Time / itr.** (ms)
Off	N	$5.96 \cdot 10^{-12}$	41 970	0.09
On		$4.24 \cdot 10^{-12}$	11 150	0.24
Off	N_{111}	$4.15 \cdot 10^{-12}$	37 500	0.09
On		$4.24 \cdot 10^{-12}$	11 130	0.25
Off	$N_{0001101}$	$4.91 \cdot 10^{-7}$	4 170	0.10
On		$1.17 \cdot 10^{-6}$	23 150	0.25

(b) Turbo Car

Table 8.4 Performance of preconditioning on x, for the different nodes and systems. The PC x column indicates whether or not preconditioning on x is applied. The number of iterations (aborted at 100 000) is also presented, along with the current relative objective function error and the average execution time for each iteration. The MATLAB implementation of ADMM is used for these results.

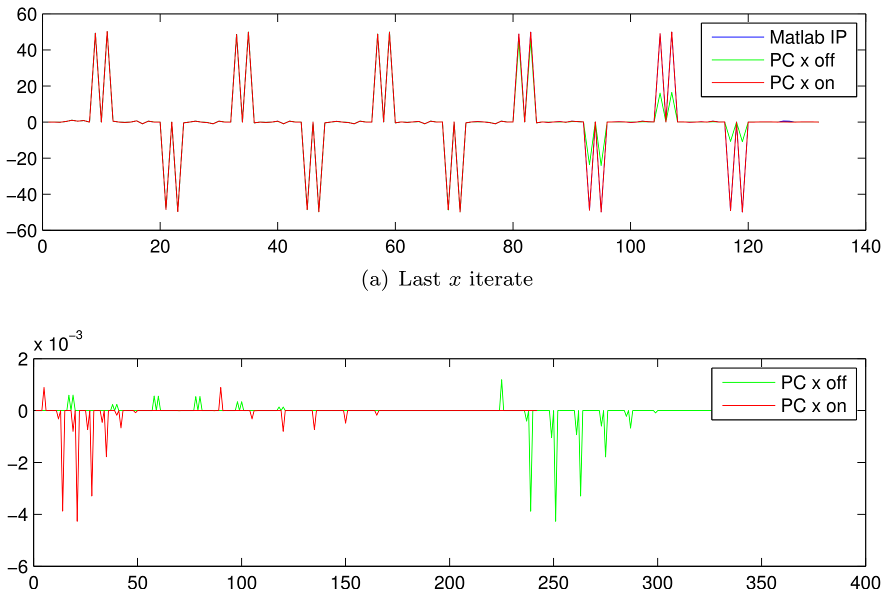
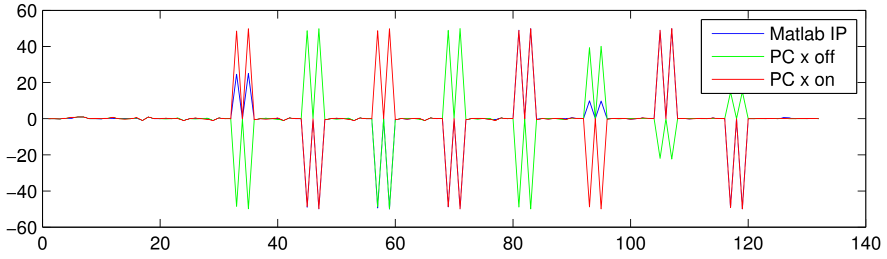


Figure 8.2 Last x and λ iterates for the root node (values on y-axis, vector index on x-axis). The red iterates are obtained with preconditioning on x, and the green iterates without. For x, there is also a blue reference solution, obtained by Matlab's interior point solver. The latter can however not be seen behind the red line.

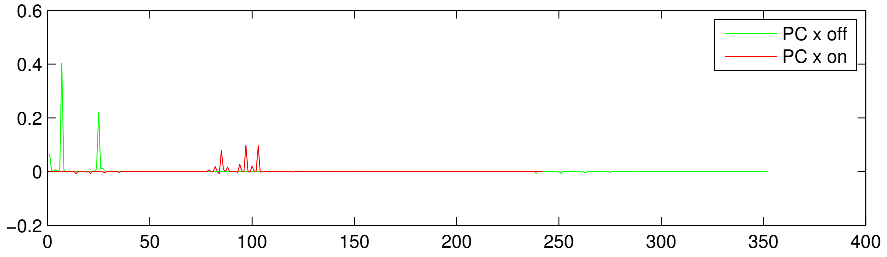
Figure 8.2(a) shows a reference solution[7] x^* at the root node, along with corresponding last iterates $\hat{x}^*$ and $\hat{x}^*_{PCx}$ of ADMM (with or without preconditioning on x). The high peaks correspond to the variables z_2 and z_4, for which $M = -m = 100$. At the root node, $\hat{x}^*$ has not yet (after 100 000 iterations) reached the high peaks of the "big M" auxiliary variables in x^*. However, applying preconditioning on x gives a solution $\hat{x}^*_{PCx}$ close to x^* (so that the latter cannot be distinguished underneath), after only 380 iterations.

When looking at the other nodes (see Figures 8.3 and 8.4), we see that ADMM is far from the optimal "big M" auxiliary variables, both with or without preconditioning on x. When it comes to the dual variables λ we do

[7] As not all costs for the Spring-Damper example are positive-definite, there may not be a unique solution. Therefore comparison between iterates should be done with precaution. In this case we compare iterates with very different magnitude at certain elements, and are confident (but not entirely sure) that for instance the "big M" aux variables need to be large at any solution.



(a) Last x iterate



(b) Last λ iterate

Figure 8.3 Last x and λ iterates for node N_{111} (values on y-axis, vector index on x-axis). The red iterates are obtained with preconditioning on x, and the green iterates without. For x, there is also a blue reference solution, obtained by Matlab's interior point solver.

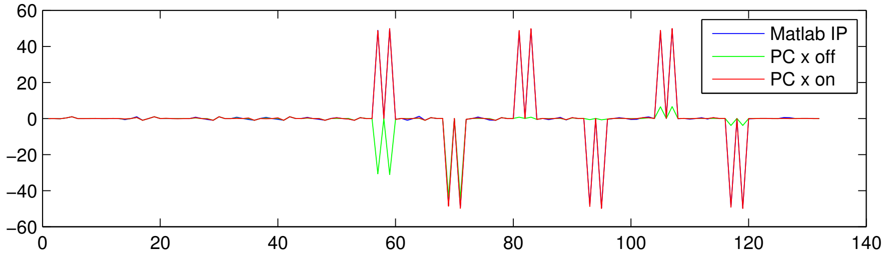
not have any reference solution λ^* to compare to[8]. Despite this, we do some reasoning based on the last dual variable iterates $\hat{\lambda}$ without preconditioning on x, and $\hat{\lambda}_{PCx}$ with preconditioning on x[9].

At N_{111} and $N_{0001101}$ we observe much larger values of $\hat{\lambda}^*$ as compared to the root node. We see that although also λ^*_{PCx} attains larger values at these nodes, there is still a considerable difference in magnitude between $\hat{\lambda}^*$ and $\hat{\lambda}^*_{PCx}$, which was not the case for the root node (see Figure 8.2(b)).

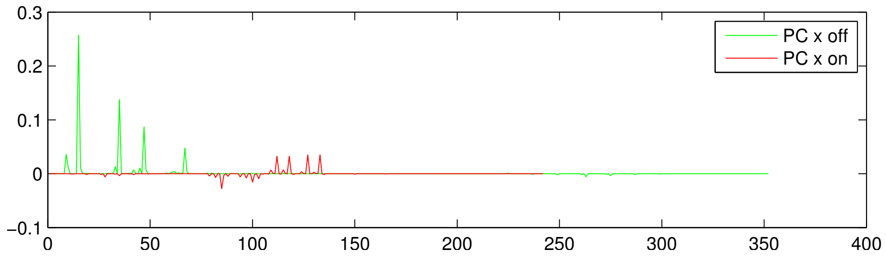
We believe that also the optimal λ^* should have very large values, as we observed a growth in λ. In that case, one possibility could be that the heavy preconditioning on the "big M" aux variables is somehow preventing

[8] We supply Matlab's interior point solver with the original QP on the form (2.1), and therefore neither y nor the constraints $EC_{\mathcal{I}}x = Ey$ with the corresponding dual variables λ_E are present.

[9] Note that the C matrix in $Cx = y$ is different depending on whether we apply preconditioning on x or not. Thus, $\hat{\lambda}^*$ cannot be compared to $\hat{\lambda}_{PCx}$ directly (and certainly not element-wise since they may be of different length). We will only consider the magnitude of $\hat{\lambda}$ and $\hat{\lambda}_{PCx}$.



(a) Last x iterate



(b) Last λ iterate

Figure 8.4 Last x and λ iterates for node $N_{0001101}$ (values on y-axis, vector index on x-axis). The red iterates are obtained with preconditioning on x, and the green iterates without. For x, there is also a blue reference solution, obtained by Matlab's interior point solver.

λ to take large steps, which indicates that the preconditioning needs to be modified. Another possibility is that the larger values of λ simply require a larger step size. By increasing E_{ii} for the large elements λ_i of λ, we did in fact get fast convergence at some additional nodes as well, which of course is promising. We did however not manage to find a general recipe for how to select E for the different nodes, in the limited time frame of ours. Neither did we have time to present the actual nodes and preconditionings that gave satisfactory results.

8.4 Simplified Line Search with Preconditioning on x

As mentioned in Section 5.6, we are not able to use the line search together with the preconditioning on x. However, we present results of the combination with the simplified version of the line search, without the troublesome cost $\mathcal{J}_3$ from Section 5.5. We have shown in the previous examples that the two methods perform well by them selves on many problems, but that neither will make all problems easily solvable. We have observed that the combination works well for many problem, however the method also has some drawbacks.

In Figure 8.5 we show how the solver is performing in each iteration with a combination of the different settings on the Spring Damper example. We see that the line search and preconditioning on x do indeed seem to work together. The performance is improved when any of the methods is added to the other.

It should be noted that this is not always the case and that this example was crafted specifically to illustrate this result. The sampling time was changed from 0.1 to 0.3 and the horizon chosen to 1. For most of the examples however, the method performed best with either only line search or only preconditioning on x. On the other hand, it seems like the combination performs better on average than choosing only one of the methods. For almost all the examples where one of the methods converged, so did the combination. We did however find one case where the solver didn't seem to converge with the combination. This is not too surprising since we are ignoring part of the optimality criteria in the line search.

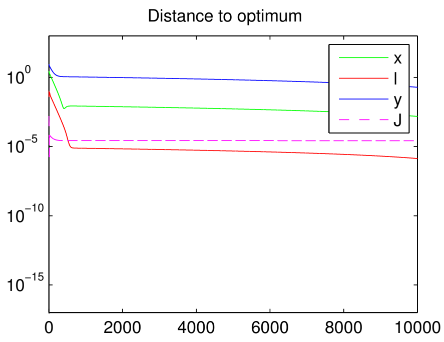
There are a few more noteworthy observations to be made from the plots. Firstly, we see common behavior that when only checking for the line search each 100th iteration, the jumps will be much rarer, but also much longer. The convergence is more even with the more frequent line search, but in this case, also slightly slower. Lastly, we see that even though the preconditioning on x is keeping a steady rate of convergence, the line search is still able to take a large step. The reduction from 10 000 iterations with barely any improvement to 300 iterations to reach machine precision is quite impressive.

8.5 Branch and Bound

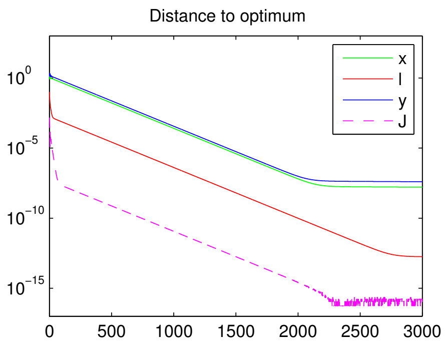
In this section we test several features, whose performance we evaluate by the execution time of solving an entire MIQP problem. Many of the features benefit from or rely on each other. Both the Spring-Damper and the Turbo Car example are considered, with horizon length $N = 10$. However, some features rely on our ADMM extensions and could only be evaluated for the Turbo Car due to lots of nodes not converging with ADMM for the Spring-Damper example.

Table 8.5(a) shows the performance when applying ADMM to the Turbo Car example with the different features:

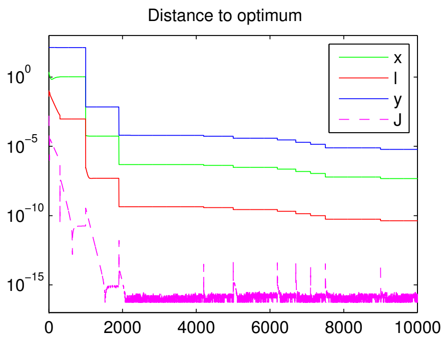
- $X\%$ Suboptimal Branch and Bound ($\mathbf{S}_{X\%}$) – (see Section 6.1.1).
- Warmstarting of nodes ($\mathbf{W_N}$) and trees ($\mathbf{W_T}$) – (see Section 7.2).
- Early Termination ($\mathbf{E}$) – (see Section 5.3).



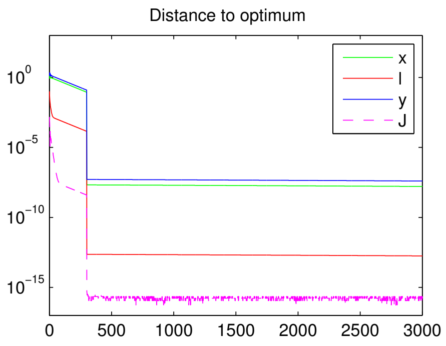
(a) **No** line search, **No** preconditioning on x.



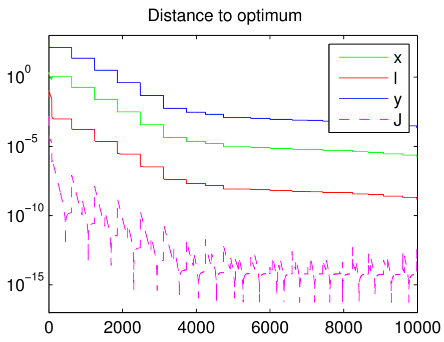
(b) **No** line search, preconditioning on x.



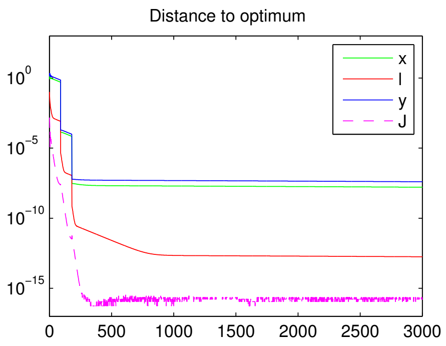
(c) Line search 100, **No** preconditioning on x.



(d) Line search 100, preconditioning on x.



(e) Line search 10, **No** preconditioning on x.



(f) Line search 10, preconditioning on x.

Figure 8.5 Plots of the convergence of the solver with and without preconditioning on x, as well as combined with the line search.

	Subopt.	**Time** (s)	**Nodes**	**Rel. error**	**Itrs./node**
No feats.	0%	0.67	43	$2.3 \cdot 10^{-7}$	3291
$\mathbf{W_N}$		0.55	43	$2.2 \cdot 10^{-7}$	2730
$\mathbf{W_N}$	1%	0.49 ± 0.03	42.2 ± 1.1	$2.2 \pm 0.0 \cdot 10^{-7}$	2756 ± 31
$\mathbf{W_N}\|\mathbf{W_T}$		0.48 ± 0.02	41.4 ± 0.6	$2.2 \pm 0.0 \cdot 10^{-7}$	2741 ± 21
$\mathbf{W_N}\|\mathbf{W_T}\|\mathbf{E}$		0.33 ± 0.02	41.4 ± 0.6	$2.2 \pm 0.0 \cdot 10^{-7}$	1632 ± 20
$\mathbf{W_N}$	10%	0.35 ± 0.10	32.2 ± 9.6	$4.8 \pm 2.3 \cdot 10^{-2}$	2561 ± 223
$\mathbf{W_N}\|\mathbf{W_T}$		0.22 ± 0.01	15.0 ± 0.0	$1.1 \pm 0.0 \cdot 10^{-2}$	3480 ± 1
$\mathbf{W_N}\|\mathbf{W_T}\|\mathbf{E}$		0.22 ± 0.02	15.0 ± 0.0	$1.1 \pm 0.0 \cdot 10^{-2}$	3209 ± 1

(a) The Turbo Car problem solved by our Branch and Bound with QPgen

	Subopt.	**Time** (s)	**Nodes**	**Rel. error**
No feats.	0%	0.517	87	$4.0 \cdot 10^{-8}$
No feats.	1%	0.485 ± 0.039	80.4 ± 6.1	$4.9 \pm 2.9 \cdot 10^{-3}$
$\mathbf{W_T}$		0.358 ± 0.002	59.0 ± 0.0	$1.8 \pm 0.0 \cdot 10^{-3}$
No feats.	10%	0.439 ± 0.127	75.1 ± 23.3	$3.0 \pm 2.3 \cdot 10^{-2}$
$\mathbf{W_T}$		0.166 ± 0.004	27.0 ± 0.0	$1.8 \pm 0.0 \cdot 10^{-3}$

(b) The Spring-Damper problem solved by our Branch and Bound with Gurobi's QP Solver

	Time (s)
Spring-Damper	0.0916
Turbo Car	0.0057

(c) Gurobi's MIQP Solver

Table 8.5 Performance of different features: Warmstarting of nodes ($\mathbf{W_N}$) and trees ($\mathbf{W_T}$), Early Termination (**E**) and Suboptimal Branch and Bound. When Suboptimal Branch and Bound is applied, the priority assignment is random according to (6.5). We have thus run the solver 15 times and presented estimated means and standard deviations.

As more features are added, we see that the total execution time decreases. Applying $\mathbf{W_N}$ gives a decrease from 0.67 s to 0.55 s. When also adding $\mathbf{S}_{1\%}$ or $\mathbf{S}_{10\%}$, we get an additional decrease to 0.49 s or 0.35 s, correspondingly. Then incorporating $\mathbf{W_T}$ gives a further improvement, but since it depends on Suboptimal Branch and Bound the decrease is much greater for the case $\mathbf{S}_{10\%}$ with more suboptimality. Finally, $\mathbf{E}$ gives an improvement for the case $\mathbf{S}_{1\%}$ but not[10] for $\mathbf{S}_{10\%}$. The decrease in execution time is due to the reduced number of iterations carried out at the nodes, which can be seen in the rightmost column. In neither case the number of nodes is decreased, which is the expected behavior.

Table 8.5(b) shows the performance of suboptimal Branch and Bound and warmstarting of the tree for the Spring-Damper example. The relaxations of the nodes are solved by Gurobi's QP solver [Gurobi Optimization, 2015], since the convergence of ADMM is bad at most nodes. The results are similar to Table 8.5(a).

Finally, Table 8.5(c) shows the performance of Gurobi's MIQP solver for reference. Clearly, it outperforms our methods[11].

8.6 Cutting Planes

We did not implement a full method to test how well infeasible nodes can be detected and used to automatically restrict the solutions. This would require a significant amount of work with a limited value to this thesis. However, we did create a specific example which illustrates the possible benefits clearly.

We consider the Turbo Car example as explained in Section 4.2. A counter c is used to keep track of how many uses are left of the binary turbo. The counter is decreased by 1 every time the turbo it is used. This means that a large portion of the nodes in the Branch and Bound tree will be infeasible. In practice, every node with more uses of the turbo than c is initialized to. If it is known that the system will be initialized with $c = 3$ uses of the turbo remaining, then we know that $c \leq 3$ for all future times.

It can thus be beneficial to in advance go through all the nodes and find the ones that will be infeasible, given the constraint $0 \leq c \leq 3$ on the initial state. The information about which nodes are infeasible can then be used to constrain the set of feasible solutions to the relaxed problems.

[10] The number of nodes are very few (15). The most probable explanation to the lack of improvement is that there are not many of these nodes to be discarded. Thus the improvement of discarding these very early may not contribute very much.

[11] It is however worth noting that Gurobi's MIQP solver is implemented in parallel, and executed on a quad-core machine. Our implementation runs on a single core, and although QPgen is implemented in C, the branch and bound algorithm is written in MATLAB.

It is easy to see for this example that we can formulate such a constraint without actually testing which nodes are infeasible:

$$\sum_k t(k) \leq 3 \tag{8.2}$$

where $t(k)$ is the binary input that regulates thruster use. The idea is that this constraint would disallow a relaxed solution like

$$[t(1)\ t(2)\ t(3)\ t(4)] = [1\ 1\ 0.6\ 0.6]$$

even if it satisfies the system dynamics. For this example however, the dynamics will indirectly generate a constraint identical to (8.2) through the system dynamics $c(k+1) = c(k) - t(k)$, the constraint $c(k) \geq 0$ and the initial condition $c(0) = 3$. We ran the Branch and Bound solver and, as expected, the same number of nodes, 44, was solved with and without this constraint. It may therefore seem like we have nothing to gain from adding this restriction.

However, if we change the initial state from $c = 3$ to $c = 3.5$ we get some interesting results. For the non-relaxed problem, very little has changed. The car can still only use an integer number of thrusts and thus has no use for the last half. We will get the same solution, and all the same nodes will be feasible. On the other hand, when solving the relaxed problems, the system dynamic will no longer force $\sum_k t(k) \leq 3$, but only $\sum_k t(k) \leq 3.5$, and the solution above would be accepted. This is exactly when the cutting planes can be useful, when *the relaxed problem enable solutions that are outside of the convex hull of the feasible binary solutions.*

With the new initial state $c(0) = 3.5$ and no extra constraints, the Branch and Bound method solved a total of 101 nodes instead of the previous 44, a considerable difference. If we manually add the extra constraint $\sum_k t(k) \leq 3$, which we know will be true for any feasible binary solution, we manage to reduce the number of solved nodes to 44 again.

We have thus managed to show how extra constraints, or cutting planes, may significantly improve the performance. We have however not studied extensively how these planes can be efficiently found and added and leave this to future research.

9

Concluding Remarks

9.1 Conclusions

First of all we have discovered that applying ADMM to the QP relaxations of some hybrid systems may give very poor convergence. Due to this observation we have extended ADMM with line search, as well as developed a method for preconditioning on certain variables, utilizing the block structure of "big M" formulations. Although both of these methods have given a vast improvement in convergence for some QP relaxations, there are still relaxations which remain intractable to solve.

In addition to improvements of convergence, we have also developed some Branch and Bound features which exploit the properties of ADMM in some way or another. While the features have proven advantageous (especially when combined), the execution time for solving an MIQP is still not in the same league as commercial solvers like Gurobi's MIQP solver. Since the full range of Branch and Bound techniques have not been the focus of this thesis, the inferior performance of our MIQP solver is expected.

9.2 Individual Contributions

Mattias has analyzed the behavior of ADMM, and developed the line search due to this analysis. He has also implemented the MATLAB version of ADMM, as well as warmstarting of nodes, cutting planes, and the ability to fix the binary variables by equality constraints.

Lucas has focused on the preconditioning on x and early termination features, but has also implemented the infeasibility check and warmstarting of the tree.

For the remaining parts, most of the work has been split between the two of us.

9.3 Future Work

We propose further investigation of how to select a good preconditioning when the "preconditioning on x" feature is applied. For the method to make a practical difference in the context of solving an entire MIQP, it needs to be determined how to select preconditioning for all the nodes. In particular the magnitude of the dual variables should be investigated, in the hope of discovering a potential structure regarding which dual variables are large at which nodes.

While the line search is performing well, we have no proof of convergence. Because the line search is not limited to MIQP problems, any such results for ADMM would be of high interest, especially because of the low cost compared to the potential improvement.

While ADMM has several advantages, it is indeed sensitive to the problem conditioning. As the ill-conditioned QP relaxations caused by the hybrid systems are not trivial to cope with, solvers less sensitive to the problem conditioning may be preferable. We suggest that the possibilities for parametric optimization in the context of quasi-Newton method's are investigated, as these methods are more compliant with the problem curvature.

A

MIQP Formulation of MIPC

This section gives the details about how the MIPC problem (3.11) in Section 3.3.3 can be formulated as an MIQP of the form (2.6). We build up x by stacking states, inputs and auxiliaries one time after another:

$$x = [x_0, u_0, w_0, x_1, u_1, w_1, \dots, x_N, u_N, w_N]^T. \tag{A.1}$$

In the same way we stack the references:

$$r = [r_x^0, r_u^0, r_w^0, r_x^1, r_u^1, r_w^1, \dots, r_x^N, r_u^N, r_w^N]^T.$$

The objective function is defined by

$$H = 2 \cdot \begin{bmatrix} Q &&&& \\ & Q &&& \\ && \ddots && \\ &&& Q & \\ &&&& Q_N \end{bmatrix}, \quad f = -2 \cdot Hr, \quad c = r^T Hr,$$

while the equality constraints are defined by

$$A_{\mathcal{E}} = \left[\begin{array}{cccccccccccc} I & 0 & 0 &&&&&&&&& \\ -A & -B_u & -B_{aux} & I & 0 & 0 &&&&&& \\ &&& -A & -B_u & -B_{aux} & I & 0 & 0 &&& \\ &&&&&&&& \ddots &&& \\ &&&&&& -A & -B_u & -B_{aux} & I & 0 & 0 \\ \hline E_x^{eq} & E_u^{eq} & E_w^{eq} & \cdots &&&&&&&& \\ &&&&&&&&& E_x^{eq} & E_u^{eq} & E_w^{eq} \end{array} \right]$$

and

$$b_{\mathcal{E}} = \left[\begin{array}{c} B_{aff} + x_0 \\ B_{aff} \\ \vdots \\ B_{aff} \\ \hline E_{aff}^{eq} \\ \vdots \\ E_{aff}^{eq} \end{array}\right].$$

The inequality constraints are defined by

$$C_{\mathcal{I}} = \left[\begin{array}{ccc} \boxed{E_x^{ineq} \quad E_u^{ineq} \quad E_w^{ineq}} & & \\ & \ddots & \\ & & \boxed{E_x^{ineq} \quad E_u^{ineq} \quad E_w^{ineq}} \\ \hline & I & \end{array}\right], \qquad \text{(A.2)}$$

and

$$b_{\mathcal{I}l} = \left[\begin{array}{c} -\infty \\ \vdots \\ -\infty \\ \hline l_x \\ l_u \\ l_w \\ \vdots \\ l_x \\ l_u \\ l_w \end{array}\right], \quad b_{\mathcal{I}u} = \left[\begin{array}{c} E_{aff}^{ineq} \\ \vdots \\ E_{aff}^{ineq} \\ \hline u_x \\ u_u \\ u_w \\ \vdots \\ u_x \\ u_u \\ u_w \end{array}\right].$$

B

Efficient Line Search

This appendix will outline how we implemented an efficient algorithm for the line search in ADMM. The goal is to solve the following problem

$$\min_t \mathcal{J}(x,y,\lambda) = \min_t \left(\mathcal{J}_1(x,y) + \mathcal{J}_2(x,\lambda) + \mathcal{J}_3(y,\lambda)\right) =$$
$$\min_t \left(\|Cx - y\|^2 + \left\|P_{\mathrm{K}}(Hx + f + C^T\lambda)\right\|^2 + \sum_{i=1..n} \left(\mathrm{clip}_{\tilde{c}_i(y)}(\lambda_i)\right)^2 \right),$$

given $y \in \mathcal{Y}$, where $(x,y,\lambda) = (x^0, y^0, \lambda^0) + t(\tilde{x}, \tilde{y}, \tilde{\lambda})$.
To do this we first find the maximum t such that $y^0 + t\tilde{y} \in \mathcal{Y}$, and denote it $t_{\max}$. Because the set $\mathcal{Y}$ is convex, the same set of constraints $b_{\mathcal{I}l,i} \leq y_i \leq b_{\mathcal{I}u,i}$ will be active on the entire interval $t \in (0, t_{\max})$. The functions $\tilde{c}_i(y)$ will therefore be constant on the interior, which means that we can write the function on the closed interval $t \in [0, t_{\max}]$ as

$$\mathcal{J}_3(y(t), \lambda(t)) = \begin{cases} \sum_{i=1..n} \left(\mathrm{clip}_{\tilde{c}_i(y_i^0)} \lambda_i(t)\right)^2 & \text{if } t = 0 \\ \sum_{i=1..n} \left(\mathrm{clip}_{\tilde{c}_i(y_i^0 + t_{\mathrm{int}}\tilde{y}_i)} \lambda_i(t)\right)^2 & \text{if } t \in (0, t_{\max}) \\ \sum_{i=1..n} \left(\mathrm{clip}_{\tilde{c}_i(y_i^0 + t_{\max}\tilde{y}_i)} \lambda_i(t)\right)^2 & \text{if } t = t_{\max} \end{cases},$$

where t_{int} is any $t \in (0, t_{\max})$. Minimization of $\mathcal{J}$ can therefore be done by comparing the values at $t = 0$ and $t = t_{\max}$ to the minimum on the interior. It is important to note that $\tilde{c}_i(y^0 + t_{\max}\tilde{y}) \subseteq \tilde{c}_i(y^0 + t_{\mathrm{int}}\tilde{y})$ which results in

$$\mathrm{clip}_{\tilde{c}_i(y^0 + t_{\max}\tilde{y})}(\lambda_i) \leq \mathrm{clip}_{\tilde{c}_i(y^0 + t_{\mathrm{int}}\tilde{y})}(\lambda_i).$$

There is therefore no risk in finding false minimums at the ends of the interval when minimizing the cost function on the interior.
For minimization on the interior, we look at the terms $\mathrm{clip}_{\tilde{c}_i(y_i^0 + t_{\mathrm{int}}\tilde{y}_i)}\lambda_i(t)$ in $\mathcal{J}_3$. Depending on which constraints on y_i are active and the sign of $\tilde{\lambda}_i$, they will each take one of three distinctive shapes, also shown in figure B.1:

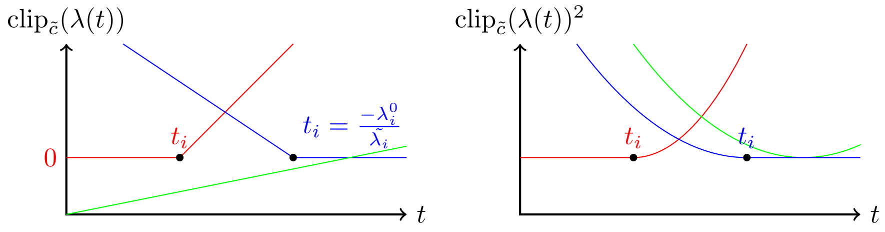


Figure B.1 Illustrations of the different types of functions in the sum $\mathcal{J}_3$. Red illustrates the type T_1, blue type T_2 and green type T_3.

T_1:

$$\begin{cases} 0 & \text{if } t \leq t_i = \frac{-\lambda_i^0}{\tilde{\lambda}_i} \\ (t-t_i)^2\tilde{\lambda}_i^2 & \text{if } t \geq t_i \end{cases}$$

T_2:

$$\begin{cases} (t-t_i)^2\tilde{\lambda}_i^2 & \text{if } t \leq t_i = \frac{-\lambda_i^0}{\tilde{\lambda}_i} \\ 0 & \text{if } t \geq t_i \end{cases}$$

T_3:

$$(\lambda_i^0 + t\tilde{\lambda}_i)^2 \quad \text{for all } t$$

We can therefore see that the function $\mathcal{J}_3$ (and thus $\mathcal{J}$) will be convex and piece-wise quadratic on the interior $t \in (0, t_{\max})$ The method we are using to minimize this function first computes all the points $t_i = \frac{-\lambda_i^0}{\tilde{\lambda}_i}$ for the terms of type T_1 and T_2. We then sort these times in increasing order and denote them, through abuse of notation, by changing their indices as $t_1, t_2, ...t_n$. If we let the function $\mathcal{J}$ be described on the interval $t \in (t_i, t_{i+1})$ by $p_i(t) = a_i t^2 + b_i t + c_i$, we can compute the constants recursively as

$$\begin{aligned} p_i(t) &= p_{i-1} + \mathrm{sgn}_i \cdot (t-t_i)^2\tilde{\lambda}_i^2 && \text{if } i > 0 \\ p_0(t) &= \sum_{j\in \mathrm{T}_2} (t-t_j)^2\tilde{\lambda}_j^2 + \sum_{j\in \mathrm{T}_3} (\lambda_j^0 + t\tilde{\lambda}_j)^2 + \mathcal{J}_1(t) + \mathcal{J}_2(t) \end{aligned} \tag{B.1}$$

where $\mathrm{sgn}_i = 1$ if $i \in \mathrm{T}_1$ and $\mathrm{sgn}_i = -1$ if $i \in \mathrm{T}_2$. An illustration of $\mathcal{J}$ is shown in figure B.2. It is now simple to calculate and evaluate the polynomials at the points t_i until $p_i(t_i) \leq p_{i+1}(t_{i+1})$. From convexity it is then clear that the minimum must either be at $t = \frac{b_i}{2a_i}, t = \frac{b_{i+1}}{2a_{i+1}}$ or t_i.

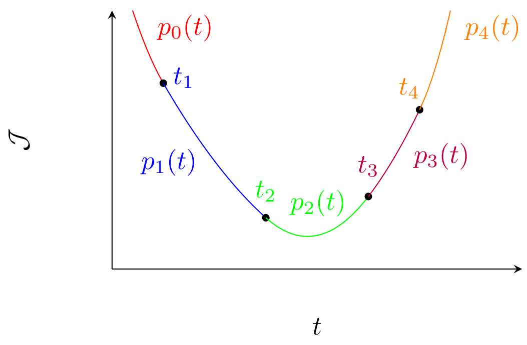


Figure B.2 An illustration of the cost function $\mathcal{J}$ in the line search.

C

Details for Test Cases

This section presents the details for the hybrid systems defined in Section 4.

C.1 Spring-Damper

The sampling interval of the system is $h = 0.1\,\mathrm{s}$. The masses are $m_1 = 2.29\,\mathrm{kg}$ and $m_2 = 2.044\,\mathrm{kg}$, and the damping coefficients are $d_1 = 3.12\,\mathrm{N/m/s}$ and $d_2 = 3.73\,\mathrm{N/m/s}$.

$$A_0 = \begin{bmatrix} 0.390 & 0.0709 & 0.610 & 0.0223 \\ -8.01 & 0.293 & 8.01 & 0.5693 \\ 0.672 & 0.0250 & 0.328 & 0.0667 \\ 8.68 & 0.638 & -8.68 & 0.206 \end{bmatrix} \qquad B_0 = \begin{bmatrix} 0.00182 \\ 0.0310 \\ 0.000296 \\ 0.0109 \end{bmatrix}$$

$$A_1 = \begin{bmatrix} 0.979 & 0.0928 & 0.0207 & 0.000669 \\ -0.402 & 0.853 & 0.402 & 0.0195 \\ 0.0229 & 0.000750 & 0.977 & 0.0907 \\ 0.440 & 0.0218 & -0.440 & 0.812 \end{bmatrix} \qquad B_1 = \begin{bmatrix} 0.00208 \\ 0.0405 \\ 0.00000833 \\ 0.000327 \end{bmatrix}$$

$$B_b = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \qquad Q_x = h \cdot \begin{bmatrix} 100 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 100 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad Q_x^N = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Q_u = Q_u^N = h \cdot \begin{bmatrix} 10^{-3} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad Q_w = Q_w^N = \begin{bmatrix} 0 \end{bmatrix}$$

C.2 Turbo Car

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B_u = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -1 & 0 \end{bmatrix}, \quad B_{aux} = \begin{bmatrix} 1 \\ 0.5 \\ 0 \end{bmatrix}, \quad B_{aff} = 0_{3\times 1} \tag{C.1a}$$

$$E_x^{ineq} = 0_{6\times 3}, \quad E_u^{ineq} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 3 & 2 \\ 3 & -2 \\ -3 & 1 \\ -3 & -1 \end{bmatrix}, \quad E_{aux}^{ineq} = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \\ -1 \\ 1 \end{bmatrix}, \quad E_{aff}^{ineq} = \begin{bmatrix} 2 \\ 2 \\ 3 \\ 3 \\ 0 \\ 0 \end{bmatrix} \tag{C.1b}$$

$$l_x = \begin{bmatrix} -50 \\ -10 \\ 0 \end{bmatrix}, \quad l_u = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad l_w = \begin{bmatrix} -2 \end{bmatrix} \tag{C.1c}$$

$$u_x = \begin{bmatrix} 50 \\ 10 \\ 5 \end{bmatrix}, \quad u_u = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad u_w = \begin{bmatrix} 2 \end{bmatrix} \tag{C.1d}$$

$$Q_x = Q_x^N = I_{3\times 3}, \quad Q_u = Q_u^N = \begin{bmatrix} 10^{-4} & 0 \\ 0 & 10^{-4} \end{bmatrix}, \quad Q_w = Q_w^N = \begin{bmatrix} 1 \end{bmatrix} \tag{C.1e}$$

All matrices E_{**}^{eq} are absent in this example.

Bibliography

Axehill, D. and A. Hansson (2006). “A mixed integer dual quadratic programming algorithm tailored for mpc”. In: *Decision and Control, 2006 45th IEEE Conference on*, pp. 5693–5698. DOI: `10.1109/CDC.2006.377215`.

Axehill, D., T. Besselmann, D. Martino Raimondo, and M. Morari (2014). “A parametric branch and bound approach to suboptimal explicit hybrid MPC”. *Automatica* **50**:1, pp. 240–246.

Bellman, R. (1957). *Dynamic Programming.* Princeton University Press, Princeton, NJ, USA.

Boyd, S., N. Parikh, E. Chu, B. Peleato, and J. Eckstein (2011). “Distributed optimization and statistical learning via the alternating direction method of multipliers”. *Found. Trends Mach. Learn.* **3**:1, pp. 1–122. ISSN: 1935-8237. DOI: `10.1561/2200000016`. URL: `http://dx.doi.org/10.1561/2200000016`.

Giselsson, P. (2014). “Improved fast dual gradient methods for embedded model predictive control”. In: *Proceedings of the 2014 IFAC World Congress*, pp. 2303–2309.

— (2015). *QPgen* MATLAB *toolbox.* [Accessed 2015-01-19]. URL: `http://www.control.lth.se/user/pontus.giselsson/qpgen/`.

Giselsson, P. and S. Boyd (2014a). “Diagonal scaling in douglas-rachford splitting and admm”. In: *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, pp. 5033–5039.

— (2014b). “Monotonicity and restart in fast gradient methods”. In: *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, pp. 5058–5063.

— (2014c). “Preconditioning in fast dual gradient methods”. In: *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, pp. 5040–5045.

Gurobi Optimization, I. (2015). *Gurobi optimizer reference manual.* URL: `http://www.gurobi.com`.

Kvasnica, M., P. Grieder, and M. Baotić (2004). *Multi-Parametric Toolbox (MPT).* URL: `http://control.ee.ethz.ch/~mpt/`.

MATLAB (2014). *version 8.3.0 (R2014a).* The MathWorks Inc., Natick, Massachusetts.

Mayne, D., J. Rawlings, C. Rao, and P. Scokaert (2000). "Constrained model predictive control: stability and optimality". *Automatica* **36**:6, pp. 789–814. ISSN: 0005-1098. DOI: `http://dx.doi.org/10.1016/S0005-1098(99)00214-9`. URL: `http://www.sciencedirect.com/science/article/pii/S0005109899002149`.

Mignone, D., A. Bemporad, and M. Morari (1999). "A framework for control, fault detection, state estimation, and verification of hybrid systems". In: *American Control Conference, 1999. Proceedings of the 1999.* Vol. 1, 134–138 vol.1. DOI: `10.1109/ACC.1999.782755`.

Raghunathan, A. U. and S. Di Cairano (2014). "Infeasibility detection in alternating direction method of multipliers for convex quadratic programs". In: *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, pp. 5819–5824. DOI: `10.1109/CDC.2014.7040300`.

Torrisi, F. D. and A. Bemporad (2004). "Hysdel-a tool for generating computational hybrid models for analysis and synthesis problems." *IEEE Trans. Contr. Sys. Techn.* **12**:2, pp. 235–249. URL: `http://dblp.uni-trier.de/db/journals/tcst/tcst12.html#TorrisiB04`.

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER´S THESIS *Date of issue* June 2015 *Document Number* ISRN LUTFD2/TFRT--5981--SE
Author(s) Mattias Fält Lucas Jimbergsson	*Supervisor* Pontus Giselsson, Dept. of Automatic Control, Lund University, Sweden Bo Bernhardsson, Dept. of Automatic Control, Lund University, Sweden (examiner) *Sponsoring organization*

Title and subtitle

Using ADMM for Hybrid System MPC

Abstract

Model Predictive control (MPC) has been studied extensively because of its ability to handle constraints and its great properties in terms of stability and performance [Mayne et al., 2000]. We have in this thesis focused on MPC of Hybrid Systems, i.e. systems with both continuous and discrete dynamics. More specifically, we look at problems that can be cast as Mixed Integer Quadratic Programming (MIQP) problems which we are solving using a Branch and Bound technique. The problem is in this way reduced to solving a large number of constrained quadratic problems. However, the use in real time systems puts a requirement on the speed and efficiency of the optimization methods used. Because of its low computational cost, there have recently been a rising interest in the Alternating Direction Method of Multiplies (ADMM) for solving constrained optimization problems. We are in this thesis looking at how the different properties of ADMM can be used and improved for these problems, as well as how the Branch and Bound solver can be tailored to accompany ADMM. We have two main contributions to ADMM that mitigate some of the downsides with the often ill-conditioned problems that arise from Hybrid Systems. Firstly, a technique for greatly improving the conditioning of the problems, and secondly, a method to perform fast line search within the solver. We show that these methods are very efficient and can be used to solve problems that are otherwise hard or impossible to precondition properly.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 1-84	*Recipient's notes*
Security classification		

http://www.control.lth.se/publications/