

Decentralised Motion Planning for Holonomic Multi-Agent Systems Using a Hybrid ADMM-CBF Approach: Application to Industrial Magnetic Levitation Platforms

Stan Servaes
Bavo Tistaert

Thesis submitted for the degree of
Master of Science in Mechanical
Engineering

Supervisor:

Prof. Dr. Ir. Jan Swevers

Assessors:

Dr. Ir. Wilm Decré

Ir. Maria Isabel Artigas Alfonso

Assistant-supervisors:

Ir. Ibrahim Ibrahim

Ir. Alejandro Gonzalez-Garcia

© Copyright KU Leuven

Without written permission of the supervisor and the authors it is forbidden to reproduce or adapt in any form or by any means any part of this publication. Requests for obtaining the right to reproduce or utilize parts of this publication should be addressed to Faculteit Ingenieurswetenschappen, Kasteelpark Arenberg 1 bus 2200, B-3001 Leuven, +32-16-321350.

A written permission of the supervisor is also required to use the methods, products, schematics and programmes described in this work for industrial or commercial use, and for submitting this publication in scientific contests.

Preface

This thesis year has been a challenging and busy, yet very educational and valuable experience. We would like to take a moment to thank the people who helped us through this journey.

First and foremost, we would like to thank our family, not just for being there this year, but throughout our entire education. Thank you for your endless support and interest in our thesis and other endeavours. It means a lot to us.

We would like to thank our promotor, Prof. Swevers. Your support and encouraging stance served as a great motivation for us. Thank you for providing plenty of freedom and opportunities to develop and expand the thesis.

To our mentors Ibrahim and Alex – thank you for the weekly advice, quick responses, and useful feedback. We learned a great deal from you this year.

To Joris – thank you for releasing CasADi 3.7.0 just when we needed it, and for the several hours of technical support. And to Koen and Heinz from Beckhoff for troubleshooting the XPlanar setup with us.

To our friends – thank you for sharing the difficult moments, the numerous thesis rants, and for interrupting our joint work sessions with perfectly timed distractions. No, we were not able to make the movers simply jump over each other.

In addition, I (Stan) would like to thank my housemates for being a constant source of support, patience, and humor during this thesis process. I would also like to give a special thanks to Catherine, for always being there when I needed it most. Your support during the most stressful times made all the difference.

Lastly, I (Bavo) would like to thank my two Charlottes. The first one being my girlfriend – thank you being incredibly supportive and caring to me this year. The second one being my cat that recently passed away – thank you for your loyal and talkative presence, which always eased the stress, even on the hardest days.

*Stan Servaes
Bavo Tistaert*

Contents

Preface	i
Abstract	iii
List of Figures and Tables	iv
List of Abbreviations and Symbols	viii
1 Introduction	1
1.1 Motivation and goals	2
1.2 Literature study	2
1.3 Overview and contributions	4
2 Preliminaries	9
2.1 Optimal control	9
2.2 Model predictive control	13
2.3 ADMM	14
2.4 Control barrier functions	21
3 Centralised motion planning	27
3.1 Free-time problem	27
3.2 Centalised MPC	30
4 Decentralised and hybrid methods	35
4.1 ADMM	35
4.2 ADMM-CBF	41
4.3 Neighbour selection	46
5 Practical implementation	49
5.1 The XPlanar system	49
5.2 Offline implementation details	52
5.3 Real-time implementation details	57
6 Results and discussion	63
6.1 Simulation results	63
6.2 Experimental results	72
7 Conclusion	77
7.1 Future research	78
Bibliography	81

Abstract

Industry requires more complex processes and logistics every day, sparking the rise of multi-agent systems. Here, multiple robots, vehicles, or machines operate in the same environment. This stimulates the need for multi-agent motion planning techniques that coordinate and plan the motion of these agents in an efficient and safe manner.

Therefore, this thesis presents a novel model predictive control (MPC)-based motion planning method for holonomic multi-agent systems, by combining a decentralised formulation of the alternating direction method of multipliers (ADMM) with a centralised formulation of control barrier functions (CBFs). The method is developed with an eye on, but not limited to, industrial magnetic levitation platforms such as the Beckhoff XPlanar system.

The classic centralised MPC approach is shown to scale with $\sim O(N^{2.96})$, which renders the method intractable with an increasing number of agents. While decentralised ADMM significantly reduces the time complexity, it cannot guarantee safety, i.e. there is a risk of collision. Therefore, the method is complemented with an extra optimisation problem that incorporates CBFs. The result is a safe method that scales with $\sim O(N^{1.82})$. This safety guarantee allows us to fix the number of ADMM iterations, admitting trade-off between performance and computational efficiency.

To further improve real-time performance, this work provides a C++ implementation of the hybrid method, which exploits the decentralised ADMM structure by multithreading the code. This reduces the computational time for 5 and 10 agents with 90.1% and 60.5% respectively. The hybrid method is then qualitatively and quantitatively compared with the centralised MPC method. Finally, a practical proof of concept is presented by implementing and testing the algorithms on the XPlanar system.

List of Figures and Tables

List of Figures

1.1	The Beckhoff XPlanar setup in the robotics lab. This industrial machine is used to transport goods in production lines.	1
1.2	Multi-agent motion planning on a non-convex arena. The trajectories of the agents must be planned towards their target without collisions. . . .	2
2.1	The Hessian of the Lagrangian for an OCP using the single and multiple shooting method. The black squares denote nonzero elements in the matrices. The OCP used in this comparison comes from an implementation of OCP 3.6 for 2 agents. It is clear that, compared with the single shooting method, the multiple shooting method has more decision variables, but entails a sparsity pattern in its Hessian which can be exploited. These figures are generated using Rockit [14].	12
3.1	The square movers of the XPlanar are modelled with circumscribed circles. This induces conservatism as the movers keep a distance between them that is larger than necessary when passing laterally.	29
4.1	Qualitative illustration of the planned trajectories over the ADMM iterations, applied to a case where two agents need to switch positions. First, the agents plan the most optimal trajectory, independent of each other and communicate their findings (OCP_1). Then, they each re-plan their own and each other's paths, purely such that they do not collide (NLP_1). Once again, they communicate their findings. As these paths might be dynamically infeasible, the agents again plan their own trajectory (OCP_2), independent of each other, but deviating as little as possible from these non-colliding paths of NLP_1 . As these new trajectories may still collide, the above process is repeated (NLP_2 , OCP_3 , ...).	39
4.2	The 2D trajectory of all 5 agents, resulting from decentralised ADMM. Even when performing $m = 50$ ADMM iterations per time step, the method has not fully converged yet. Hence, the resulting trajectories yield collisions, e.g. at $t = 0.4s$ and $t = 0.9s$	42

4.3	The paths that two agents take to switch places. When the CBF is tuned aggressively, they come closer to each other. When making the tuning more passive, the agents stay further away from each other. In both cases there are no collisions, as safety is guaranteed.	46
4.4	A neighbour selection case, where each agent needs to find their closest neighbour. Agent 3 selects agent 2, but agent 2 selects agent 1. Hence, they are not simultaneously in each other's neighbourhood.	48
5.1	The mover and tile types used in the experimental setup.	50
5.2	The XPlanar system in the robotics lab. This configuration is used in the experiments. Its local coordinate system is placed in the upper left corner of the tile configuration, with the x-axis and y-axis respectively pointing in the direction along its longest and shortest edge.	50
5.3	The XPlanar system architecture in the lab. The mover is a passive component floating above the tile. The tile contains all active components necessary to drive the movers. To that end, it receives commands from an industrial PC over an EtherCAT G fieldbus. The PC runs TwinCAT, which provides all necessary software functionalities (Section 5.1.2).	51
5.4	An example of splitting the arena into as large corridors as possible and defining the corridor transition points (red dots). Implemented on the arena of the Bechkhoff Xplanar	54
5.5	An example of how an agents tracks sub-targets to change corridors. The final sub-target is the destination of the agent. The transition points are heuristically chosen an lead to sub-optimal trajectories, but allow to consider non-convex arenas.	55
5.6	Communication between XPlanar, the C++ code, and the motion planning algorithm. In case of offline motion planning, the motion planning algorithm is replaced by a csv file containing a precomputed trajectory.	56
5.7	A schematic representation of the multithread scheme in which the parallel steps are followed up by a central communication and synchronization step.	58
5.8	Comparison of the different steps taken to accelerate the computation of the hybrid method. These tests are performed for 5 and 10 agents with ADMM-CBF performance mode. Every step decreases computational time, except parallel implementation in C++ for 10 agents.	59
5.9	The durations of each step in the hybrid method stacked on top of each other for every time step. This test was performed with ADMM-CBF fast mode for 5 agents. Even if the number of agents is below the number of hardware threads, the parallel implementation is not simply 5 times faster than the sequential implementation. This is because the sum of the longest OCP and NLP duration determine the computational time of a parallelised ADMM iteration.	61

5.10	Proposed diagram of how calculations can be distributed across multiple computers. A coordinator computer stores the trajectories and can forward them on request from the Beckhoff industrial PC.	62
6.1	The initial and target position of each agent for the case of 5 agents. . .	64
6.2	The initial and target position of each agent for the case of 10 agents. . .	64
6.3	The inter-agent distance for the case of 5 agents for free-time and centralised MPC. The red dashed line denotes the minimal distance between the centres of the agents before they collide. Both free-time and centralised MPC approach the threshold up to their safety margins, $5mm$ and $20mm$ respectively.	67
6.4	The inter-agent distance for the case of 5 agents for ADMM-CBF fast mode and performance mode. The red dashed line denotes the minimal distance between the centres of the agents before they collide. Fast mode exhibits a margin of $5mm$ equal to the safety margin, while performance mode had a margin of $10mm$ which is larger then its safety margin. . .	68
6.5	The inter-agent distance for the case of 5 agents for ADMM with $m = 50$ and $\mu = 70$. The red dashed line denotes the minimal allowed distance between two two agent centres. Even though 50 ADMM iterations are performed during each time step, there are still collisions.	69
6.6	Scalability plot of centralised MPC and ADMM+CBF (fast mode and performance) with 20 random cases for each number of agents. The blue line denotes the real-time capability threshold. The scalability is denoted by s . These tests were made in Python on a Linux computer with a 7th Gen Intel®Core™i7-7700K CPU @ 4.20GHz.	70
6.7	Scalability plot of the different implementations of the hybrid method ($m = 10$ and penalty parameter of 5) with 10 random cases for each number of agents. The blue horizontal line denotes the real-time capability threshold. The scalability exponent is denoted by s . These tests were run on a Linux pc with a 10th Gen Intel®Core™i7-10750H processor @ 2.60GHz, with 12 hardware threads.	72
6.8	Comparison between simulation and experiment on the XPlanar, performed for 5 agents with centralised MPC.	74
6.9	Comparison between simulation and experiment on the XPlanar, performed for 5 agents with ADMM-CBF performance mode.	75
6.10	Comparison between simulation and experiment on the XPlanar, performed for 5 agents with ADMM-CBF fast mode.	76

List of Tables

5.1	Relative decrease in computational time for each step that speeds up the computations. This corresponds to Fig. 5.8.	60
-----	--	----

5.2	The relative total time the program is busy solving the NLP. The larger the number of agents, the more time (with respect to the total time of the program) is spent solving the NLP. This test was done using the parallel C++ implementation.	60
6.1	Transit time and RCT for the 5 and 10 agent cases compared between the different motion planners. As the free-time problem is a method that is only performed once offline, the RCT has no meaning for this method. These tests were performed in Python on a Linux pc with a 7th Gen Intel®Core™i7-7700K CPU @ 4.20GHz.	65

List of Abbreviations and Symbols

This chapter provides an overview of the abbreviations and symbols used throughout the thesis, in order of appearance.

Abbreviations

Abbreviation	Description
MPC	Model Predictive Control
ADMM	Alternating Direction Method of Multipliers
CBF	Control Barrier Function
UAV	Unmanned Aerial Vehicle
RRT	Rapidly Exploring Random Tree
DFO	Distributed First-Order
DSCP	Distributed Sequential Convex Programming
HOCBF	High-Order Control Barrier Function
OCP	Optimal Control Problem
QP	Quadratic Program
QCQP	Quadratically Constrained Quadratic Program
NLP	Non-Linear Program
ACR	Average Consensus Residual
PLC	Programmable Logic Controller
PCB	Printed Circuit Board
ADS	Automation Device Specification
FIFO	First In, First Out
RCT	Relative Computation Time

Symbols

Symbol	Description
$\mathbf{x}(t)$	State vector of a dynamical system
$\mathbf{u}(t)$	Input vector of a dynamical system
J	Objective function of an OCP
J_0, J_f	Initial and final objective function of an OCP
T_f	Constant time horizon of an OCP
t	Time
τ	Non-dimensionalised time
$\mathbf{x}_0$	Initial condition on the state vector
$\mathbf{F}$	Vector function that describes the first-order differential equations of the dynamical system
$\mathbf{F}_d$	Vector function that describes the first-order differential equations of the dynamical system, after transcription
$\mathbf{g}$	Vector function that describes the stage-wise equality constraints of an OCP
$\mathbf{h}$	Vector function that describes the stage-wise inequality constraints of an OCP
N_{hor}	Number of control grid points of the transcribed OCP
$\mathbf{x}[k]$	Discretised state vector of a dynamical system
$\mathbf{u}[k]$	Discretised input vector of a dynamical system
J_d	Objective function of an OCP after transcription
Δt	Time step
T	Time horizon of an OCP as a decision variable
x, z	Decision vectors of an optimisation problem
f, g	Objective functions of an optimisation problem
A, B	Constant matrices describing linear constraints
b, c	Constant vectors describing linear constraints
$\mathcal{L}$	The Lagrangian of an optimisation problem
q	The dual function of an optimisation problem
δ^k	Step size of a gradient ascent iteration
$(\cdot)^k$	k-th iteration
i	Agent index
j	Neighbour index
N	Number of agents
f_i	Part of separable function belonging to agent i
x_i	Part of decision vector belonging to agent i
μ	Penalty parameter of the augmented Lagrangian
χ_i	Full copy of the decision vector, local to agent i
$\ \cdot\ ^2$	Euclidean norm

Symbol	Description
X_i	Set that encompasses the individual constraints on agent i
$g_{i,j}$	Pairwise coupled equality constraints
$\mathcal{N}_i$	Neighbourhood of agent i
z_i	Local copy of an agent's own decision variables
$z_{i,j}$	Local copy of an agent's neighbour decision variables
λ_i	Dual variables of an agent's consensus constraints w.r.t. its own trajectory
$\lambda_{i,j}$	Dual variables of an agent's consensus constraints w.r.t. its neighbour's trajectory
θ, η	Vector functions describing the state space of a generic nonlinear affine system
L	Lie derivative
$\mathcal{C}$	Superlevel set of a function
$\partial\mathcal{C}$	Boundary of the set $\mathcal{C}$
$\mathcal{L}_\mu$	Augmented Lagrangian
$\mathcal{L}_{\mu,i}$	Sub-Lagrangian of agent i
α	Class $\mathcal{K}$ function
ψ_i	Intermediate functions of the HOCBF
$\mathcal{C}_i$	Superlevel sets of the intermediate functions of the HOCBF
α_i	Class $\mathcal{K}$ functions belonging to the intermediate functions of the HOCBF
$\mathbf{u}^*$	Input generated by an unsafe controller
$\mathbf{x}_{0,i}$	State of agent i at the start of the OCP
$\mathbf{x}_{f,i}$	State of agent i at the end of the OCP
$\mathbf{A}, \mathbf{B}$	Continuous-time state space matrices representing the double integrator
$\mathbf{A}_d, \mathbf{B}_d$	Discrete-time state space matrices representing the double integrator
$\mathbf{p}$	Position vector
$\mathbf{v}$	Velocity vector
$\mathbf{a}$	Acceleration vector
v_{max}	Maximum allowed velocity magnitude
a_{max}	Maximum allowed acceleration magnitude
R	Radius of an agent
ϵ	Safety margin
ρ_{Nhor}	Number of control grid points per meter
d	longest distance between two agents at the start of a simulation
Q_i	Penalisation matrix on the states of agent i
R_i	Penalisation matrix on the inputs of agent i

Symbol	Description
$\mathbf{x}_{init,i}$	State of agent i at the start of its trajectory
$\mathbf{x}_{c,i}$	Current state of agent i at the start of its trajectory
$\mathbf{u}_{c,i}$	Current input of agent i at the start of its trajectory
J_i	Agent i 's objective
$\mathbf{z}_i(t), \mathbf{z}_{i,j}(t)$	Agent i 's local copies of its own and its neighbour's position functions
$\lambda_i(t), \lambda_{i,j}(t)$	Agent i 's dual variables for its consensus constraints
m	Number of ADMM iterations
m_{pre}	Number of ADMM pre-iterations
$h_{i,j}$	Collision avoidance constraints
$\text{cbf}_{i,j}$	Control barrier function constraint between agent i and j
K_1, K_2	Proportional constants of the linear class K functions used in the HOCBF
$p_{x,min}, p_{y,min}$	Lower position bounds of the arena
$p_{x,max}, p_{y,max}$	Upper position bounds of the arena
w	Width of an XPlanar mover
ρ_{agent}	Agent density
l_{square}	Side of the imaginary square in which the agents are placed in simulation
s	Scalability exponent

Chapter 1

Introduction

Complex and large-scale processes often rely on multi-agent systems. Here, multiple agents, such as machines, robots, and vehicles, operate in the same environment of the process, possibly working together. An important aspect of these operations is safety, which means that agents should not collide with each other. The growing trend in automatisisation has spread to this area, sparking the need for multi-agent motion planning algorithms. The purpose of the latter is to plan and coordinate the motion of all agents in an efficient and safe manner. Important applications include autonomous driving and ground vehicles [32], autonomous surface vessels [36], drone or UAV (Unmanned Aerial Vehicle) swarms [21], and transportation of goods in industrial production lines. This thesis focusses on the latter, with an eye on the Beckhoff XPlanar [6] as a transport system (Fig. 1.1).

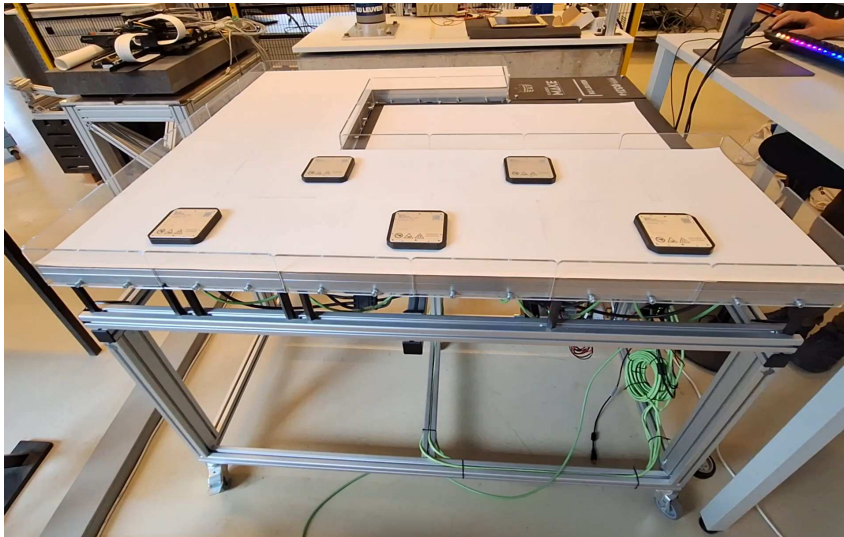


FIGURE 1.1: The Beckhoff XPlanar setup in the robotics lab. This industrial machine is used to transport goods in production lines.



FIGURE 1.2: Multi-agent motion planning on a non-convex arena. The trajectories of the agents must be planned towards their target without collisions.

1.1 Motivation and goals

Transportation of goods in the industry is often coordinated by pre-defined tracks with traffic rules. These are popular due to their inherent safety, robustness, and simplicity. However, they usually lack flexibility and are susceptible to traffic jams, which can significantly increase transit times and delay production. Consequently, there is much research devoted to these issues.

This thesis aims to develop, implement and test multi-agent motion planning algorithms focussing on the XPlanar system, but not necessarily limited to this specific application. The goal of these algorithms is to simultaneously generate trajectories for all agents that drive each of them to their target or destination in the most optimal way possible, without crashing into each other, as depicted in Fig. 1.2. Additionally, the motion planners should be maximally flexible by providing the possibility to replan in real-time, to ensure that they can be used in continuous operation with consecutively changing targets.

1.2 Literature study

There exist many motion planning techniques in literature, each with their own advantages and disadvantages. There are discrete planners such as rapidly-exploring random tree (RRT) [19], which searches the space by randomly expanding to neighbouring directions, and A* [16], which searches for the shortest path on a graph. However, the XPlanar can move and position goods anywhere on its arena with very high accuracy ($\leq 5\mu m$). As this flexibility should be exploited, continuous space methods are preferred. A popular continuous method is the potential field method [18]. This method models the targets as potential fields that attract the agent, while

obstacles repel them. However, this method suffers from deadlocks and local minima where agents can get stuck. There is also a growing trend of learning techniques in robotics. Due to their favourable scalability, they have been applied to multi-agent motion planning, especially when the number of agents is very high. However, due to the lack of sufficient formal proofs in this area, they often have to be coupled with existing motion planning techniques [24], [41].

Instead, this thesis makes use of optimal control techniques and model predictive control (MPC). MPC allows to plan in real-time while naturally complying with constraints, which is important for avoiding collisions, actuation limits, and staying within a specified arena. This also ensures the possibility to include kinodynamic constraints for trajectory planning, which is supported by the XPlanar. Due to its inherent optimality, the resulting trajectories are very efficient, leading to reduced transit times.

When using MPC, the most optimal trajectories would be obtained by solving one central optimisation problem for all the agents. However, the computational time for this problem often scales very badly with the number of agents, ruining the possibility for real-time motion planning. Moreover, since this strategy requires a central processing unit, it is often practically infeasible for on-board applications, e.g. when every agent has its own processing unit. In multi-agent systems, this issue is often addressed by the use of distributed optimisation methods [28], [29] which include distributed first-order methods (DFO), distributed sequential convex programming (DSCP), and the alternating direction method of multipliers (ADMM). Here, the optimisation problem is decoupled and the sub-problems are distributed over the agents, with the possibility for parallel computations.

ADMM [8], [15] holds the possibility to distribute heavily constrained (convex) optimisation problems. Distributed ADMM communicates the local variables of each sub-problem through a global variable or central collector, to inform the agents of each other's actions [10], [29], [28]. Decentralised ADMM takes it a step further and removes the central collector. Here, local variables are communicated directly between the sub-problems [32], [31], [26].

Even in distributed or decentralised algorithms, computation time can still scale poorly. The collision avoidance constraints are a large contributor to these problem due to their inherent non-convexity. Consequently, many have tried to reformulate these collision avoidance constraints. They have been linearised [26] and approximated by convex sets [40], [20], which usually introduces a lot of conservatism because it restricts the free space in which agents can plan. Other methods reduce conservatism by using hyperplanes [32], or even by obtaining an exact convex representation of the space [39]. However, they require extra optimisation variables, increasing the problem size.

Another strategy to limit scalability is to introduce neighbourhoods. The idea

is that an agent does not need to take every other agent into account when planning, e.g. if they are far away or moving to an opposite direction. Thus, an agent defines its neighbourhood by only considering the nearest neighbours, or those that are a threat to its safety. This way, the problem size and scaling can effectively be saturated. The most popular way to determine an agent's neighbours is by including those in a constant radius around it [26]. In the search for more representative strategies, schemes that also take into account the velocity of the agents and other criteria were developed [9], or even based on future collision detections [21].

Another important aspect in multi-agent motion planning, and in the industry in general, is safety. When the original motion planner is not inherently safe for any reason, e.g. critical constraints that cannot always be satisfied, control barrier functions (CBFs) [3] can provide a safety guarantee for systems with a relative degree of one. It does so by solving a computationally efficient quadratic program that modifies the trajectory in a minimally invasive way while satisfying a safety constraint. These techniques have therefore been applied to collision avoidance constraints in multi-agent systems [7], [35], where they can also be decentralised. More recently, this concept was generalised to high-order control barrier functions (HOCBF) [37], which apply to systems of any relative degree.

As ADMM is a method that solves convex optimisation problems, it cannot guarantee convergence when applied to non-convex problems. In this case, the method may be unsafe and CBFs could offer a solution. This idea was also very recently explored in [17], where first-order discrete-time CBFs are directly incorporated into the ADMM scheme. However, this thesis focusses on computing trajectories for systems such as the XPlanar, which run on a central computer and whose trajectory dynamics have a relative degree of 2. Therefore, unlike [17], we aim to exploit this architecture by developing a method that complements ADMM by adding a separate centralised step to the algorithm that uses continuous-time HOCBFs.

1.3 Overview and contributions

The above literature study shows the state-of-the-art of MPC based motion planning and other relevant optimisation techniques, along with its specific challenges. It is the starting point on which this work is built. This section presents a detailed overview of the thesis and its contributions.

Preliminaries This chapter introduces all concepts and mathematics that are used for the motion planning algorithms. It starts with the basics of optimal control and MPC. Based on [25] and [12], we discuss how to formulate and solve optimal control problems (OCPs), and how they may be used in MPC. Then, ADMM is discussed, closely following [8]. To build sufficient understanding and provide motivation for

this method, dual ascent and dual decomposition are discussed. Then, ADMM is introduced as an augmented Lagrangian generalisation of dual ascent. Lastly, methods to distribute and eventually decentralise ADMM are discussed. Finally, CBFs and HOCBFs are introduced based on [3], [11], and [37], focussing on how they can guarantee safety and how they can be incorporated in an optimisation problem.

Centralised motion planning This chapter explores centralised optimisation techniques as a possible strategy for multi-agent motion planning on the XPlanar, as it is technically feasible since this system has one central computer. Specifically, it looks at the so-called *free-time problem* to establish a benchmark for time-optimality. Then, a centralised MPC scheme is developed as a first actual motion planning algorithm. Here, agents are approximated as circles to avoid discontinuities in the collision avoidance constraints. Rather than trying to “convexify” them, the latter are formulated by simply imposing a minimum Euclidean distance between all agents, to avoid further conservatism. After implementing the strategies in Python, their tunings are established. As formally shown in Chapter 6, the centralised techniques do not scale well and show little promise for real-time applications, except in small-scale systems.

Decentralised and hybrid methods To cope with the unfavourable scalability of the centralised methods, this chapter decentralises the problem using ADMM. However, the problem is inherently non-convex due to the collision avoidance constraints, while ADMM is only guaranteed to converge for convex problems [8]. Moreover, to use this strategy in real-time applications, only a limited number of iterations is performed per time step, with the idea that it eventually converges over the time steps [32]. Consequently, the method cannot guarantee safety.

Instead of reformulating the collision avoidance constraints, a hybrid method is proposed that complements the original strategy with an extra quadratic program that uses a HOCBF, which guarantees safety with little extra computational cost. Taking advantage of the central architecture of the XPlanar and the fact that this quadratic program is computationally efficient, this extra step is performed in a centralised fashion.

Both methods are once again implemented in Python, although still as a sequential scheme. As shown in Chapter 6, the hybrid method is promising for real-time applications in medium-scale systems. Lastly, this chapter aims to couple the hybrid method with a neighbour selection scheme to limit the scalability, with the intention of going to large-scale systems. It is observed that ADMM is not well suited to be coupled with such schemes whenever an agent’s neighbourhood changes frequently.

Practical implementation This chapter discusses the XPlanar and the integration of the algorithms on this system. After a brief discussion on its hardware and software, the necessary steps are taken to perform offline motion planning on the system.

First, the algorithms must be paired with boundary constraints such that they do not fly off the arena. The constraints for convex arenas are discussed. Motion planning in more general non-convex arenas is significantly more challenging. To this end, a multi-corridor strategy, where the agents navigate between convex sub-arenas, is presented as a possible extension. Then, to link the frequency of the motion planner with that of the XPlanar, the trajectories are interpolated. Next, a buffer communication protocol is explained, which the XPlanar uses to properly coordinate the timing between receiving and applying trajectory samples.

This chapter also discusses the steps necessary to prepare the hybrid method for real-time motion planning. This includes translating the code to C++ and developing a multithreading scheme for parallel computation. When running the multithreaded algorithm on the XPlanar, it turned out that its computer was way slower compared to the laptop on which simulations were performed. Therefore, instead of performing the calculations on the Beckhoff XPlanar industrial PC, a scheme is proposed to perform the calculations on another computer (or multiple computers) and then forward them to the Beckhoff industrial PC. However, this scheme has not yet been implemented.

Results and discussion This chapter contains a simulation-based comparative analysis of all algorithms, as well as experimental tests on the XPlanar. First, all algorithms are quantitatively compared on performance, safety, and scalability. Then, the real-time capability is checked to see the maximum number of agents for which the trajectories can be planned in real-time. It is shown that the resulting algorithm is fast enough for real-time computations up to 13 agents. Finally, the motion planning algorithms are tested in a practical setting by pre-computing the trajectories and applying them to the XPlanar. It turns out that the trajectories which the motion planners compute are well suited for the system and its applications. These tests were also performed for first-year students in the context of a “demo-day”.

Conclusion In this final chapter of the thesis, the main results are repeated and critically discussed. Several possibilities for future research are also given to comple-

ment the thesis.

In summary, the main contributions of this thesis are:

- A novel hybrid multi-agent motion planner which combines decentralised ADMM with centralised HOCBFs, along with a multithreaded C++ implementation which is fast enough for real-time motion planning in medium-scale systems.
- A heuristic approach to perform motion planning in rectangular non-convex arenas, such as the XPlanar configuration in the experimental setup.
- A qualitative discussion and quantitative comparison between centralised and decentralised methods for multi-agent motion planning, based on performance, safety, and scalability.
- A proof of concept for offline multi-agent motion planning on the XPlanar.

Chapter 2

Preliminaries

The motion planning algorithms presented in this thesis make heavy use of optimisation and control techniques. Therefore, this section introduces the basic mathematical results from these domains, limited to the extent needed to understand the rest of the thesis. First, it discusses optimal control problems and how they can be incorporated into a model predictive control scheme. Then, the alternating direction method of multipliers (ADMM) is presented as a strategy to solve multi-agent optimisation problems in a distributed manner. To provide this link with sufficient understanding, the method is introduced in a stepwise manner with gradually increasing complexity. Finally, the theory behind control barrier functions (CBF's) is discussed, with an eye on why they can guarantee safety and how to use them.

2.1 Optimal control

Optimal control is a powerful technique for generating trajectories for a dynamical system. This section provides an overview on how to formulate and solve these typical problems. A special version, the so-called *free-time problem*, is also introduced. The discussions in this section closely follow [25] and [12].

2.1.1 Formulation

The idea is to solve an appropriate optimisation problem to obtain a series of control inputs $\mathbf{u}(t)$ that ensure the states of the system $\mathbf{x}(t)$ follow an optimal trajectory. This specific type of problem is called an optimal control problem (OCP) and has the following general form [12]:

$$\min_{\mathbf{x}(t), \mathbf{u}(t)} J_0(\mathbf{x}(0)) + \int_0^{T_f} J(\mathbf{x}(t), \mathbf{u}(t), t) dt + J_f(\mathbf{x}(T_f)) \quad (2.1a)$$

$$\text{s.t. } \mathbf{x}(0) = \mathbf{x}_0 \quad (2.1b)$$

$$\dot{\mathbf{x}}(t) = \mathbf{F}(\mathbf{x}(t), \mathbf{u}(t), t) \quad (2.1c)$$

$$\mathbf{g}(\mathbf{x}(t), \mathbf{u}(t), t) = 0 \quad (2.1d)$$

$$\mathbf{h}(\mathbf{x}(t), \mathbf{u}(t), t) \geq 0 \quad (2.1e)$$

With $J, J_0, J_f, \mathbf{F}, \mathbf{g}, \mathbf{h}$ twice continuously differentiable. The objective (Equation 2.1a) is an integral over time of some function J of the state vector $\mathbf{x}(t) \in \mathbb{R}^{n_x}$, the input vector $\mathbf{u}(t) \in \mathbb{R}^{n_u}$ and time t . It also includes an initial and terminal cost J_0 and J_f . This function decides what should be minimised and thus depends on the application. It becomes more concrete in Chapter 3. The integral runs from 0 to T_f , where T_f denotes the control horizon, also called the prediction horizon. It constitutes how far in the future the OCP predicts and computes the motion.

To determine the evolution of a dynamical system, one is usually given an initial condition (unless finding the initial condition is part of the problem). In that case, this condition must also be incorporated as a constraint in the OCP (Equation 2.1b). The dynamic constraints (Equation 2.1c) ensure that the solution of the OCP is physical, meaning that it satisfies the governing kinematic or dynamic equations. The evolution of the system can usually be described by a set of first order differential equations, represented by $\dot{\mathbf{x}}(t) = \mathbf{F}(\mathbf{x}(t), \mathbf{u}(t), t)$. Lastly, the OCP can have stagewise equality 2.1d and/or inequality 2.1e constraints (that have to hold over the whole trajectory). These are usually related to physical limits of the system or boundaries constituted by the environment. Constraints that only hold at the end of the trajectory are also sometimes necessary, these are called terminal constraints and are used in Section 2.1.4.

2.1.2 Transcription

Since the states and inputs are functions of time, OCP 2.1 is infinite-dimensional. The process of converting the OCP to a form that can be solved is called *transcription*. There are many transcription methods, but this thesis only makes use of the direct multiple shooting method due to its relevance for large scale systems. This is a method that aims to discretise the OCP, resulting in the following finite-dimensional problem [12]:

$$\min_{\mathbf{x}[k], \mathbf{u}[k]} J_0(\mathbf{x}[0]) + \sum_{k=0}^{N_{hor}-1} J_d(\mathbf{x}[k], \mathbf{u}[k], k) + J_f(\mathbf{x}[N_{hor}]) \quad (2.2a)$$

$$\text{s.t. } \mathbf{x}[0] = \mathbf{x}_0 \quad (2.2b)$$

$$\mathbf{x}[k+1] = \mathbf{F}_d(\mathbf{x}[k], \mathbf{u}[k], k) \quad (2.2c)$$

$$\mathbf{g}(\mathbf{x}[k], \mathbf{u}[k], k) = 0 \quad (2.2d)$$

$$\mathbf{h}(\mathbf{x}[k], \mathbf{u}[k], k) \geq 0 \quad \forall k \in \{0, \dots, N_{hor} - 1\} \quad (2.2e)$$

Where the subscript $(\cdot)_d$ denotes the discrete equivalent of the function. The method constructs a control grid, which divides the time horizon in a finite number of time steps Δt . The function $\mathbf{u}(t)$ is then approximated by a finite series of values $\mathbf{u}[k]$, which represent the values of the control signal on the subsequent control grid points. In between, a zero order hold condition is assumed. The number of grid points is denoted by N_{hor} . Thus, k runs from 0 to $N_{hor} - 1$. It also determines the resolution of the grid:

$$\Delta t = \frac{T_f}{N_{hor}} \quad (2.3)$$

The larger N_{hor} , the more accurate the discretisation. However, it also determines the number of decision variables and thus the computation time for the solver. The objective function and dynamic constraints are discretised using a numerical integrator. In this thesis, the runge-kutta-4 method is used.

When the OCP is transcribed using the multiple shooting method (as in OCP 2.2), every state and input on every grid point becomes a decision variable. It is possible to eliminate the states by plugging in the dynamic constraints (Equation 2.2c) everywhere. This strategy is called the single shooting method. It would significantly reduce the number of decision variables, but the expressions for the objective and remaining constraints become much more complicated and expensive to evaluate. Overall, the multiple shooting method is generally still preferred for large-scale systems, because the variables, and thus also possible non-linearities, are spread over a larger number of more manageable expressions. The problem size is larger, but sparse. As an example, Fig. 2.1 shows the structure of the hessian of a typical OCP for the single and multiple shooting method. This sparsity pattern of the latter can be exploited by the solver.

2.1.3 Solving

Once the objective and all the constraints are determined for a specific problem, the OCP along with all its expressions has to be cast in a standard form that the employed solver can recognise. However, this step can be bypassed by using (open-source) software toolkits such as CasADi [4] and Rockit [14] (layer built on

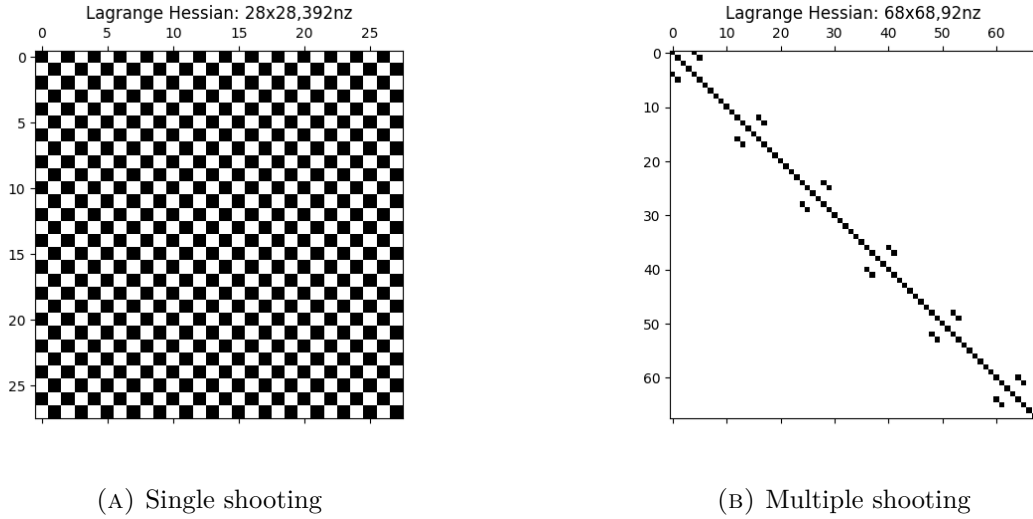


FIGURE 2.1: The Hessian of the Lagrangian for an OCP using the single and multiple shooting method. The black squares denote nonzero elements in the matrices. The OCP used in this comparison comes from an implementation of OCP 3.6 for 2 agents. It is clear that, compared with the single shooting method, the multiple shooting method has more decision variables, but entails a sparsity pattern in its Hessian which can be exploited. These figures are generated using Rockit [14].

top of CasADi, specifically for optimal control). They allow for fast and intuitive implementation of optimisation problems and OCPs, and the ability to call built-in solvers. Rockit also automatically takes care of the transcription. Because of these (and more) features, the tools are extensively used in this thesis.

The last step is actually solving the problem. To this end, an appropriate solver needs to be chosen. This thesis employs the following ones:

- IPOPT [34]: This is a very robust, general purpose nonlinear program solver that can handle large scale problems. It uses a primal-dual interior-point method and filter techniques for line search. It is the standard solver in CasADi. It can be used for many types of optimisation problems, making it very flexible. Consequently, it is quite slow compared to specialised solvers.
- FATROP [33]: This is a very fast nonlinear optimal control solver, inspired by IPOPT. It trades generality for speed by exploiting the typical structure of an OCP.
- qpOASES [13]: This is a reliable quadratic program solver that uses an active set method. It is well suited for model predictive control schemes.

The above solvers are all open source and interfaced with CasADi. Each has advantages and disadvantages. Thus, it is important to select a solver that is tailored for the specified application.

2.1.4 Free-time problem

The free-time problem is a special type of OCP where the time horizon is a decision variable as well. This section discusses the case where the objective directly minimises the transit time of the motion, as in [22]. The OCP is formulated as follows [12]:

$$\min_{\mathbf{x}(\tau), \mathbf{u}(\tau), T} T \quad (2.4a)$$

$$\text{s.t. } \mathbf{x}(0) = \mathbf{x}_0, \mathbf{x}(1) = \mathbf{x}_f \quad (2.4b)$$

$$\dot{\mathbf{x}}(\tau) = T \mathbf{F}(\mathbf{x}(\tau), \mathbf{u}(\tau), \tau T) \quad (2.4c)$$

$$\mathbf{g}(\mathbf{x}(\tau), \mathbf{u}(\tau), \tau T) = 0 \quad (2.4d)$$

$$\mathbf{h}(\mathbf{x}(\tau), \mathbf{u}(\tau), \tau T) \geq 0 \quad (2.4e)$$

$$\forall \tau \in [0, 1] \quad (2.4f)$$

The objective function is simply T , which denotes the total transit time. The decision variables are the state vector $\mathbf{x}$, the input vector $\mathbf{u}$ and the transit time T itself.

The states and inputs are still functions of time. However, OCP 2.4 uses non-dimensionalised time, denoted by τ . This trick must be used, as the transit time T , which also serves as the time horizon in this problem, is unknown. This makes it impossible to know on beforehand where to evaluate the variables for possible terminal conditions. More importantly, it would be impossible to discretise the problem. For a given number of discretisation points N_{hor} , the time resolution dt would be unknown. Thus, by defining

$$\tau = \frac{t}{T} \quad (2.5)$$

the initial and final conditions are always evaluated at $\tau = 0$ and $\tau = 1$. Furthermore, the time resolution is now given by $d\tau = N_{hor}^{-1}$.

In the dynamic constraints 2.4c, an extra factor of T pops up because the derivative is also non-dimensionalised. The rest of the constraints are the same as in OCP 2.1.

2.2 Model predictive control

The field of model predictive control (MPC), both theory and design, is very comprehensive. It produces controllers that handle unstable, non-linear and constrained

systems. However, in this thesis, MPC is not used to design an actual controller but rather as a tool for motion planning. Therefore, this section only briefly introduces the concept of MPC, without going into much detail about the underlying theory. For further reading, see [25], on which this section is based.

Solving an OCP is an inherently open-loop strategy. To obtain the optimal trajectory, the problem is solved once offline. MPC, which is the closed-loop version, recomputes the OCP in every time step. The general MPC strategy is summarised in the following actions, which have to be executed in every time step:

- Measure the current state of the system and use this as initial condition $\mathbf{x}_0$ in Constraint 2.1b.
- Solve OCP 2.1.
- Only extract the input at $t = 0$ ($\mathbf{u}[0]$) from the solution and apply it to the system.
- Update the horizon of the OCP: e.g. after the first time step, the horizon runs from $t = \Delta t$ to $t = T_f + \Delta t$, instead of from $t = 0$ to $t = T_f$. In practice, one can just assume a local time reference for the OCP, which makes sure that the mathematical representation of the horizon stays the same.

Here, the OCP's are solved in real-time and this process can continue indefinitely as long as the computation can be done within each time step. This strategy allows the use of the same OCP for each trajectory within a certain application, apart from some parameters such as the initial condition or certain targets in the objective, but the structure does not change. Furthermore, the time horizon no longer needs to cover the entire trajectory. However, the choice of T_f is still important. The larger the horizon, the more optimal the resulting trajectory. If the OCP is able to look further into the future, it anticipates and takes into account more events. Then, the inputs are adapted accordingly, and the trajectory becomes more efficient. Lastly, this strategy allows for feedback between the subsequent steps. In fact, it inherently uses state feedback for its initial condition, but it is also possible to update other parameters to react to unexpected events (e.g. unexpected obstacles or changes in target positions).

2.3 ADMM

The alternating direction method of multipliers (ADMM) is widely used for distributed optimisation, making it suitable for large scale systems. There are many ways to derive, formulate, and interpret the method, and there are many variations

[38]. After some definitions, this section first introduces dual ascent, dual decomposition and their relevance for multi-agent systems. To deal with the shortcomings of dual ascent, the general augmented Lagrangian interpretation of ADMM is discussed. Finally, two special versions are introduced, restoring the link with multi-agent systems. The first three subsections closely follow [8]. The last subsection is based on [32].

2.3.1 Dual ascent and dual decomposition

Following [8], dual ascent and dual decomposition are discussed first. Dual ascent is a method that aims to solve the dual problem of a linear equality constrained problem:

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & Ax = b \end{aligned} \quad (2.6)$$

With $x \in \mathbb{R}^n$ as the decision vector. The Lagrangian is [8]:

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^T (Ax - b) \quad (2.7)$$

Consequently, the dual function is [8]:

$$q(\lambda) = \inf_x \mathcal{L}(x, \lambda) \quad (2.8)$$

And the dual problem [8]:

$$\max_{\lambda} q(\lambda) \quad (2.9)$$

Which is an unconstrained maximisation problem, due to the fact that there are no inequality constraints in Problem 2.6. Subsequently, one can simply apply gradient ascent to Problem 2.9. In fact, evaluating the gradient of the dual function requires no extra computations. The theoretical expression of the dual function is [8]:

$$\begin{aligned} q(\lambda) &= \mathcal{L}(x^*(\lambda), \lambda) \\ \text{with } x^*(\lambda) &= \arg \min_x \mathcal{L}(x, \lambda) \end{aligned} \quad (2.10)$$

Following [23], the gradient can be found by using the chain rule:

$$\begin{aligned} \nabla q(\lambda) &= \nabla \mathcal{L}(x^*(\lambda), \lambda) \\ &= \nabla_x \mathcal{L}(x^*(\lambda), \lambda) \nabla x^*(\lambda) + \nabla_{\lambda} \mathcal{L}(x^*(\lambda), \lambda) \\ &= \nabla_{\lambda} \mathcal{L}(x^*(\lambda), \lambda) \\ &= Ax^*(\lambda) - b \end{aligned} \quad (2.11)$$

Where the third equality follows from Lagrangian stationarity ($\nabla_x \mathcal{L}(x^*(\lambda), \lambda) = 0$). Thus, the dual ascent method employs the following two steps in every iteration [8]:

$$\begin{aligned} x^{k+1} &= \arg \min_x \mathcal{L}(x, \lambda^k) \\ \lambda^{k+1} &= \lambda^k + \delta^k (Ax^{k+1} - b) \end{aligned} \quad (2.12)$$

Where δ^k denotes the step size of the gradient ascent. Starting from some initial guesses x^0 and λ^0 , this method continuously solves Problem 2.8 (the Lagrangian minimisation) to be able to evaluate the gradient of the dual function, before doing an actual gradient ascent iteration on Problem 2.9 (the dual update).

This method is relevant in multi-agent optimisation due to its ability to distribute computations when the cost function is separable into a sum of N functions, which could each represent an agent's own objective that only depends on its own decision variables x_i [8]:

$$\begin{aligned} \min_x \quad & \sum_{i=1}^N f_i(x_i) \\ \text{s.t.} \quad & Ax = \sum_{i=1}^N A_i x_i = b \end{aligned} \quad (2.13)$$

With $x = [x_1, \dots, x_i, \dots, x_N]^T$ and $A = [A_1, \dots, A_i, \dots, A_N]$. The Lagrangian can now be written as follows:

$$\begin{aligned} \mathcal{L}(x_i, \lambda) &= \sum_{i=1}^N f_i(x_i) + \lambda^T \left(\sum_{i=1}^N A_i x_i - b \right) \\ &= \sum_{i=1}^N (f_i(x_i) + \lambda^T A_i x_i) - \lambda^T b \end{aligned} \quad (2.14)$$

It is clear that in this case it is possible to do the Lagrangian minimisation over each x_i independently. This opens the door to parallel computation. For example, in a multi-agent environment, each agent could optimise over its own variables at the same time. This significantly speeds up computations not only due to the parallelisation, but also due to the fact that each sub-problem has become more computationally efficient. Following Equations 2.12, the dual decomposition method can be summarised by the following iterative scheme [8]:

$$\begin{aligned} x_i^{k+1} &= \arg \min_{x_i} \mathcal{L}(x_i, \lambda^k) \quad \forall i \in \{1, \dots, N\} \\ \lambda^{k+1} &= \lambda^k + \delta^k \left(\sum_{i=1}^N A_i x_i^{k+1} - b \right) \end{aligned} \quad (2.15)$$

Despite the advantage of decentralising the problem, this method is rarely used because it inhibits slow convergence, which can only be proven under many unpractical conditions. One solution to this is to augment the Lagrangian.

2.3.2 The augmented Lagrangian interpretation of ADMM

In [8], ADMM considers the following form of the optimisation problem:

$$\begin{aligned} \min_{x,z} \quad & f(x) + g(z) \\ \text{s.t.} \quad & Ax + Bz = c \end{aligned} \quad (2.16)$$

The objective is assumed to be separable in two functions f and g . The decision vector is also split such that f only depends on its first part x , and g on its second part z . The problem is still coupled through linear equality constraints. This is similar to Problem 2.13, with $N = 2$. It will become clear why this form is assumed and that it is more general than it seems. The Lagrangian of this problem is:

$$\mathcal{L}(x, z, \lambda) = f(x) + g(z) + \lambda^T (Ax + Bz - c) \quad (2.17)$$

However, to improve the convergence properties w.r.t. the previous section, ADMM uses an augmented Lagrangian by adding a quadratic term [8]:

$$\mathcal{L}_\mu(x, z, \lambda) = f(x) + g(z) + \lambda^T (Ax + Bz - c) + \frac{\mu}{2} \|Ax + Bz - c\|^2 \quad (2.18)$$

Here, μ is a positive number called the penalty parameter. Note that the quadratic term in Equation 2.18 prohibits decomposition as in Equation 2.14. To circumvent this, the minimisation of the Lagrangian in the iterative scheme is split up [8]:

$$\begin{aligned} x^{k+1} &= \arg \min_x \mathcal{L}_\mu(x, z^k, \lambda^k) \\ z^{k+1} &= \arg \min_z \mathcal{L}_\mu(x^{k+1}, z, \lambda^k) \\ \lambda^{k+1} &= \lambda^k + \mu(Ax^{k+1} + Bz^{k+1} - c) \end{aligned} \quad (2.19)$$

Note that ADMM uses the penalty parameter μ as the step size for the gradient ascent iterations. Furthermore, the Lagrangian minimisation now consists of two consecutive (but not independent) steps. The minimisation could also be done in one step. This is what the so-called “method of multipliers” does. ADMM, on the other hand, uses Equations 2.19 to not only obtain beneficial convergence properties (by using an augmented Lagrangian), but to simultaneously allow decomposition.

The formal convergence proofs fall outside the scope of this thesis, but it is important to understand what conditions must be met. Convergence cannot be guaranteed without the following two conditions [8]:

- f and g , which can be extended-real-valued functions, must be convex.
- The Lagrangian (Equation 2.17) must have a saddle point.

Contrary to the dual decomposition method, ADMM does not require a differentiable and everywhere finite objective, since it allows f and g to be extended-real-valued functions, which means they may take on any value including $+\infty$. This notably allows for constrained sub-problems in the scheme, since f and g could thereby contain an indicator function of a set, i.e. a function that is zero within the set and $+\infty$ outside the set. This essentially allows to incorporate hard constraints and makes it more suitable for realistic applications.

2.3.3 Distributed consensus ADMM

Consider the following separable problem and remember that the objective could contain constraints through indicator functions [8]:

$$\min_{\chi} f(\chi) = \sum_{i=1}^N f_i(\chi) \quad (2.20)$$

The objective is separable, but each part still depends on the full decision vector $\chi \in \mathbb{R}^n$ here. This could be the case if agents have their own objective but common decision variables, or if they need to take the other agents into account through coupled constraints. To parallelise the computations, the problem is first transformed to an equivalent one [8]:

$$\begin{aligned} \min_{\forall \chi_i, z} \quad & \sum_{i=1}^N f_i(\chi_i) \\ \text{s.t.} \quad & \chi_i - z = 0 \quad \forall i \in \{1, \dots, N\} \end{aligned} \quad (2.21)$$

Where χ_i are local copies of the full decision vector χ this time, such that each agent's objective only depends on this local variable (which is now independent of the other variables). The problem is still coupled through the constraints, where z can now be seen as the global variable or the central collector. Problem 2.21 is a special case of Problem 2.16, where $x = [\chi_1, \dots, \chi_N]^T$, $g(z) = 0$, $A = I_{Nn}$, $B = -[I_n, \dots, I_n]^T$ (N times), and $c = 0$. Also, f is separable and only depends on local variables, meaning that the Lagrangian minimisation can be carried out in parallel. The augmented Lagrangian is now [8]:

$$\mathcal{L}(\chi_i, z, \lambda) = \sum_{i=1}^N f_i(\chi_i) + \lambda_i^T (\chi_i - z) + \frac{\mu}{2} \|\chi_i - z\|^2 \quad (2.22)$$

This leads to the following updates [8]:

$$\begin{aligned}
\chi_i^{k+1} &= \arg \min_{\chi_i} f_i(\chi_i) + \lambda_i^T (\chi_i - z^k) + \frac{\mu}{2} \|\chi_i - z^k\|^2 \quad \forall i \in \{1, \dots, N\} \\
z^{k+1} &= \frac{1}{N} \sum_{i=1}^N (\chi_i^{k+1} + \frac{1}{\mu} \lambda_i^k) \\
\lambda_i^{k+1} &= \lambda_i^k + \mu (\chi_i^{k+1} - z^{k+1}) \quad \forall i \in \{1, \dots, N\}
\end{aligned} \tag{2.23}$$

In the second step, the minimisation of the Lagrangian w.r.t. z is done analytically. Thus, despite the fact that it can be computed in parallel, the first step in this scheme is still by far the most computationally expensive one, especially when these subproblems are heavily constrained or when the number of agents is large (since the local copies contain information about all the agents in this scheme).

2.3.4 Decentralised consensus ADMM

To circumvent the issue mentioned at the end of the previous section, a fully decentralised ADMM scheme is introduced, following [32]. It was used for motion planning of holonomic vehicles in formation, but can also be employed for other applications with pairwise coupled constraints. For clarity about the form of these constraints, they are explicitly written down in the discussion below, instead of assuming that they are contained in the objective (as in Problem 2.20).

The method is applicable to problems of the following form [32]:

$$\begin{aligned}
\min_{\forall x_i} \quad & \sum_{i=1}^N f_i(x_i) \\
\text{s.t.} \quad & x_i \in X_i \\
& g_{i,j}(x_i, x_j) = 0 \quad j \in \mathcal{N}_i \\
& \forall i \in \{1, \dots, N\}
\end{aligned} \tag{2.24}$$

Each agent i has its own objective f_i which depends only on the agents own unique decision variables x_i , hence there is no central collector in decentralised ADMM, as opposed to distributed ADMM. These variables can be constrained separately to a set X_i , e.g. actuation constraints or physical boundaries for the agent's own system and environment. The pairwise coupled constraints between two agents are denoted by $g_{i,j}(x_i, x_j) = 0$. In [32], they constitute formation constraints. They are present for each agent-neighbour pair, where the neighbourhood of agent i is denoted by $\mathcal{N}_i$. This “neighbourhood” is a set of indices of other agents that agent i considers as “neighbours”, in this case, the agents with which it forms a formation. It is clear that Problem 2.24 is a more specific case of Problem 2.20. Here, the only coupling lies in these pairwise constraints. This case allows for the following transformation [32]:

$$\begin{aligned}
 \min_{\forall x_i} \quad & \sum_{i=1}^N f_i(x_i) \\
 \text{s.t.} \quad & x_i \in X_i \\
 & x_i = z_i, \quad x_j = z_{i,j} \\
 & g_{i,j}(z_i, z_{i,j}) = 0 \quad j \in \mathcal{N}_i \\
 & \forall i \in \{1, \dots, N\}
 \end{aligned} \tag{2.25}$$

Each agent introduces the slack variables z_i (a copy of its own decision variables) and $z_{i,j}$ (a copy of its neighbours decision variables). The pairwise constraints can now be written as a function of these slack variables, making them dependent on variables that only belong to agent i . The consensus constraints are now $x_i = z_i$ and $x_j = z_{i,j}$, to make sure the problem is equivalent to the original. Thus, in this case, the z -variables should be interpreted as local copies. Subsequently, the Lagrangian is [32]:

$$\begin{aligned}
 \mathcal{L}_\mu = \sum_{i=1}^N & \left[f_i(x_i) + \lambda_i^T(x_i - z_i) + \frac{\mu}{2} \|x_i - z_i\|_2^2 \right. \\
 & \left. + \sum_{j \in \mathcal{N}_i} \left(\lambda_{i,j}^T(x_j - z_{i,j}) + \frac{\mu}{2} \|x_j - z_{i,j}\|_2^2 \right) \right]
 \end{aligned} \tag{2.26a}$$

$$= \sum_{i=1}^N \mathcal{L}_{\mu,i}(x_i, z_i, \lambda_i, x_j, z_{i,j}, \lambda_{i,j}) \tag{2.26b}$$

$$= \sum_{i=1}^N \mathcal{L}_{\mu,i}(x_i, z_i, \lambda_i, x_i, z_{j,i}, \lambda_{j,i}) \tag{2.26c}$$

Which can be written as a sum of N sub-Lagrangians $L_{\mu,i}$, representing each agent, as shown in Equation 2.26b. An important property of this Lagrangian is that the terms can be reshuffled as in Equation 2.26c, such that each sub-Lagrangian is no longer dependant on its neighbour's x -variables. This is only possible if the following holds [32]:

$$j \in \mathcal{N}_i \iff i \in \mathcal{N}_j \tag{2.27}$$

This is called the bidirectional identity. It implies that if agent j is in the neighbourhood of agent i , it must also consider agent i as one of its neighbours. If this is not the case, the Lagrangian would lack some terms necessary to rewrite it to the desired form. Equations 2.26b and 2.26c can now both be used by ADMM since they are equal. This leads to the following updates [32]:

$$\begin{aligned}
 x_i^{k+1} = \arg \min_{x_i} \quad & \mathcal{L}_{\mu,i}(x_i, z_i^k, \lambda_i^k, x_i, z_{j,i}^k, \lambda_{j,i}^k) \\
 \text{s.t.} \quad & x_i \in X_i
 \end{aligned} \tag{2.28}$$

$$\begin{pmatrix} z_i^{k+1} \\ z_{i,j}^{k+1} \end{pmatrix} = \begin{array}{ll} \arg \min_{z_i, z_{i,j}} & \mathcal{L}_{\mu,i}(x_i^{k+1}, z_i, \lambda_i^k, x_j^{k+1}, z_{i,j}, \lambda_{i,j}^k) \\ \text{s.t.} & g_{i,j}(z_i, z_{i,j}) = 0 \quad \forall j \in \mathcal{N}_i \end{array} \quad (2.29)$$

$$\begin{aligned} \lambda_i^{k+1} &= \lambda_i^k + \mu(x_i^{k+1} - z_i^{k+1}) \\ \lambda_{i,j}^{k+1} &= \lambda_{i,j}^k + \mu(x_j^{k+1} - z_{i,j}^{k+1}) \quad \forall j \in \mathcal{N}_i \end{aligned} \quad (2.30)$$

Where in the x -update 2.28, the reshuffled form of the Lagrangian 2.26c is used to allow decomposition. The z -update and λ -update are decomposable as such. The result is a fully decentralised algorithm, where only the results of each separate update need to be communicated between the agents. More specifically, each agent needs their neighbour's variables x_j for the z -update, so they communicate their results after the x -updates. Similarly, the slack variables $z_{i,j}$ and dual variables $\lambda_{i,j}$ are communicated after the λ -update, to be used in the next x -update by the neighbours (since the reshuffled version of the Lagrangian is used in Equation 2.28).

In contrast to distributed consensus ADMM (Section 2.3.3), during the x -update, the agents only need to optimise over their own decision variables. Furthermore, the constraints are spread over two updates, without ruining the possibility of parallelisation. Consequently, the x -update becomes significantly more tractable, at the cost of the z -update. This last one does have a number of decision variables that scales linearly with the amount of neighbours but has a more tractable objective and only needs to deal with the pairwise constraints.

Lastly, this ADMM version is a special case of distributed consensus ADMM, which is in turn a special case of the general ADMM method. For a more exact discussion of the link between Problems 2.25 and 2.16, see [26]. Furthermore, the use of nondifferentiable, extended real-value functions is allowed by the ADMM method without consequences for the convergence properties. Consequently, the method described in this section inherits the convergence guarantee if Sub-problems 2.28 and 2.29 are convex.

2.4 Control barrier functions

An important concept in industry and engineering in general is safety. For dynamical systems (e.g. machines or robots), this often translates to compliance with certain boundaries or constraints throughout the whole time span of their operation. Mathematically speaking, this is closely related to the invariance of sets which incorporate these constraints. Incorporating control barrier functions (CBF's) can provide a

safety guarantee if the original controller is not inherently safe. As discussed in [3], there exist great similarities with control Lyapunov functions (CLF's), allowing many techniques to be adopted. Where finding and/or designing CLF's can render a dynamical system stable, CBF's can render it safe. This section discusses the basic ideas and conditions of time-invariant CBF's, following [3] and [11]. Then, the generalisation to High-Order CBF's (HOCBF's) is made based on [37]. Finally, we discuss a way to incorporate them in a controller, following the same sources.

The following definitions, all taken from [37], are used:

Definition 1 *A control system is nonlinear affine if the state space can be described in the following form [37]:*

$$\dot{\mathbf{x}} = \boldsymbol{\theta}(\mathbf{x}) + \boldsymbol{\eta}(\mathbf{x})\mathbf{u} \quad (2.31)$$

Definition 2 *“A continuous function $\alpha : [0, a) \rightarrow [0, \infty)$, $a > 0$ belongs to class $\mathcal{K}$ if it is strictly increasing and $\alpha(0) = 0$.” [37]*

Definition 3 *“A set $\mathcal{C}$ is forward invariant for a system if its solutions starting at any $\mathbf{x}(t_0) \in \mathcal{C}$ satisfy $\mathbf{x}(t) \in \mathcal{C}$ for $\forall t > t_0$.” [37] For a controlled system like 2.31, a set can often be made invariant by choosing appropriate control inputs.*

Definition 4 *“The Lie derivative of a function along a vector field captures the change in the value of the function along the vector field.” [37] For example, $L_V h(\mathbf{x})$ denotes the change of the function h along the field V .*

Definition 5 *“The relative degree of a (sufficiently) differentiable function h with respect to system 2.31 is the number of times we need to differentiate it along the dynamics of 2.31 until the control $\mathbf{u}$ explicitly shows.” [37]*

2.4.1 First-order CBFs

This subsection follows [3]. Consider a set described as the superlevel set of a continuously differentiable function $h(\mathbf{x})$:

$$\mathcal{C} = \{\mathbf{x} \in \mathbb{R}^{n_x} : h(\mathbf{x}) \geq 0\} \quad (2.32)$$

The goal is to render this set invariant for system 2.31, i.e. the states may not leave this set by violating the constraint $h(\mathbf{x}) \geq 0$, which could for example represent the distance between a robot and an obstacle. If the relative degree is one, then:

$$\begin{aligned} \dot{h}(\mathbf{x}, \mathbf{u}) &= \nabla h(\mathbf{x}) \cdot \dot{\mathbf{x}} \\ &= \nabla h(\mathbf{x}) \cdot (\boldsymbol{\theta}(\mathbf{x}) + \boldsymbol{\eta}(\mathbf{x})\mathbf{u}) \\ &= L_{\boldsymbol{\theta}}h(\mathbf{x}) + L_{\boldsymbol{\eta}}h(\mathbf{x})\mathbf{u} \end{aligned} \quad (2.33)$$

Where the last equality uses the notation of the Lie derivative, since the change

of the function is captured along the dynamics of the system. Note that the input appears as a linear dependence. Intuitively speaking, $\mathcal{C}$ would be forward invariant if [3]:

$$\exists \mathbf{u} \in U \text{ s.t. } \dot{h}(\mathbf{x}, \mathbf{u}) \geq 0 \quad \forall \mathbf{x} \in \partial \mathcal{C} \quad (2.34)$$

With $U \subset \mathbb{R}_{n_u}$ the feasible set of inputs. Thus, if there exists a feasible control action that ensures the system evolves such that $\dot{h}$ is positive at the boundary of $\mathcal{C}$, the system will never leave the set, rendering $\mathcal{C}$ forward invariant. However, it is quite unpractical to have a condition that only holds at the boundary of a set. Moreover, there is no limit on the rate of decline h in the interior of the set, meaning that the actions do not anticipate the boundary. Therefore, the following condition is used instead [3], where Equation 2.33 is also plugged in:

$$\exists \mathbf{u} \in U \text{ s.t. } L_{\theta}h(\mathbf{x}) + L_{\eta}h(\mathbf{x})\mathbf{u} \geq -\alpha(h(\mathbf{x})) \quad \forall \mathbf{x} \in \mathcal{C} \quad (2.35)$$

With α a class $\mathcal{K}$ function. It is clear that the rate of decline of h is now lower bounded by $-\alpha(h(\mathbf{x}))$ over the whole set. At the boundary ($h(\mathbf{x}) = 0$), this lower bound becomes zero and we recover the original condition ($\dot{h} \geq 0$), since $\alpha(0) = 0$. Finally, the formal condition for h to be a CBF is [3]:

$$\sup_{\mathbf{u} \in U} [L_{\theta}h(\mathbf{x}) + L_{\eta}h(\mathbf{x})\mathbf{u} + \alpha(h(\mathbf{x}))] \geq 0 \quad \forall \mathbf{x} \in \mathcal{C} \quad (2.36)$$

The link with Condition 2.35 is clear, the supremum at the LHS of Condition 2.36 is only larger than zero if there exists a $\mathbf{u}$ that can ensure it. According to this definition, if a class $\mathcal{K}$ function α exists such that $h(\mathbf{x})$ complies with Condition 2.36, $h(\mathbf{x})$ is a control barrier function for the system, and in that case, its superlevel set is forward invariant for that system. Thus, to guarantee safety, this set must be contained in the safe set (i.e., the set described by all constraints on the system needed to maintain safety). However, it is customary to try to make these two sets coincide as much as possible, to avoid the CBF being too restricting for the system.

In summary, one needs to find a function that complies with Condition 2.36 for a given system, and simultaneously make sure that its superlevel set, which will be rendered invariant, is not too restricting w.r.t. the safe set. This is not always straightforward.

2.4.2 High-order CBFs

This subsection follows [37]. When the relative degree r of $h(\mathbf{x})$ is greater than one, its first time derivative does not depend on the inputs. In this case, Condition 2.36 becomes useless and $h(\mathbf{x})$ cannot act as a CBF as such. To generalise this condition, the following functions are defined [37]:

$$\begin{aligned} \psi_i(\mathbf{x}) &= \dot{\psi}_{i-1}(\mathbf{x}) + \alpha_i(\psi_{i-1}(\mathbf{x})) \quad \forall i \in \{1, \dots, r\} \\ \text{with } \psi_0(\mathbf{x}) &= h(\mathbf{x}) \end{aligned} \quad (2.37)$$

Where α_i are class $\mathcal{K}$ functions. Along with these functions, the following sets are defined [37]:

$$\mathcal{C}_i = \{\mathbf{x} \in \mathbb{R}^{n_x} : \psi_{i-1}(\mathbf{x}) \geq 0\} \quad \forall i \in \{1, \dots, r\} \quad (2.38)$$

Where, due to the relative degree of r , only $\psi_r(\mathbf{x}, \mathbf{u})$ depends on the inputs. It is now possible to prove invariance with a recursive argument by imposing the following [37]:

$$\sup_{\mathbf{u} \in U} [\psi_r(\mathbf{x}, \mathbf{u})] \geq 0 \quad \forall \mathbf{x} \in \mathcal{C}_1 \cap \dots \cap \mathcal{C}_r \quad (2.39)$$

Following [37], if $\sup_{\mathbf{u} \in U} [\psi_r(\mathbf{x}, \mathbf{u})] \geq 0$, then $\sup_{\mathbf{u} \in U} [\dot{\psi}_{r-1}(\mathbf{x}) + \alpha_r(\psi_{r-1}(\mathbf{x}))] \geq 0$, which according to Condition 2.36, renders the set $\mathcal{C}_r$ invariant. Consequently, $\psi_{r-1}(\mathbf{x}) \geq 0$ for $t \in [t_0, \infty)$. This implies $\dot{\psi}_{r-2}(\mathbf{x}) + \alpha_{r-1}(\psi_{r-2}(\mathbf{x})) \geq 0$, which renders $\mathcal{C}_{r-1}$ invariant, and so forth. This shows that Condition 2.39 implies that the set $\mathcal{C}_1 \cap \dots \cap \mathcal{C}_r$ is invariant for system 2.31. Using Definitions 2.37 and the Lie derivative notation, Condition 2.39 can be rewritten as follows [37]:

$$\begin{aligned} \sup_{\mathbf{u} \in U} \left[L_{\boldsymbol{\theta}}^r h(\mathbf{x}) + L_{\boldsymbol{\eta}} L_{\boldsymbol{\theta}}^{r-1} h(\mathbf{x}) \mathbf{u} + \sum_{i=1}^{r-1} L_{\boldsymbol{\theta}}^i (\alpha_{r-i}(\psi_{r-i-1}(\mathbf{x}))) + \alpha_r(\psi_{r-1}(\mathbf{x})) \right] \geq 0 \\ \forall \mathbf{x} \in \mathcal{C}_1 \cap \dots \cap \mathcal{C}_r \end{aligned} \quad (2.40)$$

As expected, because the relative degree is r , $L_{\boldsymbol{\eta}}$ only appears at first order and the expression is still linear in $\mathbf{u}$. Thus, if there exist class $\mathcal{K}$ functions α_i such that Condition 2.40 holds, then $h(\mathbf{x})$ is a HOCBF for system 2.31.

2.4.3 Incorporating CBFs in a controller

Condition 2.36 can be used to construct a controller that renders a subset of the safe set invariant, if the CBF is chosen properly. The only problem is that a valid CBF only guarantees that there exists a control action which makes sure this condition is satisfied; it does not say how to find this control action. One way to do this is to construct an optimisation problem where this condition is enforced as a constraint [3]:

$$\begin{aligned} \min_{\mathbf{u}} \quad & \|\mathbf{u} - \mathbf{u}^*\|^2 \\ \text{s.t.} \quad & L_{\boldsymbol{\theta}} h(\mathbf{x}) + L_{\boldsymbol{\eta}} h(\mathbf{x}) \mathbf{u} + \alpha(h(\mathbf{x})) \geq 0 \end{aligned} \quad (2.41)$$

Where $\mathbf{u}^*$ is a control input generated by an unconstrained or unsafe controller.

This input is modified such that this step is minimally invasive, but the system remains in the superlevel set of h , and thus in the safe set. Hence, one could say this optimisation problem “maps” the input to the safe set. An important remark is that $\mathbf{u}$ represents the control vector at one time instant here.

The question might arise as to why not using an optimisation based controller from the start is more satisfactory, since the exact safe set can be enforced in the constraints. Problem 2.41 is a computationally efficient quadratic program (QP), since its objective is quadratic and the constraint linear in $\mathbf{u}$. Subsequently, this strategy offers a fast and inexpensive safety guarantee compared to alternatives. Lastly, the generalisation to include HOCBF constraints is analogous.

Chapter 3

Centralised motion planning

This section discusses centralised optimal control for motion planning. It is the most straightforward way to generate optimal non-colliding trajectories for multiple agents. Here, only one central optimal control problem is formulated, for which the states and inputs of all the agents are decision variables. First, a free-time problem is formulated to strive for time-optimality. Then, to ensure the possibility for replanning and online computation, a model predictive control scheme is explored.

3.1 Free-time problem

Due to its possibility for time-optimality, the free-time problem is usually employed in applications where motion planning is performed offline. However, this thesis aims to perform real-time computations (to further increase the flexibility), so this scheme is only be used as a baseline for time optimality, to compare other algorithms.

3.1.1 Mathematical model

The free-time OCP structure from Section 2.1.4 can be applied specifically to multi-agent motion planning. We formulate it as follows:

$$\min_{\forall \mathbf{x}_i(\tau), \mathbf{u}_i(\tau), T} T \quad (3.1a)$$

$$\text{s.t. } \mathbf{x}_i(0) = \mathbf{x}_{0,i}, \mathbf{x}_i(1) = \mathbf{x}_{f,i} \quad (3.1b)$$

$$\dot{\mathbf{x}}_i(\tau) = T[\mathbf{A}\mathbf{x}_i(\tau) + \mathbf{B}\mathbf{u}_i(\tau)] \quad (3.1c)$$

$$\|\mathbf{v}_i(\tau)\|^2 \leq v_{max}^2 \quad (3.1d)$$

$$\|\mathbf{a}_i(\tau)\|^2 \leq a_{max}^2 \quad (3.1e)$$

$$\|\mathbf{p}_i(\tau) - \mathbf{p}_j(\tau)\|^2 \geq (2R + \epsilon)^2 \quad j \in \mathcal{N}_i \quad (3.1f)$$

$$\forall i \in \{1, \dots, N\}, \tau \in [0, 1]$$

The subscripts i and j refer to the agents and their neighbours. $\mathbf{p}_i$, $\mathbf{v}_i$, and $\mathbf{a}_i$

respectively denote the position, velocity, and acceleration vectors of an agent. These are not extra variables, as they are contained in the states and inputs $\mathbf{x}_i$ and $\mathbf{u}_i$. It is more comprehensible to give these quantities their own symbol, as they appear frequently in the constraints. An agent's state consists of its position and velocity, while its input is taken to be the acceleration. This choice is motivated by the fact that the XPlanar takes these quantities as reference values for its controllers. Thus, in general, recalling that we are in 2D space:

$$\mathbf{x}_i(t) = [p_{x,i}(t), p_{y,i}(t), v_{x,i}(t), v_{y,i}(t)]^T \quad (3.2)$$

$$\mathbf{u}_i(t) = [a_{x,i}(t), a_{y,i}(t)]^T \quad (3.3)$$

Where the x and y subscripts denote the two orthogonal directions. Subsequently, the dynamics are simply a double integrator and can be represented by the following continuous-time state space matrices:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (3.4)$$

The imposed constraints are:

- The initial and terminal conditions: Equations 3.1b determine the starting positions and velocities of each agent, as well as their targets, which they need to reach at the end of the horizon.
- The dynamic constraints: Equations 3.1c impose the trajectories to comply with the state space equations of the system, described by the matrices 3.4. Note that these equality constraints become non-linear due to the extra factor T , rendering them non-convex.
- The velocity and acceleration constraints: Equations 3.1d and 3.1e impose a limit on the magnitude of each agent's velocity and acceleration, since $\|\cdot\|^2$ denotes the Euclidean norm of a vector in this thesis. Every mechatronic system has limits due to its drive system or due to structural or safety reasons. Due to reasons explained in Section 6.2, the limits are taken to be $v_{max} = 1 \text{ m/s}$ and $a_{max} = 5 \text{ m/s}^2$. In some cases, it might be necessary to further decrease these values.
- The collision avoidance constraints: Equations 3.1f ensure safety by imposing that the Euclidean distance between any two agents, which are modelled as circles, remains larger than twice their radius R plus a safety margin ϵ

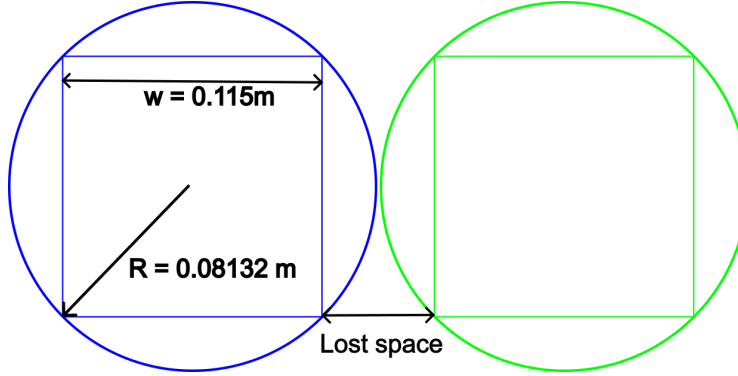


FIGURE 3.1: The square movers of the XPlanar are modelled with circumscribed circles. This induces conservatism as the movers keep a distance between them that is larger than necessary when passing laterally.

at all times. The agents could have any shape, but this approximation is used very often due to its straightforward analytical description and smooth form, which ensures the solver has no difficulties in evaluating the gradient of this constraint. However, it is conservative because it takes away free space that could otherwise have been used by other agents. To prevent any more conservatism, the constraints are not linearised or convexified. The movers from the XPlanar have dimensions $115 \times 115 \text{ mm}$. Approximating this square with a circle gives $R = \frac{0.115\sqrt{2}}{2} = 0.08132 \text{ m}$ (see Fig. 3.1). The selection of ϵ depends on the choice of N_{hor} , as will be discussed below. It is clear that these constraints are also non-convex.

Constraints 3.1b-3.1e must be repeated for each agent, and the collision avoidance constraints for each agent-neighbour pair. Furthermore, they should ideally hold at all times. For now, the neighbourhood of agent i contains all other agents, thus: $\mathcal{N}_i = \{1, \dots, N\} \setminus \{i\}$. This guarantees safety between all agents, but can needlessly drive up the computational cost because not all agents are close to each other all the time. There exist techniques to properly determine the neighbours, which are discussed in Section 4.3.

3.1.2 Implementation guidelines

The free-time strategy has some advantages w.r.t. classical OCP's if they are only solved once offline. The latter do not have the ability to directly minimise transit time because the time horizon is taken to be constant. This also determines the length of the trajectory computed by the OCP. Since the duration of the trajectories from start to finish is not fixed, this becomes impractical. If T_f is guessed too low, the OCP does not compute the motion entirely. An overestimation can lead to unnecessary large computational cost. This also severely reduces the flexibility of

this strategy, since a specific OCP with the right time horizon must be built for every trajectory. The free-time problem does not suffer from this issue as the time horizon itself is a decision variable. This flexibility makes it an attractive candidate for offline motion planning w.r.t. the classical OCP's.

However, the fact that the time horizon is variable also has a downside. If the OCP is solved using the multiple shooting method, one has to choose the number of control grid points N_{hor} . This is important as the method only enforces constraints on the available grid points. This means that if the chosen grid is too coarse, the trajectories might violate constraints between grid points. However, since the time horizon is unknown, it is hard to predict how many grid points are needed to obtain a sufficiently dense grid. To ensure safety, this would force us to take a rather high N_{hor} . To that end, the safety margin ϵ also forms an additional parameter that can be tuned simultaneously with N_{hor} . One could opt to take a very high N_{hor} to obtain a very dense control grid that makes sure the collision avoidance constraints are not violated, but this also greatly increases the computation time. On the other hand, one could opt for a somewhat coarser grid, but then ϵ must be taken sufficiently high such that the agents do not collide between the grid points, which in turn induces conservatism as the agents are forced to stay further away from each other. To find the right balance, extensive simulations were performed.

From experiments, the following heuristic formula was found to work well:

$$N_{hor} = \rho_{N_{hor}} * d \tag{3.5a}$$

$$\text{with } \rho_{N_{hor}} = 15 \tag{3.5b}$$

The idea is that the longest distance between two agents d is taken as a reference. Then, it is multiplied by the desired amount of grid points per meter $\rho_{N_{hor}}$ to obtain N_{hor} (after rounding to the nearest integer). This heuristic allowed us to take $\epsilon = 0.005 \text{ m}$.

3.2 Centalised MPC

A first improvement in the strategy is to introduce replanning. To this end, a model predictive control scheme is well suited. This scheme ensures the freedom to modify certain parameters of the OCP between consecutive time steps to handle dynamic environments, e.g. handling unexpected obstacles or changing targets. Most importantly for this thesis, it opens the door to real-time motion planning. However, it will become apparent that centralised MPC is still very computationally costly, so real-time motion planning is only possible for a limited number of agents.

3.2.1 Mathematical model

In MPC, the idea is to solve the same OCP each time step, i.e. once every Δt seconds. Since the free-time problem has a variable time horizon, this time step (which is given by Equation 2.3) varies from case to case, while MPC requires a constant replanning frequency. Moreover, the free-time problem always calculates the trajectory from start to finish. This can become a problem in continuous operation, which is why the free-time problem is usually only used for offline motion planning. As a consequence, MPC uses a constant time horizon and the classical OCP structure, as in Section 2.1.1:

$$\min_{\forall \mathbf{x}_i(t), \mathbf{u}_i(t)} \int_0^{T_f} \sum_{i=1}^N [(\mathbf{x}_i(t) - \mathbf{x}_{f,i}(t))^T \mathbf{Q}_i (\mathbf{x}_i(t) - \mathbf{x}_{f,i}(t)) + \mathbf{u}_i(t)^T \mathbf{R}_i \mathbf{u}_i(t)] dt \quad (3.6a)$$

$$\text{s.t. } \mathbf{x}_i(0) = \mathbf{x}_{0,i} \quad (3.6b)$$

$$\dot{\mathbf{x}}_i(t) = \mathbf{A}\mathbf{x}_i(t) + \mathbf{B}\mathbf{u}_i(t) \quad (3.6c)$$

$$\|\mathbf{v}_i(t)\|^2 \leq v_{max}^2 \quad (3.6d)$$

$$\|\mathbf{a}_i(t)\|^2 \leq a_{max}^2 \quad (3.6e)$$

$$\|\mathbf{p}_i(t) - \mathbf{p}_j(t)\|^2 \geq (2R + \epsilon)^2 \quad j \in \mathcal{N}_i \quad (3.6f)$$

$$\forall i \in \{1, \dots, N\}, t \in [0, T_f]$$

The OCP now uses normal time t , but the main difference is that agents are driven to their target by penalising their deviation from it. It is not recommended to impose terminal conditions here, as it could be impossible for agents to reach their targets within the imposed time horizon T_f , leading to infeasibilities. The objective function now consists of quadratic penalisations on the deviation of the system's state w.r.t. it's desired final state, integrated over time and summed over all agents. It also includes a term which penalises the input. By tuning the matrices $\mathbf{Q}$ and $\mathbf{R}$, the motion planner becomes more aggressive (large $\mathbf{Q}_i$, the agents reach their target fast) or passive (large $\mathbf{R}_i$, the actuation effort is lowered). These parameters are determined by simulation in the next section. Moreover, since the time horizon and time step are constant, N_{hor} is no longer a parameter. Instead, it is determined through Equation 2.3, which is repeated here for convenience:

$$N_{hor} = \frac{T_f}{\Delta t} \quad (3.7)$$

3.2.2 Implementation guidelines

The MPC strategy is not actually used for the design of a controller, but rather for a motion planner. Its task is to generate collision-free trajectories, more specifically, a set of positions, velocities, and accelerations. These are then fed as reference signals to the actual controllers of the system. Therefore, no physical feedback from measurements is used. Rather, to obtain the new initial condition for the next time

step, the dynamics are simply integrated. There are two reasons for this decision. Firstly, in large-scale systems, the high computational load of solving OCPs causes a large delay. Due to this, the feedback cannot be directly leveraged. Secondly, the Xplanar system is very precise ($\leq 5\mu m$). As long as the external set-points are fed at a sufficiently high rate, which can be done by interpolating the solutions of the OCP, we can assume perfect tracking.

Algorithm 1 summarises the centralised MPC strategy. The inputs are the initial states of each agent $\mathbf{x}_{init,i}$ and the final states (or targets) $\mathbf{x}_{f,i}$. The algorithm is implemented in Python, where Rokit is used to model and solve the OCP. To speed up computations, the OCP is serialised as a CasADi function to avoid much of the overhead that comes along with setting the parameters and sampling the solutions. The multiple shooting method is used, and the transcribed OCP is solved with IPOPT.

In lines 1-5, the algorithm initialises $\mathbf{x}_{0,i}$ and $\mathbf{x}_{c,i}$. These variables respectively represent the initial state for agent i , as seen by the OCP, and the current state of agent i . Note that $\mathbf{x}_{init,i}$ and $\mathbf{x}_{0,i}$ are not the same. $\mathbf{x}_{init,i}$ represents the physical initial condition of an agent, while $\mathbf{x}_{0,i}$ is used for the initial condition constraint of the OCP (Constraint 3.6b), and moves with the horizon as it is updated.

Lines 6-15 constitute the actual MPC loop. During each iteration, OCP 3.6 is solved, using the solution from the previous iteration as an initial guess. The first element of the computed input is given to $\mathbf{u}_{c,i}$, which represents the input in the current time step. Next, the current state is updated by simulating the dynamics for one time step, using the discretised state space matrices (obtained by discretising the continuous-time state space matrices 3.4 with the runge-kutta-4 method). Then, $\mathbf{x}_{0,i}$ is updated with the new current state, such that the OCP can use it as the new initial condition during the next iteration. Finally, the current states and inputs are logged to keep track of the history of the trajectory. This loop continues until every agent is sufficiently close to their targets. All steps have to be repeated for each agent, except for solving the OCP (since it is one centralised problem).

To tune the new parameters, algorithm 1 is extensively simulated. As the overall goal is to get the agents to their destination as quickly as possible, the OCP should be tuned quite aggressively. However, the inputs still need to be slightly penalised to avoid the agents oscillating around their targets. It sufficed to take $\mathbf{Q}_i = I_{n_x \times n_x}$ and $\mathbf{R}_i = 10^{-4} I_{n_u \times n_u}$, where $n_x = 4$ and $n_u = 2$ denote the number of states and inputs per agent. When an application has other priorities for the trajectories, such as smoothness, or if it needs a lower the actuation effort, $\mathbf{R}_i$ could be increased.

As discussed in Section 2.2, the time horizon T_f is also an important parameter. Here, the trade-off is as follows: A large time horizon yields more optimal results because the OCP looks farther into the future, but from Equation 3.7 it is clear that this also increases N_{hor} , which in its turn increases the problem size and computation time. Tuning for the system at hand, we find the following trade-off: $T_f = 1$ s. When

Algorithm 1 Centralised MPC

Inputs: $\mathbf{x}_{init,i}, \mathbf{x}_{f,i}$ **Parameters:** $\mathbf{Q}_i, \mathbf{R}_i, T_f, \epsilon$ **Outputs:** $\mathbf{x}_i(t), \mathbf{u}_i(t)$

```

1: /* Initialise variables */
2: for all agents  $i \in \{1, \dots, N\}$  do
3:    $\mathbf{x}_{0,i} \leftarrow \mathbf{x}_{init,i}$ 
4:    $\mathbf{x}_{c,i} \leftarrow \mathbf{x}_{init,i}$ 
5: end for

6: /* Start MPC loop */
7: while  $\sum_{i=1}^N \|\mathbf{x}_{c,i} - \mathbf{x}_{f,i}\|^2 \geq 10^{-6}$  do
8:    $\forall \mathbf{x}_i(t), \mathbf{u}_i(t) \leftarrow \text{argmin OCP } \textcolor{red}{3.6}$ 
9:   for all agents  $i \in \{1, \dots, N\}$  do
10:     $\mathbf{u}_{c,i} \leftarrow \mathbf{u}_i(0)$ 
11:     $\mathbf{x}_{c,i} \leftarrow \mathbf{A}_d \mathbf{x}_{c,i} + \mathbf{B}_d \mathbf{u}_{c,i}$ 
12:     $\mathbf{x}_{0,i} \leftarrow \mathbf{x}_{c,i}$ 
13:    Log  $\mathbf{x}_{c,i}$  and  $\mathbf{u}_{c,i}$ 
14:   end for
15: end while

```

taking $\Delta t = 0.1$ s, for reasons explained in Section [5.2.3](#), the number of control grid points becomes $N_{hor} = 10$.

As the resulting control grid is quite coarse, the safety margin had to be increased to $\epsilon = 0.03$ m in this case. Additionally, Rockit allows the user to construct an integrator grid (used to integrate the dynamics) that encompasses more grid points than the control grid. The collision avoidance constraints can then also be enforced of the former grid. To guarantee safety between the grid points, this thesis uses that trick for centralised MPC, where the integrator grid is taken to be three times denser than the control grid.

It is already known from literature (Section [1.2](#)) that centralised MPC does not scale well with the number of agents. Section [6.1.3](#) presents a more detailed scalability analysis for the specific motion planning problem of this thesis, once all algorithms have been tested. As it turns out, centralised MPC scales as $\sim O(N^{2.96})$. Thus, it would not suffice to simply use a more powerful computer or to translate the code to C++. It is the scaling itself that must be tackled first, which is the aim of the next chapter.

Chapter 4

Decentralised and hybrid methods

This chapter first introduces a motion planning method based on decentralised ADMM in order to improve the scalability w.r.t. centralised MPC. Then, an additional optimization problem is added to render the trajectories safe using CBFs. This step is performed in a centralised manner, resulting in a hybrid method. Lastly, a neighbour selection scheme is explored to see if the scalability can be further improved.

4.1 ADMM

This section first discusses how the centralised algorithm from the previous section can be mathematically converted to a decentralised algorithm using ADMM. Then, the pseudocode needed for the implementation is discussed. Finally, the algorithm is tuned through simulation.

4.1.1 Mathematical model

As the centralised MPC method (OCP 3.6) has a separable cost, it can be transformed using decentralised ADMM in a similar fashion as Problem 2.25. This results in the following OCP:

$$\min_{\forall \mathbf{x}_i(t), \mathbf{u}_i(t)} \int_0^{T_f} \sum_{i=1}^N J_i(\mathbf{x}_i(t), \mathbf{u}_i(t)) dt \quad (4.1a)$$

$$\text{s.t. } \mathbf{x}_i(0) = \mathbf{x}_{0,i} \quad (4.1b)$$

$$\dot{\mathbf{x}}_i(t) = \mathbf{A}\mathbf{x}_i(t) + \mathbf{B}\mathbf{u}_i(t) \quad (4.1c)$$

$$\|\mathbf{v}_i(t)\|^2 \leq v_{max}^2 \quad (4.1d)$$

$$\|\mathbf{a}_i(t)\|^2 \leq a_{max}^2 \quad (4.1e)$$

$$\mathbf{p}_i(t) = \mathbf{z}_i(t), \mathbf{p}_j(t) = \mathbf{z}_{i,j}(t) \quad j \in \mathcal{N}_i \quad (4.1f)$$

$$\|\mathbf{z}_i(t) - \mathbf{z}_{i,j}(t)\|^2 \geq (2R + \epsilon)^2 \quad j \in \mathcal{N}_i \quad (4.1g)$$

$$\forall i \in \{1, \dots, N\}, t \in [0, T_f]$$

Where $J_i(\mathbf{x}_i(t), \mathbf{u}_i(t)) = (\mathbf{x}_i(t) - \mathbf{x}_{f,i}(t))^T \mathbf{Q}_i (\mathbf{x}_i(t) - \mathbf{x}_{f,i}(t)) + \mathbf{u}_i(t)^T \mathbf{R}_i \mathbf{u}_i(t)$. This is the individual cost function that drives the agent to its target position, which also appears in Equation 3.6a. Unlike Problem 2.25, only a subset of the decision variables, namely the positions $\mathbf{p}_i(t)$, is coupled here. Thus, to transform the centralised problem, each agent introduces the slack variables $\mathbf{z}_i(t)$ and $\mathbf{z}_{i,j}(t)$ only for the positions of itself and its neighbours, along with the corresponding consensus constraints 4.1f. This ensures that the collision avoidance constraints 4.1g can be written as a function of $\mathbf{z}_i(t)$ and $\mathbf{z}_{i,j}(t)$ instead of $\mathbf{p}_i(t)$ and $\mathbf{p}_j(t)$. Subsequently, these slack variables can be respectively interpreted as local copies of agent i 's own position and its neighbours' positions.

With this transformation, OCP 4.1 can be decentralised using ADMM. When only dualising the consensus constraints, the augmented Lagrangian becomes:

$$\begin{aligned} \mathcal{L}_\mu = & \int_0^{T_f} \sum_{i=1}^N \left[J_i(\mathbf{x}_i(t), \mathbf{u}_i(t)) + \boldsymbol{\lambda}_i(t)^T (\mathbf{p}_i(t) - \mathbf{z}_i(t)) + \frac{\mu}{2} \|\mathbf{p}_i(t) - \mathbf{z}_i(t)\|_2^2 \right] dt \\ & + \int_0^{T_f} \sum_{i=1}^N \sum_{j \in \mathcal{N}_i} \left[\boldsymbol{\lambda}_{i,j}(t)^T (\mathbf{p}_j(t) - \mathbf{z}_{i,j}(t)) + \frac{\mu}{2} \|\mathbf{p}_j(t) - \mathbf{z}_{i,j}(t)\|_2^2 \right] dt \end{aligned} \quad (4.2a)$$

$$= \sum_{i=1}^N \mathcal{L}_{\mu,i}(\mathbf{x}_i(t), \mathbf{u}_i(t), \mathbf{p}_i(t), \mathbf{z}_i(t), \boldsymbol{\lambda}_i(t), \mathbf{p}_j(t), \mathbf{z}_{i,j}(t), \boldsymbol{\lambda}_{i,j}(t)) \quad (4.2b)$$

$$= \sum_{i=1}^N \mathcal{L}_{\mu,i}(\mathbf{x}_i(t), \mathbf{u}_i(t), \mathbf{p}_i(t), \mathbf{z}_i(t), \boldsymbol{\lambda}_i(t), \mathbf{p}_i(t), \mathbf{z}_{j,i}(t), \boldsymbol{\lambda}_{j,i}(t)) \quad (4.2c)$$

Where the other constraints are implicitly contained in the objective. Since we are decentralising an OCP here, the dual variables $\boldsymbol{\lambda}_i(t), \boldsymbol{\lambda}_{i,j}(t)$ are a function of

time and the whole expression is integrated over the horizon. This is due to the fact that the consensus constraints 4.1f set the positions and their local copies, which are both a function of time, equal to each other. Subsequently, the dual variables become continuous, resulting in an infinite number of constraints. Before solving, the sub-problems are discretised with the multiple shooting method, taking care of this issue. Using this Lagrangian, we find the following ADMM updates in a similar fashion to Equations 2.28 - 2.30:

$$\begin{aligned} \begin{pmatrix} \mathbf{x}_i^{k+1}(t) \\ \mathbf{u}_i^{k+1}(t) \end{pmatrix} = \arg \min_{\mathbf{x}_i(t), \mathbf{u}_i(t)} & L_{\mu,i}(\mathbf{x}_i(t), \mathbf{u}_i(t), \mathbf{z}_i^k(t), \boldsymbol{\lambda}_i^k(t), \mathbf{p}_i(t), \mathbf{z}_{j,i}^k(t), \boldsymbol{\lambda}_{j,i}^k(t)) \\ \text{s.t.} & \mathbf{x}_i(0) = \mathbf{x}_{0,i} \\ & \dot{\mathbf{x}}_i(t) = \mathbf{A}\mathbf{x}_i(t) + \mathbf{B}\mathbf{u}_i(t) \\ & \|\mathbf{v}_i(t)\|^2 \leq v_{max}^2 \\ & \|\mathbf{a}_i(t)\|^2 \leq a_{max}^2 \end{aligned} \quad (4.3)$$

$$\begin{aligned} \begin{pmatrix} \mathbf{z}_i^{k+1}(t) \\ \mathbf{z}_{i,j}^{k+1}(t) \end{pmatrix} = \arg \min_{\mathbf{z}_i(t), \mathbf{z}_{i,j}(t)} & L_{\mu,i}(\mathbf{p}_i^{k+1}(t), \mathbf{z}_i(t), \boldsymbol{\lambda}_i^k(t), \mathbf{p}_j^{k+1}(t), \mathbf{z}_{i,j}(t), \boldsymbol{\lambda}_{i,j}^k(t)) \\ \text{s.t.} & \|\mathbf{z}_i(t) - \mathbf{z}_{i,j}(t)\|^2 \geq (2R + \epsilon)^2 \quad j \in \mathcal{N}_i \end{aligned} \quad (4.4)$$

$$\begin{aligned} \boldsymbol{\lambda}_i^{k+1}(t) &= \boldsymbol{\lambda}_i^k(t) + \mu(\mathbf{p}_i^{k+1}(t) - \mathbf{z}_i^{k+1}(t)) \\ \boldsymbol{\lambda}_{i,j}^{k+1}(t) &= \boldsymbol{\lambda}_{i,j}^k(t) + \mu(\mathbf{p}_j^{k+1}(t) - \mathbf{z}_{i,j}^{k+1}(t)) \quad \forall j \in \mathcal{N}_i \end{aligned} \quad (4.5)$$

Here, the updates 4.3-4.5 can be given the following intuitive explanation, illustrated in Fig. 4.1. First, each agent solves OCP 4.3 to plan its own trajectory to the target position while complying with its own dynamic constraints and actuation limits. After this step is solved, the position $\mathbf{p}(t)$ of each agent is communicated with all its neighbours.

Next, each agent replans by finding its local copies $\mathbf{z}_i(t)$ and $\mathbf{z}_{i,j}(t)$ following Equation 4.4. Since $\mathbf{z}_{i,j}(t)$ belongs to agent i , it can be seen as a “suggestion” from it to its neighbour to try and make their trajectories match the objectives and safety of the other. Similarly, agent j has a slack variable $\mathbf{z}_{j,i}(t)$ that serves as its suggestion to agent i . Only the collision avoidance constraints are active in this subproblem. Hence, each agent replans its own trajectory, along with its suggestions for its neighbours, such that the latter do not collide with its own trajectory. Since the $\mathbf{z}$ -update does not involve dynamics, it is not an OCP, but rather a non-linear program (NLP). As a result, the replanned path may be physically infeasible or dynamically inconsistent. Lastly, the dual variables are updated with a gradient ascent step.

In the next iteration, the agents again plan their own trajectory, taking into account the suggestions that its neighbours have calculated in the previous iteration

during the $\mathbf{z}$ -update. This is done through the consensus constraints, which are embedded in the dual and augmented terms of the Lagrangian. The agent tries to deviate as little as possible from the path it wants to take itself $\mathbf{z}_i(t)$ and the path that the neighbours want it to take $\mathbf{z}_{j,i}(t)$. Then, the agents replan again and obtain new suggestions. This iterative method should in principle continue until they come to a consensus, i.e. they all obtain physical, non-colliding trajectories.

The method converges when all the consensus residuals $\mathbf{p}_i - \mathbf{z}_i$ and $\mathbf{p}_j - \mathbf{z}_{i,j}$ approach zero, as these constitute the gradient of the dual function of OCP 4.1 (see Section 2.3). In practice, to compare the degree of convergence between different implementations or tunings, we use the Average Consensus Residual (ACR) as a figure of merit. Here, the consensus residual is averaged over every agent, its neighbours, and the control horizon:

$$ACR = \frac{1}{2N(M+1)N_{hor}} \sum_{i=1}^N \left[\sum_{k=0}^{N_{hor}-1} \|\mathbf{p}_i[k] - \mathbf{z}_i[k]\|^2 \right]^{\frac{1}{2}} \quad (4.6a)$$

$$+ \frac{1}{2N(M+1)N_{hor}} \sum_{i=1}^N \sum_{j \in \mathcal{N}_i} \left[\sum_{k=0}^{N_{hor}-1} \|\mathbf{p}_j[k] - \mathbf{z}_{i,j}[k]\|^2 \right]^{\frac{1}{2}} \quad (4.6b)$$

Where N denotes the number of agents, M the number of neighbours, and N_{hor} the number of control grid points after transcription with the multiple shooting method.

Lastly, note that NLP 4.4 contains variables for the agent and all its neighbours, unlike OCP 4.3, which only optimises over the agent's own variables. Moreover, the former entails the non-convex collision avoidance constraints, which are notoriously hard to deal with. Consequently, NLP 4.4 is the bottleneck w.r.t. computation time and scalability. This is also formally discussed in Section 5.3.1.

Thus, to qualitatively compare the scaling of the centralised and decentralised problem, we must compare the centralised OCP 3.6 with NLP 4.4 (the bottleneck of the decentralised problem). Apart from the velocity, acceleration, and dynamic constraints which do not appear in the NLP, centralised MPC enforces collision avoidance constraints between every agent-neighbour pair in the same OCP, leading to a quadratic dependence; the number of collision avoidance constraints in centralised MPC scales with $\sim O(N^2)$, where N is the number of agents. The NLP only enforces the constraints between every neighbour and its own agent, thus the number of collision avoidance constraints in the NLP scales with $\sim O(N)$. Therefore, the NLP is significantly more tractable than the centralised OCP. From the formal scalability comparison in Section 6.1.3, it turns out that the decentralisation reduces the scalability with more than an order w.r.t. centralised MPC.

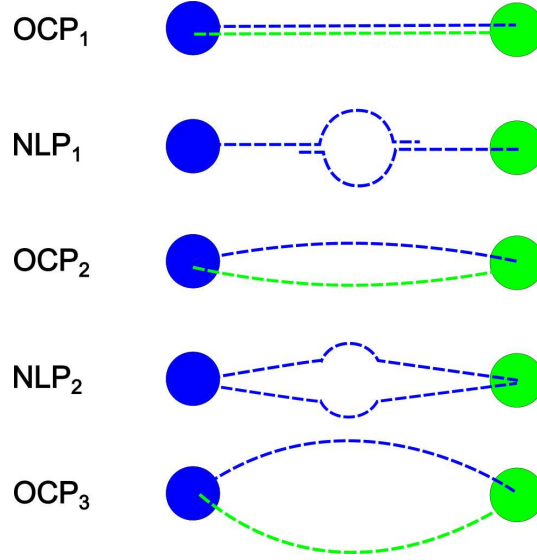


FIGURE 4.1: Qualitative illustration of the planned trajectories over the ADMM iterations, applied to a case where two agents need to switch positions. First, the agents plan the most optimal trajectory, independent of each other and communicate their findings (OCP_1). Then, they each re-plan their own and each other's paths, purely such that they do not collide (NLP_1). Once again, they communicate their findings. As these paths might be dynamically infeasible, the agents again plan their own trajectory (OCP_2), independent of each other, but deviating as little as possible from these non-colliding paths of NLP_1 . As these new trajectories may still collide, the above process is repeated (NLP_2 , OCP_3 , ...).

4.1.2 Implementation guidelines

Algorithm 2 shows how the decentralised motion planning algorithm is implemented. In this implementation, an agent's neighbourhood contains all other agents. Later, Section 4.3 explores methods that select fewer neighbours in order to reduce the computational load of NLP 4.4. As in Chapter 3, the OCPs are constructed and serialised with Rokit and solved with IPOPT.

Similar to Algorithm 1, the input includes the starting position and target position of each agent. On the other hand, Algorithm 2 introduces additional parameters m_{pre} , m and μ . They respectively represent the number of ADMM pre-iterations, the number of ADMM iterations and the penalty parameter μ in the augmented Lagrangian (Equation 2.26a). Following [32], the number of iterations is fixed at the start, with the idea that the method converges over the time steps, instead of trying to solve the problem to full convergence every time step.

The initialisation, lines 1-7, now also consists of setting $\mathbf{z}_i$, $\mathbf{z}_{i,j}$, $\boldsymbol{\lambda}_i$ and $\boldsymbol{\lambda}_{i,j}$ to zero. To bring these values to a better initial guess, m_{pre} pre-iterations are performed in the first time step, before the agents start their actual trajectory.

Algorithm 2 Decentralised ADMM

Inputs: $\mathbf{x}_{init,i}, \mathbf{x}_{f,i}$ **Parameters:** m_{pre}, m, μ **Outputs:** $\mathbf{x}_i(t), \mathbf{u}_i(t)$

```
1: /* Initialise the agents */
2: for all agents  $i \in \{1, \dots, N\}$  do
3:    $\mathbf{x}_{0,i} \leftarrow \mathbf{x}_{init,i}$ 
4:    $\mathbf{x}_{c,i} \leftarrow \mathbf{x}_{init,i}$ 
5:    $\mathbf{z}_i(t), \mathbf{z}_{i,j}(t) \leftarrow \mathbf{0}$ 
6:    $\boldsymbol{\lambda}_i(t), \boldsymbol{\lambda}_{i,j}(t) \leftarrow \mathbf{0}$ 
7: end for

8: /* Do pre-iterations */
9: for all ADMM pre-iterations  $m_{pre}$  do
10:  for all agents  $i \in \{1, \dots, N\}$  do
11:     $\mathbf{x}_i(t), \mathbf{u}_i(t)$  according to OCP 4.3
12:    Update  $\mathbf{p}_i(t)$  in shared memory
13:  end for
14:  for all agents  $i \in \{1, \dots, N\}$  do
15:     $\mathbf{z}_i(t), \mathbf{z}_{i,j}(t)$  according to NLP 4.4
16:     $\boldsymbol{\lambda}_i(t), \boldsymbol{\lambda}_{i,j}(t)$  according to Equations 4.5
17:    Update  $\mathbf{z}_i(t), \mathbf{z}_{i,j}(t), \boldsymbol{\lambda}_i(t)$  and  $\boldsymbol{\lambda}_{i,j}(t)$  in shared memory
18:  end for
19: end for

20: /* Do ADMM loop */
21: while  $\sum_{i=1}^N \|\mathbf{x}_{c,i} - \mathbf{x}_{f,i}\|^2 \geq 10^{-6}$  do
22:  for all ADMM iterations  $m$  do
23:    for all agents  $i \in \{1, \dots, N\}$  do
24:       $\mathbf{x}_i(t), \mathbf{u}_i(t)$  according to OCP 4.3
25:      Update  $\mathbf{p}_i(t)$  in shared memory
26:    end for
27:    for all agents  $i \in \{1, \dots, N\}$  do
28:       $\mathbf{z}_i(t), \mathbf{z}_{i,j}(t)$  according to NLP 4.4
29:       $\boldsymbol{\lambda}_i(t), \boldsymbol{\lambda}_{i,j}(t)$  according to Equations 4.5
30:      Update  $\mathbf{z}_i(t), \mathbf{z}_{i,j}(t), \boldsymbol{\lambda}_i(t)$  and  $\boldsymbol{\lambda}_{i,j}(t)$  in shared memory
31:    end for
32:  end for
33:  for all agents  $i \in \{1, \dots, N\}$  do
34:     $\mathbf{u}_{c,i} \leftarrow \mathbf{u}_i(0)$ 
35:     $\mathbf{x}_{c,i} \leftarrow \mathbf{A}_d \mathbf{x}_{c,i} + \mathbf{B}_d \mathbf{u}_{c,i}$ 
36:     $\mathbf{x}_{0,i} \leftarrow \mathbf{x}_{c,i}$ 
37:    Log  $\mathbf{u}_{c,i}$  and  $\mathbf{x}_{c,i}$ 
38:  end for
39: end while
```

An ADMM iteration consists of solving OCP 4.3 for all agents and updating $\mathbf{p}_i$ in shared memory. Only when this step is completed for all agents can the algorithm proceed to solve NLP 4.4 and Equation 4.5 and store the solutions in shared memory. During these pre-iterations, the agents plan their initial trajectories while remaining stationary as there is no step that simulates the system. The use of shared memory instead of direct communication originates from the fact that the system is assumed to run on a central computer, as is the case for the XPlanar.

The actual simulation loop, lines 20-38, consists of m ADMM iterations, after which the last $\mathbf{u}_i$ and $\mathbf{x}_i$ are used to simulate the system, before being logged. Note that the use of for-loops over all agents implies that this algorithm still performs ADMM in a sequential manner. Hence, this algorithm does not yet take advantage of the fact that decentralised ADMM can be parallelised. In Section 5.3, the code is multithreaded to take advantage of the ADMM structure.

From experiments, it follows that μ should be taken large enough for the agents to obtain consensus. However, as ADMM is a method that only inhibits a convergence guarantee for convex problems, it quickly becomes apparent that this method is unsafe for OCP 4.1 due to its non-convex collision avoidance constraints. Even when taking $\mu = 70$, while performing $m = 50$ iterations per time step with $m_{pre} = 200$ pre-iterations, computed trajectories can still produce collisions, as illustrated in Fig. 4.2. Apart from the lack of safety, it is undesirable to perform this many iterations per time step since this also drives up the computation time.

4.2 ADMM-CBF

Since iterating each time step to convergence is not always possible or may be computationally inefficient, the safety of ADMM cannot be guaranteed. Moreover, since NLP 4.4 is non-convex, there is no convergence guarantee (Section 2.3.2). Therefore, we design a HOCBF to map the inputs to the safe set, ensuring the overall safety of the trajectories. The algorithm is then extended to integrate the safety step. As Section 5 implements the algorithm on a system with a central computer, the mapping to the safe set is performed in a centralised manner. This results in a hybrid method, combining decentralised planning with a central safety step. Finally, the influence of the HOCBF parameters is discussed, along with the new user guideline for the algorithm.

4.2.1 Mathematical model

During each time step, the input of the system can be mapped to the safe set using the strategy from Section 2.4.3. Since the system dynamics consist of a double integrator, its relative degree is 2. This shows the need for a HOCBF of order 2. The HOCBF is chosen to be exactly the collision avoidance constraint, rewritten in the standard form (as in Equation 2.32):

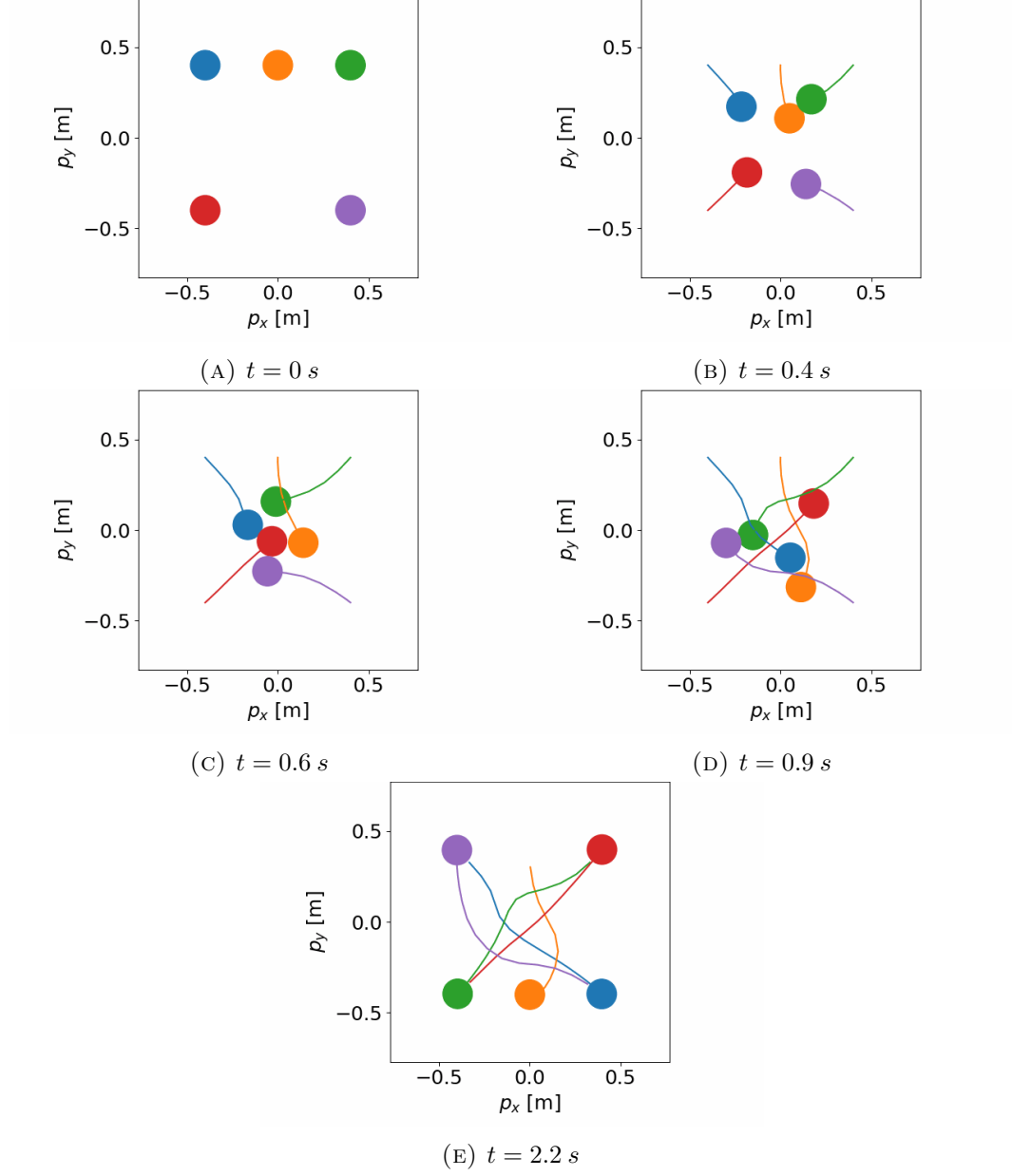


FIGURE 4.2: The 2D trajectory of all 5 agents, resulting from decentralised ADMM. Even when performing $m = 50$ ADMM iterations per time step, the method has not fully converged yet. Hence, the resulting trajectories yield collisions, e.g. at $t = 0.4\text{ s}$ and $t = 0.9\text{ s}$.

$$h_{i,j} = \|\mathbf{p}_i(0) - \mathbf{p}_j(0)\|^2 - (2R + \epsilon)^2 \quad (4.7)$$

To find the condition that $h_{i,j}$ must satisfy to be a HOCBF, we construct the following functions, similar to Equations 2.37:

$$\psi_1 = \dot{\psi}_0 + \alpha_1(h_{i,j}) \quad (4.8a)$$

$$\psi_2 = \dot{\psi}_1 + \alpha_2(\psi_1) \quad (4.8b)$$

Following Equation 2.39, the condition for forward invariance is $\psi_2 \geq 0$. Plugging Equation 4.8a in Equation 4.8b, and choosing linear class $\mathcal{K}$ functions [37] ($\alpha_i(h) = K_i h$), the condition for $h_{i,j}$ becomes:

$$\text{cbf}_{i,j} = \ddot{h}_{i,j} + (K_1 + K_2)\dot{h}_{i,j} + K_1 K_2 h_{i,j} \geq 0 \quad (4.9)$$

The constraint should be repeated for all possible combinations of agent pairs, resulting in a number of constraints that scales as $\sim O(N^2)$, with N the number of agents. The time derivatives of $h_{i,j}$ are found by differentiating over the dynamics:

$$\begin{aligned} \dot{h}_{i,j} &= 2(\mathbf{p}_i(0) - \mathbf{p}_j(0)) \cdot (\mathbf{v}_i(0) - \mathbf{v}_j(0)) \\ \ddot{h}_{i,j} &= 2\|\mathbf{v}_i(0) - \mathbf{v}_j(0)\|^2 + 2(\mathbf{p}_i(0) - \mathbf{p}_j(0)) \cdot (\mathbf{a}_i(0) - \mathbf{a}_j(0)) \end{aligned} \quad (4.10)$$

Note that indeed the constraints are linearly dependent on the acceleration input. The position and velocity variables of Equation 4.10 are already known from the initial condition of the OCP. This HOCBF can be formulated as a quadratic program (QP):

$$\begin{aligned} \min_{\forall \mathbf{u}_i} \quad & \sum_{i=1}^N \|\mathbf{u}_i(0) - \mathbf{u}_i^*(0)\|^2 \\ \text{s.t.} \quad & \text{cbf}_{i,j} \geq 0, \quad \forall j \neq i \\ & \|\mathbf{u}_i(0)\|^2 \leq a_{max}^2 \\ & \forall i \in \{1, \dots, N\} \end{aligned} \quad (4.11)$$

Where $\mathbf{u}_i^*$ is the control input obtained by ADMM. The constraints from Equation 4.1e are also added to make sure that after rendering the trajectories safe, the acceleration is still within the allowed range. Since this involves a quadratic function, Problem 4.11, although still very computationally efficient, actually becomes a quadratically constrained quadratic program (QCQP). Note that this QCQP does not involve a time horizon, as it does not plan or anticipate future steps. It only maps the current input to the safe set in every time step, which is why the variables are consistently evaluated at $t = 0$, to establish the link with the ADMM sub-problems.

However, unlike the classical collision avoidance constraints, it takes into account the acceleration and velocity of the agent pairs. This extra information ensures more cautious collision avoidance manoeuvres.

4.2.2 Implementation guidelines

QCQP 4.11 is added to Algorithm 2 after the ADMM iterations to obtain a hybrid method. While decentralised CBFs can also be safe under certain conditions, centralised CBFs are more efficient and less conservative, according to [7]. Moreover, the fact that the QCQP is very computationally efficient further motivates this choice. Moreover, the Beckhoff Xplanar, which employs a central computer controlling all agents, allows for this implementation. The new hybrid method can be found in Algorithm 3, where at line 34 the input is mapped to the safe set in a centralised manner. This QCQP is constructed and serialised with the CasADi-Opti stack. To exploit the structure of this specific problem, qpOASES [13] is used as a solver.

The formulation of QCQP 4.11 also introduces the additional parameters K_1 and K_2 . Looking back at the discussion in Section 2.4.1, it becomes apparent that they are related to the maximum negative value allowed of $\dot{h}$ and $\ddot{h}$. The larger K_1 and K_2 , the less constrained $\dot{h}$ and $\ddot{h}$ are. This means that the distance between the agents and the velocity with which they approach each other is allowed to decline faster (as long as they are not too close to each other), leading to more aggressive behaviour. When K_1 and K_2 are taken to be smaller, these rates of decline become stricter, leading to more passive behaviour, i.e. the agents stay away further from each other and avoid approaching each other at high velocities. This behaviour is shown for two extreme cases in Fig. 4.3. The parameters are tuned for the system at hand, resulting in $K_1 = 8$ and $K_2 = 7$. The CBFs also give the possibility to reduce the safety margin ϵ to 0.005 m , as opposed to centralised MPC, where it had to be taken quite high (0.03 m).

Now that a safety guarantee has been established, it is possible to lower the number of ADMM iterations per time step to reduce the computation time. There is also a trade-off when offloading most collision avoidance to QCQP 4.11. This comes at the cost of optimality since the QCQP does not actually plan ahead. There exist middle grounds depending on the desired behaviour, but to clearly illustrate the difference, this section investigates two extremes.

Performance mode: $\mu = 40$, $m = 20$, $m_{pre} = 200$.

In this mode, we exploit the safety guarantee, allowing us to decrease m , although it still performs many iterations to make sure the trajectories are coordinated and properly planned. It also uses a high m_{pre} to ensure the agents come to a consensus at the start. Subsequently, this mode usually entails a lower transit time and more

Algorithm 3 Hybrid method: ADMM with CBF**Inputs:** $\mathbf{x}_{init,i}, \mathbf{x}_{f,i}$ **Parameters:** $m_{pre}, m, \mu, K_1, K_2$ **Outputs:** $\mathbf{x}_i(t), \mathbf{u}_i(t)$

```

1: /* Initialise the agents */
2: for all agents  $i \in \{1, \dots, N\}$  do
3:    $\mathbf{x}_{0,i} \leftarrow \mathbf{x}_{init,i}$ 
4:    $\mathbf{x}_{c,i} \leftarrow \mathbf{x}_{init,i}$ 
5:    $\mathbf{z}_i(t), \mathbf{z}_{i,j}(t) \leftarrow \mathbf{0}$ 
6:    $\boldsymbol{\lambda}_i(t), \boldsymbol{\lambda}_{i,j}(t) \leftarrow \mathbf{0}$ 
7: end for

8: /* Do pre-iterations */
9: for all ADMM pre-iterations  $m_{pre}$  do
10:  for all agents  $i \in \{1, \dots, N\}$  do
11:     $\mathbf{x}_i(t), \mathbf{u}_i(t)$  according to OCP 4.3
12:    Update  $\mathbf{p}_i(t)$  in shared memory
13:  end for
14:  for all agents  $i \in \{1, \dots, N\}$  do
15:     $\mathbf{z}_i(t), \mathbf{z}_{i,j}(t)$  according to NLP 4.4
16:     $\boldsymbol{\lambda}_i(t), \boldsymbol{\lambda}_{i,j}(t)$  according to Equations 4.5
17:    Update  $\mathbf{z}_i(t), \mathbf{z}_{i,j}(t), \boldsymbol{\lambda}_i(t)$  and  $\boldsymbol{\lambda}_{i,j}(t)$  in shared memory
18:  end for
19: end for

20: /* Do ADMM loop */
21: while  $\sum_{i=1}^N \|\mathbf{x}_{current,i} - \mathbf{x}_{f,i}\|^2 \geq 10^{-6}$  do
22:  for all ADMM iterations  $m$  do
23:    for all agents  $i \in \{1, \dots, N\}$  do
24:       $\mathbf{x}_i(t), \mathbf{u}_i(t)$  according to OCP 4.3
25:      Update  $\mathbf{p}_i(t)$  in shared memory
26:    end for
27:    for all agents  $i \in \{1, \dots, N\}$  do
28:       $\mathbf{z}_i(t), \mathbf{z}_{i,j}(t)$  according to NLP 4.4
29:       $\boldsymbol{\lambda}_i(t), \boldsymbol{\lambda}_{i,j}(t)$  according to Equations 4.5
30:      Update  $\mathbf{z}_i(t), \mathbf{z}_{i,j}(t), \boldsymbol{\lambda}_i(t)$  and  $\boldsymbol{\lambda}_{i,j}(t)$  in shared memory
31:    end for
32:  end for

33: /* Render input safe */
34:  $\mathbf{u}_1(0), \dots, \mathbf{u}_N(0)$  according to QCQP 4.11
35: for all agents  $i \in \{1, \dots, N\}$  do
36:    $\mathbf{u}_{c,i} \leftarrow \mathbf{u}_i(0)$ 
37:    $\mathbf{x}_{c,i} \leftarrow \mathbf{A}_d \mathbf{x}_{c,i} + \mathbf{B}_d \mathbf{u}_{c,i}$ 
38:    $\mathbf{x}_{0,i} \leftarrow \mathbf{x}_{c,i}$ 
39:   Log  $\mathbf{u}_{c,i}$  and  $\mathbf{x}_{c,i}$ 
40: end for

41: end while

```

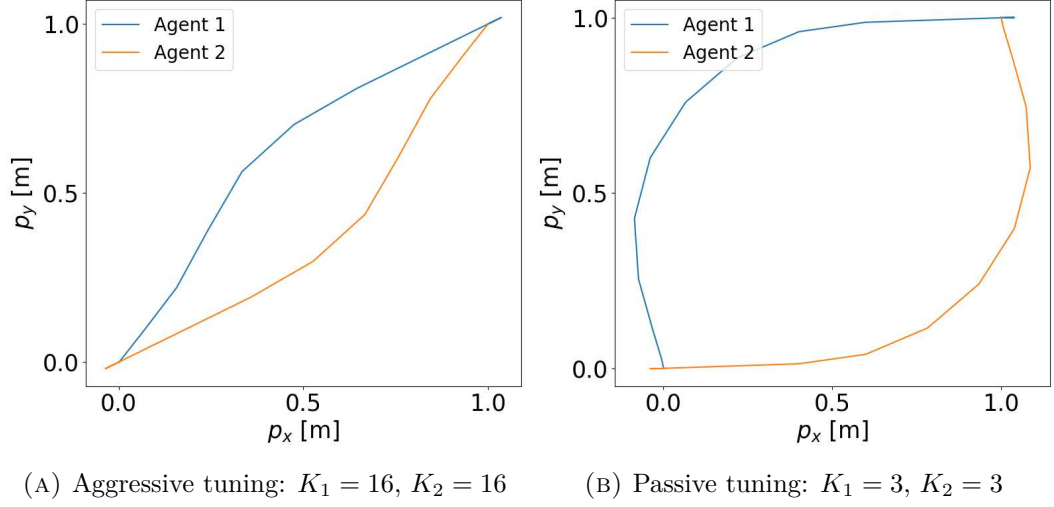


FIGURE 4.3: The paths that two agents take to switch places. When the CBF is tuned aggressively, they come closer to each other. When making the tuning more passive, the agents stay further away from each other. In both cases there are no collisions, as safety is guaranteed.

optimal trajectories, hence its name.

Fast mode: $\mu = 1, m = 1, m_{pre} = 20$.

In this mode, we maximally exploit the safety guarantee provided by the CBFs. It drastically reduces m and m_{pre} , resulting in faster execution times. Note that μ has to be reduced as well. It was observed that, when using a large μ , the first few ADMM iterations deal with the collision avoidance by simply slowing down and coming to a stop. This is of course very unsatisfactory as the agents do not reach their target this way. Therefore, either sufficient iterations have to be performed (which is done in performance mode), or μ has to be decreased. The latter also implies that agents have more difficulty in coming to a consensus, leading to less optimal trajectories. Nevertheless, they remain safe due to the CBFs.

4.3 Neighbour selection

Despite substantial improvements, ADMM's scalability remains a limiting factor for its use in very large-scale systems. The main contributor to this is NLP 4.4, as its number of variables is still related to the number of agents. It also encompasses the collision avoidance constraints. To limit the size of the subproblems in ADMM, this section aims to integrate a neighbour selection scheme [21], [9], [26] into the existing ADMM+CBF framework. An agent then only considers these neighbours in its subproblem, reducing the amount of variables and collision avoidance constraints.

This section discusses two possibilities, yielding a constant and a variable number of neighbours. As it turns out, it is not straightforward to integrate such schemes with ADMM.

4.3.1 Constant number of neighbours

One possible approach to select neighbouring agents is to rank them based on a figure of merit [9]. Each agent evaluates how much of a safety threat other agents pose and uses this measure to determine its neighbours. Then, the M most dangerous ones are chosen to be neighbours. This is an attractive strategy, as it constitutes a hard limit on the size of the subproblems. It also assures that the structure of the latter does not change in time, as the number of variables is constant.

The most straightforward figure of merit is the Euclidean distance [26], [21]. The idea is that the closest agents are seen as most dangerous and should definitely be considered neighbours. This is not necessarily the best choice as some agents might be close but moving away, which is less dangerous than an agent coming in head-on. This also does not take into account the velocity of other agents, which also plays a role in safety. A strategy that does take the latter into account, inspired by [9], is to use the value of $cbf_{i,j}$. The fact that, in our case, these values were already computed in the constraints of QCQP 4.11 motivates this choice. Hence, the lower the value of $cbf_{i,j}$, the closer they are to collision and the bigger a threat agent j is to agent i .

However, the detrimental issue of considering only the M closest or most dangerous neighbours is that it is not compatible with decentralised ADMM because it violates the bidirectional identity (Equation 2.27). The latter states that decentralised ADMM only works if the following holds: agent i is in the neighbourhood of agent j if and only if agent j is in the neighbourhood of agent i . To illustrate that this is not compatible with selecting a constant number of neighbours, consider the case in Fig. 4.4. Here, three agents each need to find their closest neighbour, i.e. $M = 1$ and the chosen figure of merit is Euclidean distance. Agents 1 and 2 pick each other, while agent 3 picks agent 2. Hence, the bidirectional identity between agents 2 and 3 does not hold. Subsequently, another strategy must be sought.

4.3.2 Variable number of neighbours

The second strategy also selects neighbours based on some figure of merit, but instead of ranking them, it defines a cut-off value for the merit function. For example, all agents within a certain radius are selected as neighbours [26], [21]. The same can be done with $cbf_{i,j}$ [9]. However, this implies that the number of neighbours M (and the neighbours themselves) may change in time. Subsequently, the ADMM subproblems have a changing amount of variables, which is not conform the current MPC strategy, which fixes the OCP structure.

To circumvent this issue, sub-problems OCP 4.3 and NLP 4.4 are still constructed in

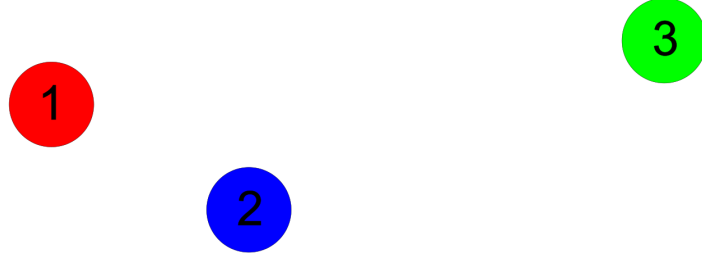


FIGURE 4.4: A neighbour selection case, where each agent needs to find their closest neighbour. Agent 3 selects agent 2, but agent 2 selects agent 1. Hence, they are not simultaneously in each other’s neighbourhood.

Rockit by including all agent’s variables but removing the objectives and collision avoidance constraints of those not selected as neighbours. This way, the problem size and structure remains the same, but they become much more tractable. The idea is that the solver does not alter variables that do not appear in the objective or in the constraints.

Time-varying neighbourhoods have another consequence. Once the strategy was implemented in the ADMM-CBF framework, it turned out that the computation time actually increased w.r.t. the case without neighbour selection. It was found that at certain specific time steps the computation time peaked. These were identified with the time steps at which the neighbourhood changed. The issue that ADMM struggles with time-varying neighbourhoods stems from the fact that each time the neighbourhood changes, ADMM has to reconverge. Moreover, the solution of the previous iteration is still based on the former neighbourhood, meaning it does not serve well as an initial guess.

There has already been some research into trying to make ADMM more compatible with time-varying neighbourhoods, e.g. in [30]. However, since this is not the main focus of this thesis, further investigation is left for future work.

As a result of these difficulties, no further neighbour selection method is applied in this thesis. An agent’s neighbourhood simply contains all other agents.

Chapter 5

Practical implementation

This thesis employs the Beckhoff XPlanar system to test and verify the algorithms in a practical and realistic setting. Hence, this chapter starts off by briefly discussing the basics of the machine. Then, the necessary steps are presented to implement the algorithms of Chapters 3 and 4 on the machine. As a first step, the trajectories are precomputed offline and fed directly to the system, which already requires some integration. Then, the trajectories are sought to be computed in real-time.

5.1 The XPlanar system

The XPlanar is a planar motor system designed to transport and position various loads with a lot of flexibility. This section briefly discusses the concept and hardware that make up the machine, as well as the software that the system uses. This discussion is based on Beckhoff's website [6] and the Beckhoff Information System [5] (which contains technical data, manuals, documentation, etc.).

5.1.1 Hardware

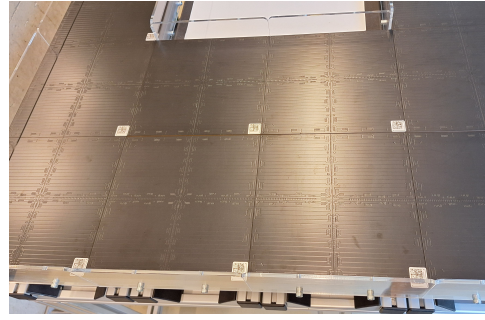
The XPlanar consists of tiles and movers which, respectively, act as stator and rotor. The movers contain only a permanent magnet, without any active components. The tiles contain coils that, when energised, make the movers float. By then energising the right coils at the right time, the tiles create a moving magnetic field that carries the movers to their desired positions. By floating a few millimetres above the surface, the movers can carry a load (e.g. pharmaceutical products, food, PCBs (Printed Circuit Board), etc.) in a precise, hygienic, frictionless, and noiseless way.

The mover and tile types used in the experimental setup are shown in Fig. 5.1. The tiles can be configured and connected in any shape to build an arena. Fig. 5.2 shows the configuration used in the experiments, along with its coordinate system convention. The first tile is then connected with an EtherCAT G fieldbus to an industrial PC which runs the software to control all this hardware. The rest of the communication goes from tile to tile. Once the physical configuration is complete, the

5. PRACTICAL IMPLEMENTATION



(A) APM4220-0000-0000 mover



(B) APS4322-0000-0000 tiles

FIGURE 5.1: The mover and tile types used in the experimental setup.

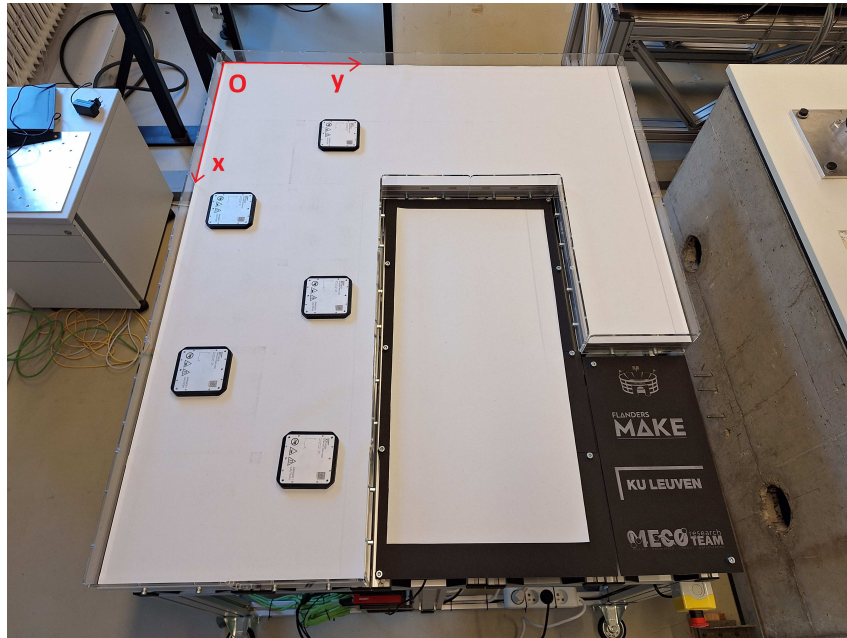


FIGURE 5.2: The XPlanar system in the robotics lab. This configuration is used in the experiments. Its local coordinate system is placed in the upper left corner of the tile configuration, with the x-axis and y-axis respectively pointing in the direction along its longest and shortest edge.

movers can be transported and positioned practically anywhere on the arena, with a position resolution of $1 \mu m$ and repeat accuracy of $\leq 5 \mu m$. All of these features result in a system that can be used with great flexibility. This motivates once again the choice of continuous optimisation-based methods (as opposed to sampling-based methods) for planning the motion of the movers with maximal usage of the given space. The broad outline of this system architecture is shown in Fig. 5.3.

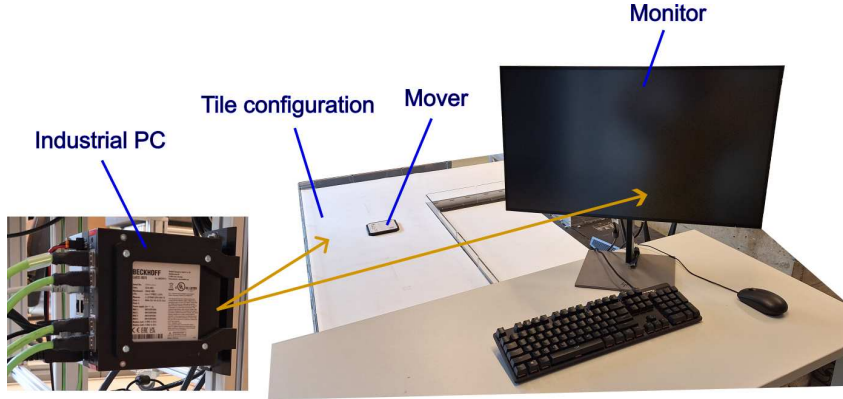


FIGURE 5.3: The XPlanar system architecture in the lab. The mover is a passive component floating above the tile. The tile contains all active components necessary to drive the movers. To that end, it receives commands from an industrial PC over an EtherCAT G fieldbus. The PC runs TwinCAT, which provides all necessary software functionalities (Section 5.1.2).

5.1.2 Software

To control the system, the industrial PC uses the TF5890 software, which is integrated in TwinCAT, a more general engineering software platform that enables real-time control of the machine from the PC. The most important features for this thesis are the Programmable Logic Controller (PLC) code, C++ integration, and the XPlanar configurator.

The PLC code contains the logic that determines the state and command mode of the movers, as well as where and when to move them. The PC then continuously loops over this code during operation. For example, to enable (float) or disable (stop floating) the movers, the PLC can define booleans for each of these states and provide logic to bring the movers to their desired state when these booleans are toggled. The PLC can also apply set-points (target positions, velocities, and accelerations) that the movers need to track through the “ExternalSetpointGeneration” command mode.

TwinCAT allows a C++ script to communicate with the PLC, which is useful for external set-point generation. It uses ADS-communication (Automation Device Specification communication) to enable the C++ code to read and write PLC variables. This way, when trajectories are generated from a separate algorithm, its output values can be written to the PLC as external set-points and applied to the system through this protocol.

The XPlanar configurator bridges the gap between the hardware configuration and software. Here, the tile configuration (layout of the arena) and the movers can be entered. Then, it automatically generates all files and objects needed for control of this specific configuration. Finally, the mover objects can be linked to the PLC,

bridging the gap.

5.2 Offline implementation details

The first step in evaluating the practical suitability of our motion planning strategies is to apply precomputed trajectories to the system. The idea is to use the algorithms from Chapters 3 and 4 to calculate the trajectories, save them to csv files and transfer them to the industrial PC. There, the files can be read out in the C++ code, and written to the PLC code whenever the movers need new set-points. As this requires consistent timing, we make use of buffers. This section discusses the necessary steps to integrate this strategy along with the algorithms on the XPlanar. To do this, the algorithms need to be updated first so that they can take into account the configuration of the arena. Then, the sampling frequency of the algorithm is matched to that of the XPlanar by interpolation. Finally, the buffer protocol is discussed from the PLC and C++ point of view and how they communicate.

5.2.1 Box constraints

When floating over the tiles, the algorithm needs to make sure that the movers do not cross the boundaries of the arena. The simplest arena that can be built using the tiles is one with a rectangular shape, hence the term “box constraints”. It is not only convenient to start with this case due to its straightforward convex shape. It also has relevance in a strategy for handling the more involved, non-convex configuration of the experimental setup (Fig. 5.2), since the latter can be seen as a concatenation of different rectangles (Section 5.2.2). The arena constraints for the single-rectangular case are:

$$\begin{aligned} p_{x,min} + (R + \epsilon) &\leq p_{x,i}(t) \leq p_{x,max} - (R + \epsilon) \\ p_{y,min} + (R + \epsilon) &\leq p_{y,i}(t) \leq p_{y,max} - (R + \epsilon) \\ \forall i \in \{1, \dots, N\} \end{aligned} \tag{5.1}$$

Each position coordinate of the agents has a lower and upper bound that depends on the length and height of the rectangle, the choice of the coordinate system, and its own dimensions. It also incorporates a safety factor ϵ as for the collision avoidance constraints. Whereas the agents had to be modelled as circles for the latter, it is not necessary for these box constraints. For 2D motion planning of these holonomic movers, their orientation does not change, which means that they always remain parallel w.r.t. the boundaries. This allows us to rewrite Constraints 5.2 in a less conservative way:

$$\begin{aligned} p_{x,min} + \left(\frac{w}{2} + \epsilon\right) &\leq p_{x,i}(t) \leq p_{x,max} - \left(\frac{w}{2} + \epsilon\right) \\ p_{y,min} + \left(\frac{w}{2} + \epsilon\right) &\leq p_{y,i}(t) \leq p_{y,max} - \left(\frac{w}{2} + \epsilon\right) \\ \forall i \in \{1, \dots, N\} \end{aligned} \tag{5.2}$$

Where $w = 0.115m$ is the width of the movers (see Fig. 3.1). Since $\frac{w}{2} = 0.0575m$ and $R = 0.0813m$, the new constraints allow the movers to approach the boundaries much more closely now. Due to various challenges discussed in Section 5.2.2, most experiments are conducted within the largest sub-rectangle of the experimental setup, which measures 6 by 2 tiles. Therefore, to be consistent with the coordinate system defined in Fig. 5.2, we find $p_{x,min} = 0m$, $p_{y,min} = 0m$, $p_{x,max} = 1.44m$ and $p_{y,max} = 0.48m$ (as the tiles have dimensions $0.24m \times 0.24m$).

Constraints 5.2 have to hold for each mover separately. Therefore, they can be incorporated as such in the centralised schemes from section 3, since these only consist of one central OCP. For the ADMM + CBF method from Section 4, we need to include the box constraints for each agent separately in their OCP 4.3, but they also need to be added jointly to QCQP 4.11, since the HOCBF step is centralised. The problem here is that the positions are no longer decision variables in the QCQP. To write Constraints 5.2 as a function of the unknown inputs (accelerations) only, a constant acceleration law was used for QCQP 4.11, resulting in the following constraints:

$$\begin{aligned} p_{x,min} + \left(\frac{w}{2} + \epsilon\right) &\leq p_{x,i}(0) + v_{x,i}(0)\Delta t + 0.5u_{x,i}(0)\Delta t^2 \leq p_{x,max} - \left(\frac{w}{2} + \epsilon\right) \\ p_{y,min} + \left(\frac{w}{2} + \epsilon\right) &\leq p_{y,i}(0) + v_{y,i}(0)\Delta t + 0.5u_{y,i}(0)\Delta t^2 \leq p_{y,max} - \left(\frac{w}{2} + \epsilon\right) \quad (5.3) \\ \forall i \in \{1, \dots, N\} \end{aligned}$$

The reason for this elementary choice is the fact that they are linear in $\mathbf{u}_i(0)$, which ensures that the HOCBF step remains a tractable QCQP. This is justified by the fact that the multiple shooting method uses a zero-order hold condition between its control grid points. Hence, the acceleration is inherently assumed to be constant in the interval $[0, \Delta t]$, meaning that Constraints 5.3 are compatible with the transcription.

5.2.2 Extension to non-convex arenas

What follows is a heuristic extension of the algorithm to non-convex arenas. The arena must satisfy one condition: it must be able to be considered as a concatenation of rectangles. The Beckhoff Xplanar as in Fig. 5.2 is used as an example. Since it is composed of a concatenation of square tiles, it always satisfies this condition.

First, the arena is divided into rectangles. It is important that each rectangle (or corridor) is chosen as large as possible. This creates overlap zones, which can be used to transfer an agent from one corridor to another. Next, the corridor transition points need to be defined. The choice of these influences the optimality of the final trajectories but can also cause jams. The corridor transition point is heuristically chosen to be the centre of the overlap zone here. A formal study of the choice of these points is beyond the scope of this thesis. These steps are shown in Fig. 5.4.

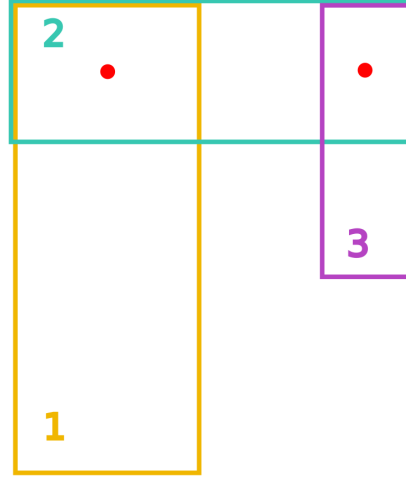


FIGURE 5.4: An example of splitting the arena into as large corridors as possible and defining the corridor transition points (red dots). Implemented on the arena of the Bechkhoff Xplanar

Here, the corridors are numbered from 1 to 3 and the corridor transition points are indicated by a red dot.

These transition points can now be interpreted as the vertices of a graph. Thus, if an agent starting in corridor 1 needs to move to a target position in corridor 3, a discrete graph planner can in principle be used to generate a list of sub-targets based on the corridor transition points that allows the agent to follow the correct sequence of corridors.

The overhead that is added to each agent is just a list of sub-targets, a list of the corresponding corridors for these sub-targets, and a variable that keeps track of the agent's current corridor. The latter is used to get the dimensions of the current corridor so they can be used in the optimization problems.

Switching between corridors occurs when the agent is in the overlapping zone of 2 (or more) corridors. Once this happens, the agent can start tracking the next (sub-)target. The agent's current corridor is also updated. This process is repeated until the agent is at its final target. This flow is illustrated in Fig. 5.5.

It is also possible to consider the exterior of the arena as a complex obstacle. There exist methods to deal with complex obstacle shapes that could subsequently be used for motion planning in non-convex arena, as in [27]. However, after implementation, this strategy turned out to be ill-suited for the problem at hand, yielding much more computational overhead. We conclude that the heuristic corridor strategy is better suited for the XPlanar. A simulation of this scheme can be found on the Github

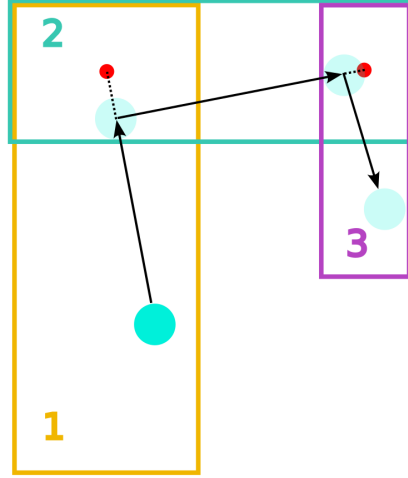


FIGURE 5.5: An example of how an agents tracks sub-targets to change corridors. The final sub-target is the destination of the agent. The transition points are heuristically chosen and lead to sub-optimal trajectories, but allow to consider non-convex arenas.

repository accompanying this thesis¹.

5.2.3 Trajectory interpolation

The frequency at which the XPlanar feeds set-points to its movers is $100Hz$. The frequency of the motion planners was taken to be larger because they need sufficient time to do computations in between time steps, as it would eventually need to perform them in real-time. The specific choice of $10Hz$ (or a time step of $0.1s$) for the motion planners is taken for the latter reason, but it also ensures that its time step is a multiple of the time step of the XPlanar. Consequently, the trajectories can be easily interpolated as the original time grid aligns with the interpolated time grid.

To interpolate the trajectories between two grid points t_1 and $t_1 + 0.1s$, the system is integrated from t_1 , but with time steps of $0.01s$ this time. Within the original interval $[t_1, t_1 + 0.1]$, the acceleration is kept constant, as is assumed in the multiple shooting method.

As discussed, a caveat about the multiple shooting method is that it only enforces constraints on the control grid. These new interpolated trajectory points provide convenient data that can also be used for extra safety checks to verify if the trajectories are safe between the control grid points. Consequently, before applying a trajectory to the XPlanar, the mutual distances between agents are checked based on these new points, as well as their distance to the arena walls.

¹<https://github.com/StanServaes/ADMM-CBF-Multi-Agent-Motion-Planning.git>

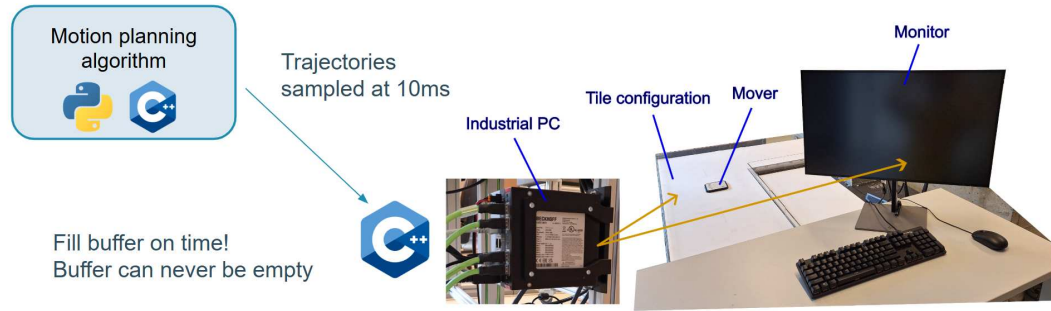


FIGURE 5.6: Communication between XPlanar, the C++ code, and the motion planning algorithm. In case of offline motion planning, the motion planning algorithm is replaced by a csv file containing a precomputed trajectory.

5.2.4 Communication with XPlanar

As discussed, the XPlanar allows a C++ script to communicate with the PLC code and its variables. This is used for offline motion planning, as the trajectories are precomputed and saved in a csv file. This file can then be read out by the C++ code and sent to the PLC code in a systematic manner.

To ensure consistent timing and limit the amount of data in the PLC code itself, buffers are used. This way, in addition to offline motion planning, the C++ code could also be coupled with a real-time motion planner, as shown in Fig. 5.6. The PLC uses a FIFO buffer (First In, First Out) with space for 40 trajectory samples (equivalent to 0.4s of trajectory data), each consisting of position, velocity, and acceleration data. The C++ code also uses a buffer but with a length of 10 samples.

The PLC code continuously feeds set-points to the movers and simultaneously checks the status of its buffer. When it contains only 30 samples or less, the PLC sends a request to the C++ code. This code then prepares a buffer of 10 samples and copies them into the PLC buffer. As long as the latter is not empty, the movers can keep going. Hence, if a real-time motion planner provides the set-points, it must make sure that it provides them at a sufficient rate to keep the PLC buffer full. Since this last one can contain 40 samples, there is room for unexpected peaks in computation time up to 0.4s (without any additional buffering), but overall the set-points must be fed sufficiently fast. However, in the current case of offline motion planning, this constitutes no challenge yet. The C++ code must simply read values from a csv file, which always happens in time.

5.3 Real-time implementation details

This section discusses different ways to accelerate the execution of the hybrid method (Chapter 4). This is done to achieve real-time execution for as many agents as possible. First, a more tailored solver is used in Python. Then the hybrid algorithm is translated into C++ because it is inherently faster than Python. As a final step, the structure of ADMM is used, and the algorithm is multithreaded.

5.3.1 C++ implementation and multithreading

The first step in speeding up the computations is to solve OCP 4.3 with FATROP [33], which is a solver specifically tailored to solve non-linear OCPs. The next step involves translating the algorithm into C++, because it allows for faster execution. The serialised functions from Chapter 4 can be reloaded into C++ as they were previously constructed and serialised for the tests in Python. A comparison is made between each acceleration step at the end of this section.

The final acceleration step is multithreading. In the previous chapters, ADMM was still performed sequentially. A major advantage of decentralised ADMM is that the problem structure allows certain steps to be multithreaded. Hence, in Algorithm 3, the lines containing “for all agents”, such as lines 10, 14, 23 and 27, can be executed in parallel each time step. This is possible because in OCP 4.3, NLP 4.4, and the λ -update 4.5, the solution of each time step does not depend on the solutions of the other agents during the same time step.

It is important to note that not all steps can be performed in parallel. For example, the OCP and NLP cannot be performed in parallel because the NLP uses the solutions of the OCP. In principle, for a given agent, the NLP can be executed if this agent and all agents in its neighbourhood have solved the OCP. However, as pointed out in Section 4.3.2, an agent’s neighbourhood contains all other agents in this report. This can be used to simplify the multithread scheme so that first all OCPs have to be executed before the NLPs can be solved.

This multithread scheme is illustrated in Fig. 5.7. Each agent solves the OCP for itself in parallel and communicates the solution to a central memory. Only after each agent has communicated its solutions can the NLP start. The latter, along with the λ -update, is also executed in parallel, again followed by a communication step. These communication steps are denoted as a central step because they also serve to bring synchronisation to the scheme. Lastly, the input is mapped to the safe set using QCQP 4.11 and simulated.

An important note is the fact that IPOPT, as distributed with CasADi, uses MUMPS [2] as a linear system solver. MUMPS is a parallel sparse direct solver that can solve large sparse linear systems. However, it is not tread-safe and thus cannot be used in the multithread scheme. A suboptimal solution to this is to use FATROP, which

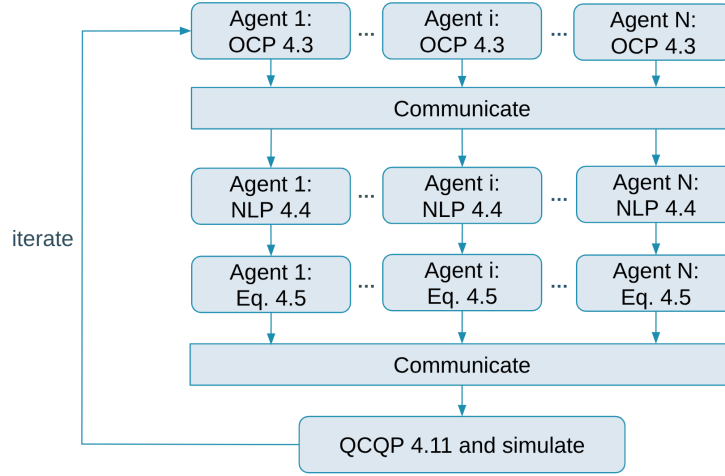


FIGURE 5.7: A schematic representation of the multithread scheme in which the parallel steps are followed up by a central communication and synchronization step.

is thread-safe, for both OCP 4.3 and NLP 4.4, although the latter does not entail dynamic constraints and is therefore technically not an OCP. This decision was made because FATROP readily comes with CasADi and was the only direct alternative to IPOPT.

To examine the influence of the different improvements that accelerate the computations, a simulation of ADMM-CBF performance mode is performed for 5 and 10 agents. The following implementations of the hybrid method are used in the comparison:

- Sequentially in Python using IPOPT for both optimization problems OCP 4.3 and NLP 4.4.
- Sequentially in Python using FATROP for the OCP and IPOPT for the NLP.
- Sequentially in C++ using FATROP for the OCP and IPOPT for the NLP.
- Parallel in C++ using FATROP for both the OCP and NLP.

The results can be found in Fig. 5.8. On the x-axis every new modification to the hybrid method is listed from left to right. The two letters between the brackets denote the solver used for the two optimization problems OCP 4.3 and NLP 4.4, “I” for IPOPT and “F” for FATROP. The first letter represents the OCP and the second letter represents the NLP. The horizontal line indicates the threshold for real-time implementation. Only the parallel implementation of the hybrid method in C++ has a computational time below this threshold, allowing for real-time application.

The relative decrease in computational time derived from Fig. 5.8 is given in Tab. 5.1.

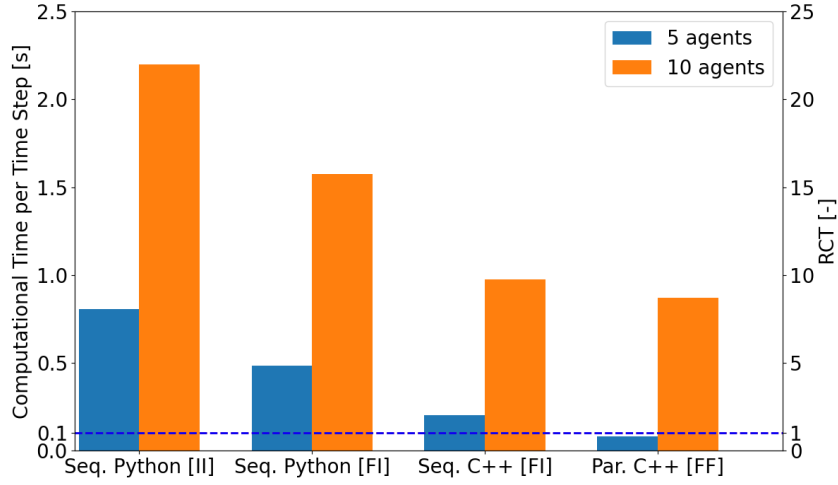


FIGURE 5.8: Comparison of the different steps taken to accelerate the computation of the hybrid method. These tests are performed for 5 and 10 agents with ADMM-CBF performance mode. Every step decreases computational time, except parallel implementation in C++ for 10 agents.

The first thing to note is that for each improvement, the relative decrease in computational time for 5 agents is greater than for 10 agents. This is because overall, NLP 4.4 accounts for the largest computational time. This is also indicated numerically by Tab. 5.2. The relative total time in NLP denotes how long the program was busy solving the NLP compared to the total time it took for the program to finish. The higher the number of agents, the higher the relative computational time of the NLP is and improvements of the algorithm relatively have less effect.

Another reason is due to the fact that there is a limit on the number of agents that can be run in parallel. Each computer has a physical amount of hardware threads; if the number of agents exceeds this amount, the program starts running in pseudo-parallel. It is then up to the scheduler of the operating system on how to select the order of threads. On top of this, each agent is currently assigned two threads: one for the OCP and one for the NLP (and Equation 4.5). This means that there is a lot of context switching between threads, causing small delays each time. A remedy to this is presented in Section 7.1. In total, the relative decrease in computational time after all these steps is 90.1% for 5 agents and 60.5% for 10 agents.

What also stands out is the low relative decrease of 11.0% when switching from sequential to parallel execution for 10 agents. This is because multithreading uses FATROP for the NLP. As mentioned earlier, the NLP does not conform to the form of an OCP so FATROP has difficulty solving problems with a larger number of agents. Details on the scalability of using FATROP for the NLP are discussed in Section 6.1.3.

Transition	5 Agents	10 Agents
Seq. Python [II] $\rightarrow$ Seq. Python [FI]	39.8 %	28.4 %
Seq. Python [FI] $\rightarrow$ Seq. C++ [FI]	57.9 %	38.1 %
Seq. C++ [FI] $\rightarrow$ Par. C++ [FF]	60.8 %	11.0 %
Total	90.1 %	60.5 %

TABLE 5.1: Relative decrease in computational time for each step that speeds up the computations. This corresponds to Fig. 5.8.

	5 Agents	10 Agents
Relative total time in NLP	70.2 %	92.8 %

TABLE 5.2: The relative total time the program is busy solving the NLP. The larger the number of agents, the more time (with respect to the total time of the program) is spent solving the NLP. This test was done using the parallel C++ implementation.

Due to the synchronisation between threads, it is the sum of the durations of the OCP and NLP that determines the total duration of an ADMM iteration. The total computational time of the hybrid algorithm implemented sequentially can therefore not be divided by the number of agents to obtain an estimate of the computational time of the parallel implementation. This would be the theoretical lower bound. This can be seen in Fig. 5.9, which shows the distribution in computation time over the steps of 1 ADMM iteration. The duration of each time step up to $t = 3s$ is around $30ms$ and $15ms$ for the sequential and parallel implementation, respectively. This results in a factor 2 difference in computational time instead of a factor of 5. Another result of the thread synchronisation is that the peaks in calculation time of the NLP become more pronounced. This figure again shows that the NLP is the bottleneck as it takes up most of the computation time. It also showcases that the QCQP is indeed very computationally efficient compared to the OCP and NLP.

5.3.2 Real-time implementation on the XPlanar

When the parallel program was implemented on the Beckhoff PC, a sharp decrease in performance was observed. For instance, a given case took 5 times longer to compute than on a Linux computer with a 12th Gen Intel® Core™ i5-12450H processor. Subsequently, we suggest the calculations to be performed on a separate PC and sent to the Beckhoff PC in the future. A first advantage of this is functional independence: if later one of the two computers needs to be replaced, this can be done without affecting the other computer.

A second advantage is that if more hardware threads are needed to run a larger

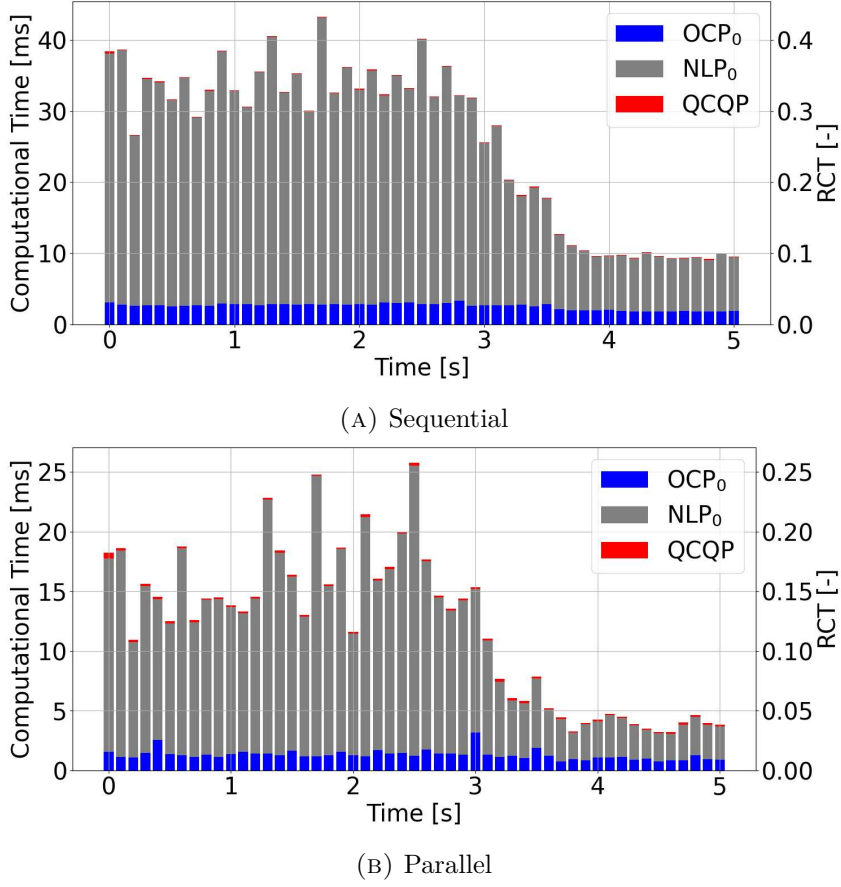


FIGURE 5.9: The durations of each step in the hybrid method stacked on top of each other for every time step. This test was performed with ADMM-CBF fast mode for 5 agents. Even if the number of agents is below the number of hardware threads, the parallel implementation is not simply 5 times faster than the sequential implementation. This is because the sum of the longest OCP and NLP duration determine the computational time of a parallelised ADMM iteration.

number of agents in parallel, more computers can be added without influencing the Beckhoff PC. This scheme is represented in Fig. 5.10. There is one coordinator computer that provides synchronisation with all other follower computers regarding the parallel computations. It also stores the trajectories that have already been calculated. These trajectories can be transmitted by request from the Beckhoff industrial PC.

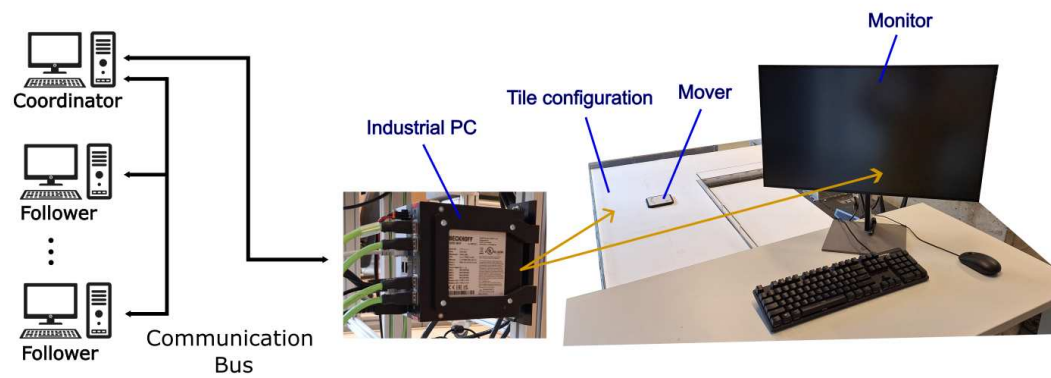


FIGURE 5.10: Proposed diagram of how calculations can be distributed across multiple computers. A coordinator computer stores the trajectories and can forward them on request from the Beckhoff industrial PC.

Chapter 6

Results and discussion

In this chapter, the algorithms are first compared in simulation. This comparison includes performance, safety, and scalability. Then, the maximum real-time capability of the hybrid method is investigated. After this, the calculated trajectories are implemented on the Beckhoff Xplanar to verify their practical suitability.

6.1 Simulation results

This section quantitatively examines how the different algorithms behave. This is done by letting all algorithms perform the same two cases for 5 and 10 agents. These results are then used to compare performance and safety. At the end, two scalability analyses are also done. The first inspects the scalability of centralised MPC, ADMM-CBF fast mode, and ADMM-CBF performance mode, implemented in Python. The second inspects the scalability of the hybrid method, parallel and sequential, in C++. At the end, it is investigated for how many agents the hybrid method can maximally run in real-time.

6.1.1 Performance comparison

Performance refers to both time-optimality and the possibility of real-time implementation, as this is a trade-off that ADMM enables. The comparison is performed by considering two cases, one for 5 agents and 10 agents. The agents in both cases are initialised in an imaginary square. The size of this imaginary square is based on the agent density, which is set to $\rho_{agent} = 5 \text{ agents}/m^2$ throughout this chapter. This heuristic is meant to introduce consistency between the initial conditions of all the simulations with different amounts of agents. It does not define an arena in which the agents must stay as the box constraints are not considered in the following results. Subsequently, l_{square} , which represents the length of the the imaginary square's, is determined as follows:

$$\rho_{agent} = \frac{N}{l_{square}^2} \iff l_{square} = \sqrt{\frac{N}{\rho_{agent}}} \quad (6.1)$$

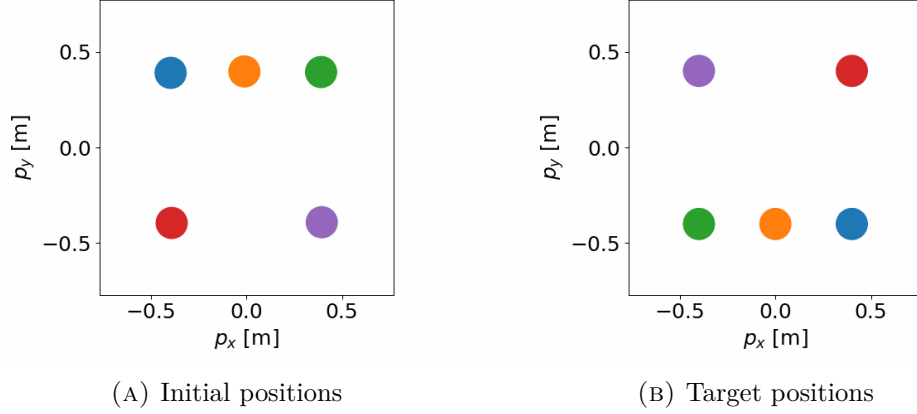


FIGURE 6.1: The initial and target position of each agent for the case of 5 agents.

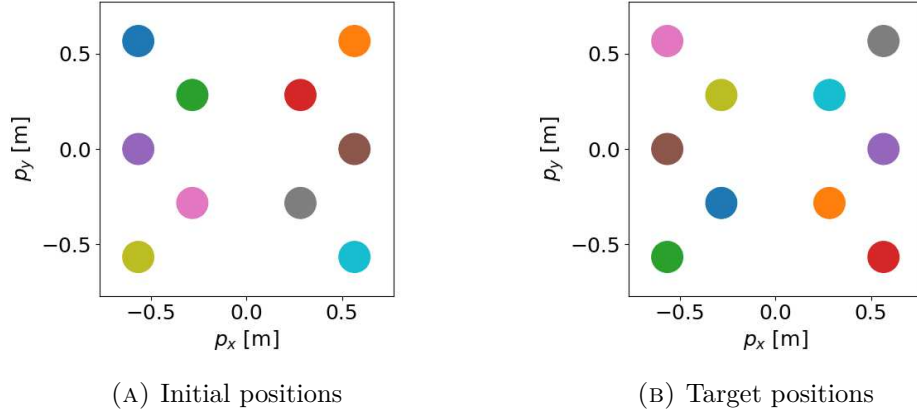


FIGURE 6.2: The initial and target position of each agent for the case of 10 agents.

Where N denotes the number of agents.

The case with 5 agents is shown in Fig. 6.1. The agents start at rest on the vertices of a square, with one additional (orange) agent in the middle of the upper side. The goal is to cross the space towards the symmetrically opposite vertex (or side) as fast as possible without colliding. They should also arrive at their destination at rest. This information is embedded in the inputs of the algorithms $\mathbf{x}_{init,i}$ and $\mathbf{x}_{f,i}$. Note that this is already a computationally expensive case, as many agents will have to pass through the middle at the same time. Moreover, two pairs of agents will have to switch positions, which imposes an extra difficulty on the solver of the optimization problem, as there is no preferred swerving direction.

The case of 10 agents is shown in Fig. 6.2. As for the case of 5 agents, they have to exchange positions among themselves, but here not all agents need to go through the centre of the arena.

The algorithms from Chapters 3 and 4 are implemented in Python using the software toolkit Rockit. The OCPs are solved with the multiple shooting method (Section 2.1.2) using IPOPT (Section 2.1.3). In these tests, the sequential ADMM implementation is used as there is no parallel version in Python. For each algorithm, the case of 5 and 10 agents is implemented. When using ADMM, agents often take a long time to reach their target position, e.g. within a $1mm$ threshold margin. This affects tests because these final iterations are relatively faster than when agents have to actively plan with each other to avoid collisions. Because of this, the threshold for checking whether agents have reached their target position is set to $0.015m$, both for centralised MPC and ADMM. This is justified because in practice, when an agent is close to its target, motion planning is passed on to a task-specific planner or controller.

As the total computation time depends on the length of the simulation, it is preferred to look at the relative computation time:

$$RCT = \frac{\text{computational time}}{\text{transit time}} \quad (6.2)$$

On average, the computation can be performed in real-time when this number is smaller than 1. The resulting transit time and RCT for all motion planners is shown in Table 6.1. The corresponding trajectories for each are shown on the Github repository accompanying this thesis¹.

	Transit Time [s]		RCT [-]	
	5 Agents	10 Agents	5 Agents	10 Agents
Free-time	1.38	1.35	-	-
Centralised MPC	1.5	1.6	1.61	10.60
ADMM-CBF Fast mode	4.6	6.3	1.16	3.18
ADMM-CBF Performance mode	2.8	4.7	12.06	31.03

TABLE 6.1: Transit time and RCT for the 5 and 10 agent cases compared between the different motion planners. As the free-time problem is a method that is only performed once offline, the RCT has no meaning for this method. These tests were performed in Python on a Linux pc with a 7th Gen Intel® Core™ i7-7700K CPU @ 4.20GHz.

What follows is an explanation of these results per algorithm based on Tab. 6.1.

Free-time problem Since the problem is non-convex, it is not possible to say whether this result constitutes a global minimum or not. However, it can still be used as a baseline to compare the other algorithms since it produces the lowest transit time of all. The RCT has no meaning for free-time because it does not offer the possibility of real-time implementation.

¹<https://github.com/StanServaes/ADMM-CBF-Multi-Agent-Motion-Planning.git>

The case with 10 agents has a slightly lower transit time than the case of 5 agents, even though the agents are placed within a larger square ($1.41m$ for 10 agents, $1m$ for 5 agents). This is because in the former case, less agents need to cross the middle.

Centralised MPC For both cases, the agents take longer to reach their target position than in the free-time implementation. This is due to the following two reasons. First, there is a difference between the cost function of free-time and MPC. In free-time, the transit time is minimised directly, while in MPC the distance to the target position integrated over time is contained in the cost. This causes more agents to cross each other in the centre of the arena, whereas free-time lets some agents take a diversion from the start to leave more room in the middle for other agents, resulting in an overall lower transit time. The second reason is that the trajectories exhibit a small overshoot.

ADMM-CBF Compared to free-time and centralised MPC, the transit times for both modes of ADMM-CBF are higher. However, the advantage of ADMM is that it allows to trade optimality for computational load. This is pronounced when comparing fast mode with performance mode. Performance mode is, on average for the two cases, able to reduce the transit time by $1.7s$ of fast mode at the expense of a higher RCT .

Fast mode also has better real-time capability than centralised MPC as its RCT is lower. These tests were executed on the same computer as the scalability tests (Section 6.1.3) to maintain consistency within the algorithms implemented in Python. Note that this computer is old. If the same case of 5 agents is run on a modern Linux pc with a 12th Gen Intel® Core™ i5-12450H processor, fast mode is able to run in real-time with $RCT = 0.78$, while centralised MPC still has $RCT = 1.27$. Also note that the ADMM scheme is still implemented sequentially in this section. The methods mentioned in Section 5.3.1 can further decrease RCT .

In addition, the change in RCT from the case of 5 agents to 10 agents illustrates another advantage of ADMM. That is, it only increases by a factor less than 3, while for centralised MPC it increases by a factor of 6.58. This shows that ADMM has a more promising capability for real-time execution with an increasing number of agents. This is investigated more formally in Section 6.1.3.

The difference in transit time between fast mode and performance mode can be explained as follows: In performance mode, due to the choice of $m = 20$ and $\mu = 40$, the ACR (Equation 4.6) decreases more than in fast mode. Subsequently, the agents anticipate and evade each other in a timely manner. In fast mode, due to its choice of $m = 1$ and $\mu = 1$, the ACR does not decrease enough for the agents to come to a consensus, meaning that the agents will collide. The QCQP slows down the agents and makes sure they do not collide. But it does not entail a time horizon

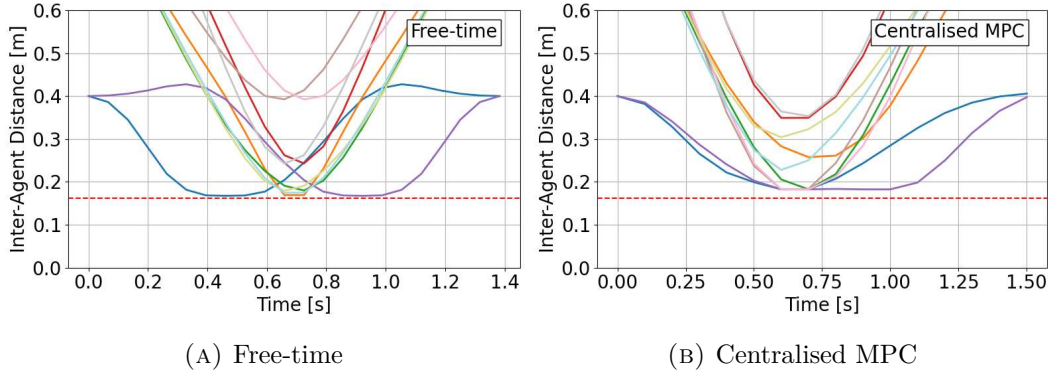


FIGURE 6.3: The inter-agent distance for the case of 5 agents for free-time and centralised MPC. The red dashed line denotes the minimal distance between the centres of the agents before they collide. Both free-time and centralised MPC approach the threshold up to their safety margins, $5mm$ and $20mm$ respectively.

and cannot plan trajectories. This results in the agents forming a cluster that slowly turns around its centre, as ADMM can still provide a direction in which the agents should turn.

6.1.2 Safety comparison

Safety and conservatism is measured by the inter-agent distance. If the distance between the centres of two agents becomes less than two times the radius, this means that the agents collide and the trajectories are not safe. Here, the case of 5 agents is used to showcase the safety of the different algorithms.

The inter-agent distance for free-time and centralised MPC can be found in Fig. 6.3. This figure shows the distance between the centres of all agent pairs. The red dashed line indicates when the distance between two centres is equal to $2R$. This figure shows that both free-time and centralised MPC are capable of generating safe trajectories, as for both plots the inter-agent distance remains above the red line. The margin with respect to collision (the minimum distance between the lowest point of the graphs and the red dashed line) is for both algorithms equal to the specified safety margin, $5mm$ and $20mm$ for free-time and centralised MPC respectively. This is because both algorithms do not introduce conservatism (apart from modelling the movers as circles) by incorporating the exact collision avoidance constraints based on the Euclidean distance.

The safety of ADMM-CBF fast mode and performance mode can be seen in Fig. 6.4. In fast mode, the agents remain close together for approximately $3s$. This is in line with what was mentioned earlier, namely that the method is not yet converged in fast mode, meaning the agents entail elevated average consensus residuals (ACRs). This implies that the originally planned trajectories may still involve collisions. However,

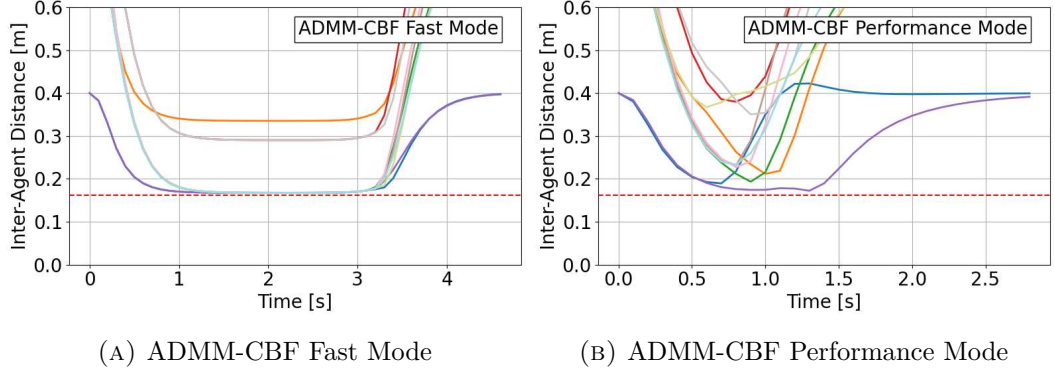


FIGURE 6.4: The inter-agent distance for the case of 5 agents for ADMM-CBF fast mode and performance mode. The red dashed line denotes the minimal distance between the centres of the agents before they collide. Fast mode exhibits a margin of $5mm$ equal to the safety margin, while performance mode had a margin of $10mm$ which is larger than its safety margin.

QCQP 4.11 prohibits this by rendering the input safe. This results in the agents slowing down when they are about to crash and then cluttering together. When this happens, there is a moment when the agents are almost stationary, so their relative acceleration and velocity become negligible. $cbf_{i,j}$ (Equation 4.9) is then reduced to the collision avoidance constraint based on Euclidean distance as in free-time and centralised MPC. Because of this, the margin until collision is the same as the safety margin, $5mm$. Note that in more common situations where the agents do not directly approach each other head on, this cluttering is less pronounced.

In performance mode, the trajectories have lower ACRs than in fast mode and are consequently more optimal, leading to less intervention of the QCQP and higher velocities of the agents. These velocities are taken into account in the CBF and consequently the QCQP renders the inputs safe in a timely manner. Unlike fast mode, agents now do have relative acceleration and speed at all times, this ensures that the terms involving velocity and acceleration in $cbf_{i,j}$ are no longer negligible. This makes the margin to collision (now $10mm$) greater than the safety margin $\epsilon = 5mm$.

To prove the necessity of the QCQP to ensure safety, the inter-agent distance for ADMM with $m = 50$ and a penalty parameter of 70 is shown in Fig. 6.5. In this figure, there are three moments where safety is violated (curves go below the red dashed line) and agents collide. This shows that even with a high number of ADMM iterations, the trajectories are not collision-free. This makes ADMM without the QCQP not useful in our case.

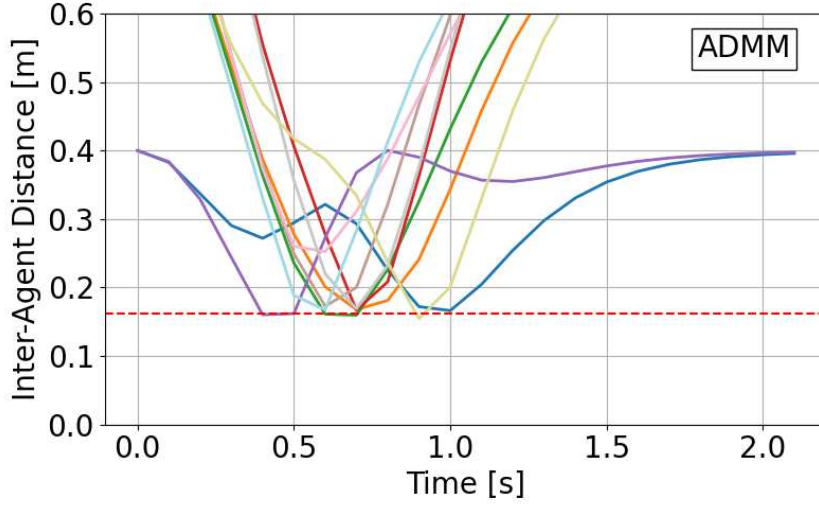


FIGURE 6.5: The inter-agent distance for the case of 5 agents for ADMM with $m = 50$ and $\mu = 70$. The red dashed line denotes the minimal allowed distance between two agent centres. Even though 50 ADMM iterations are performed during each time step, there are still collisions.

6.1.3 Scalability comparison

To examine how (sequential) ADMM-CBF behaves relative to centralised MPC with an increasing number of agents, tests were conducted up to 30 agents in Python. These tests each consist of 20 cases in which the agents are randomly placed in a square whose length is calculated based on Equation 6.1 with $\rho_{agent} = 5$ to keep the tests with different numbers of agents consistent with each other. The target positions are also chosen randomly. The only condition is that the target position is at least half the side of the imaginary square away from the initial position.

The scalability of centralised MPC, ADMM-CBF fast mode, and ADMM-CBF performance mode is shown in Fig. 6.6 as a log-log plot with the computational time per time step (where $\Delta t = 0.1s$) on the y axis. The blue dashed line denotes the real-time capability (as $RCT = 1$). The scalability exponent is denoted by the slope s of the last 11 tests. This is to exclude any kind of transition phenomenon which has to do with the fact that for a lower number of agents, the overhead of setting up the optimisation problem each time step now contributes relatively more to the total computational time.

The plot indicates that the scalability for ADMM-CBF fast mode and performance mode is equal. ADMM-CBF scales with $\sim O(N^{1.83})$. Performance mode shows a vertical upward translation in relation to fast mode. This corresponds to performance mode performing 20 times more ADMM iterations per time step than fast mode. This vertical shift on a log-log plot means that performance mode is on average 15.15

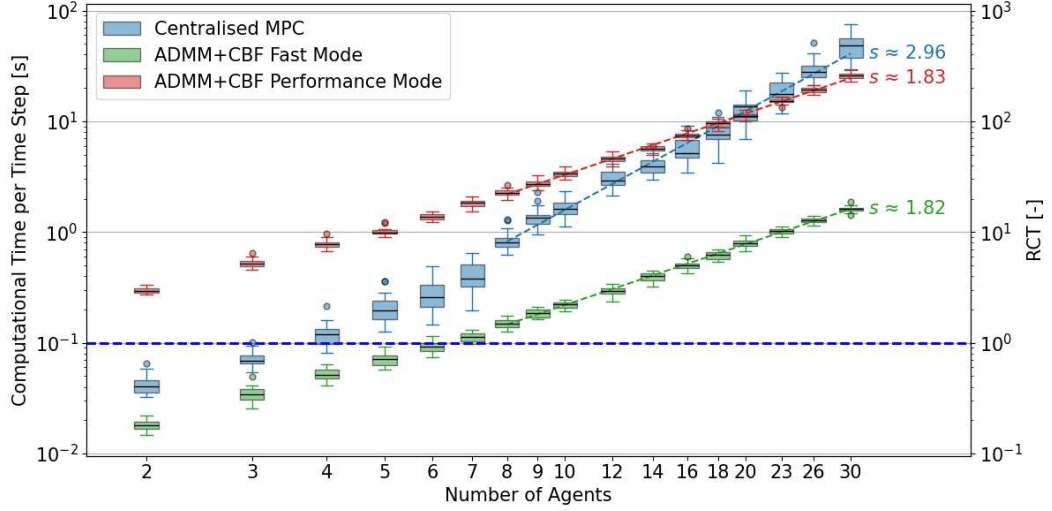


FIGURE 6.6: Scalability plot of centralised MPC and ADMM+CBF (fast mode and performance) with 20 random cases for each number of agents. The blue line denotes the real-time capability threshold. The scalability is denoted by s . These tests were made in Python on a Linux computer with a 7th Gen Intel® Core™ i7-7700K CPU @ 4.20GHz.

times slower than fast mode. This factor is smaller than 20 because the last ADMM iterations are less computationally expensive. It should be noted that these tests also showed that trajectories become less optimal for an increasing number of agents. This is because the same tuning was used for all cases. We conclude that the ADMM parameters are dependent on the number of agents.

The plot also shows that the scalability exponent of ADMM-CBF is lower than centralised MPC: $O(N^{1.83})$ versus $O(N^{2.96})$. This emphasises again that, apart from its more tractable structure, ADMM-CBF allows fewer iterations to be performed, and it can do so without compromising safety. This is in contrast to centralised MPC, where a computationally expensive OCP is solved to full convergence in each time step. Another consequence of this is a larger variance in computational time, which can be observed through the vertical spread between the first and third quartile of the box plots. Fast mode also has a lower RCT value than centralised MPC for any number of agents, indicating that it is capable of calculating the trajectories of more agents in real time.

Next, the scalability of the multithreaded scheme is investigated. For this, tests are conducted up to 20 agents in C++ where each test consists of 10 random cases. Again, the agents are initialised as mentioned earlier to keep consistency for any number of agents. The box constraints are included in the optimisation problems, but the arena was taken large enough so that they do not affect the tests. The result is shown in Fig. 6.7, with the computational time per time step on the y-axis. The

blue dashed line again denotes the real-time capability. The scalability exponent is denoted by the slope s of the last 6 tests, to exclude any transition phenomena. These tests are carried out with $m = 10$ and $\mu = 5$. This is done instead of testing both fast mode and performance mode in order to not overload the scalability plot. Also note that these tests are executed on a different computer than the scalability tests in Python.

Theoretically, the parallel implementation is expected to show a kink in the scalability plot. When the number of hardware threads is larger than or equal to the number of agents, the scheme is executed purely in parallel. If more agents are added, the scalability increases as some calculations are performed in pseudo-parallel. After there are no free hardware threads for new agents, a mix of parallel and sequential computations is performed (optimised by the operating system scheduler), leading to a higher slope in the scalability plot. However, this is not reflected in the plot. The scalability plot of the parallel implementation shows a scalability of as much as $O(N^{4.24})$. This can be explained using the scalability plot for the sequential implementation, where once FATROP and once IPOPT is used to solve the NLP.

If IPOPT is used to solve the NLP, approximately the same scalability is obtained as in the Python scalability tests ($O(N^{1.76})$). However, if this solver is now replaced by FATROP, the scalability increases to $O(N^{4.15})$. Since the code for both tests is exactly the same and the only difference is the selection of the solver, it can be shown that the poor scalability comes from using FATROP for solving an NLP since it does not have the correct form of an OCP. The choice to use FATROP for solving the NLP was based on the fact that IPOPT is not thread-safe and FATROP is the only alternative which is directly available in CasADi. However, this conclusion only applies for $N \geq 6$, as using FATROP to solve the NLP up to 6 agents is still faster than IPOPT, as shown in Fig. 6.7.

From these tests, it can be seen that the parallel implementation is on average 1.85 times faster than the sequential implementation with FATROP for the NLP. However, multithreading is relatively less beneficial (for a given number of hardware threads in the computer) as the number of agents increases. The scalability plot of the parallel implementation will then approach, but always stay under, that of the sequential implementation. This explains the slightly higher scalability of parallel compared to the sequential implementation ($s = 4.24$ versus $s = 4.15$).

6.1.4 Maximum real-time capability

To check the maximum number of agents for which motion planning can be done in real-time, we verify for fast mode and performance mode the highest number of agents for which the RCT remains smaller than 1. These tests are performed on a modern Linux computer with a 12th Gen Intel® Core™ i5-12450H processor.

For performance mode, the maximum is found to be 5 agents with $RCT = 0.86$,

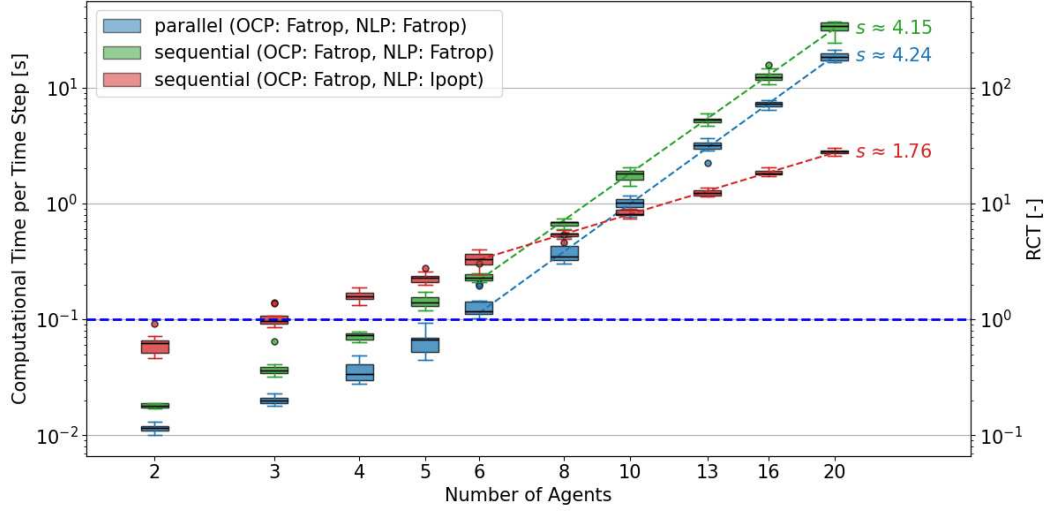


FIGURE 6.7: Scalability plot of the different implementations of the hybrid method ($m = 10$ and penalty parameter of 5) with 10 random cases for each number of agents. The blue horizontal line denotes the real-time capability threshold. The scalability exponent is denoted by s . These tests were run on a Linux pc with a 10th Gen Intel® Core™ i7-10750H processor @ 2.60GHz, with 12 hardware threads.

achieved with the parallel implementation of ADMM-CBF. Since this number of agents is low enough, the NLP is solved faster in FATROP than IPOPT. The same test using the sequential scheme with IPOPT as solver for the NLP results in $RCT = 2.95$.

Fast mode can plan trajectories for up to 13 agents in real time, with $RCT = 0.93$. This can be achieved with the sequential scheme, in which IPOPT is used to solve the NLP. Here, the number of agents is too high for multithreading to outweigh the poor scalability of using FATROP to solve the NLP. If a more dedicated thread-safe solver were used for the NLP, this number could be driven even higher because, in principle, parallel implementation should always be faster than sequential implementation. Also, if two different instances of the sequential scheme are called with IPOPT as the solver for the NLP (e.g. in two different terminals), then no problems occur with IPOPT. So another remedy is to multiprocessing the code instead of multithreading it.

Altogether, if real-time motion planning is required, the number of ADMM iterations per time step m and the step size μ can be fixed for a given number of agents to have the most optimal trajectories while still maintaining the RCT below 1.

6.2 Experimental results

As a proof of concept and to verify whether the trajectories produced by the motion planning algorithms from Chapters 3 and 4 are suitable in real industrial applications,

they are tested on the XPlanar with 5 movers. Due to the practical issue mentioned in Section 5.3, it was not yet possible to perform real-time motion planning on this system, although ADMM-CBF has real-time capability. Thus, the trajectories are currently still precomputed. Moreover, the maximum velocity and acceleration of the movers had to be set to $v_{max} = 1 \text{ m/s}$ and $a_{max} = 5 \text{ m/s}^2$ to avoid exceeding the power limit of the tiles.

Fig. 6.8, 6.9, and 6.10 show the trajectories calculated by centralised MPC, ADMM-CBF performance mode, and ADMM-CBF fast mode respectively. They show a side-by-side comparison between simulation and experiment. It is apparent that the trajectories are suitable for practical applications. The movers behave as desired in the simulation. Their trajectories are safe and compatible with the XPlanar.

Moreover, the downward trend in the optimality of the trajectories is once again visible, highlighting the “optimality versus computation time” trade-off. Although centralised MPC is computationally expensive, it ensures the movers cross the arena in 1.9s . ADMM-CBF performance mode drives the agents to their target in 3.1s , but also still entails sufficient predictive capability to perform timely collision avoidance manoeuvres. On the other hand, ADMM-CBF fast mode again shows more reactive behaviour, leading to more cluttered trajectories with a transit time of 5.2s .

Furthermore, the slight conservatism induced by the choice to model the movers as circles can be observed, at least for the collision avoidance constraints. The distance between agents always remains larger than their diagonal, even when passing each other laterally. It is also shown, especially in Fig. 6.10, that this is not the case for the agent-wall distance, since the box constraints are formulated in a non-conservative way (Section 5.2.1). While the circles in the simulation appear to violate the box constraints, the underlying squares remain within the arena. Lastly, we presented similar tests to first-year students during a lab tour in the context of an info session on their specialisation choice.

6. RESULTS AND DISCUSSION

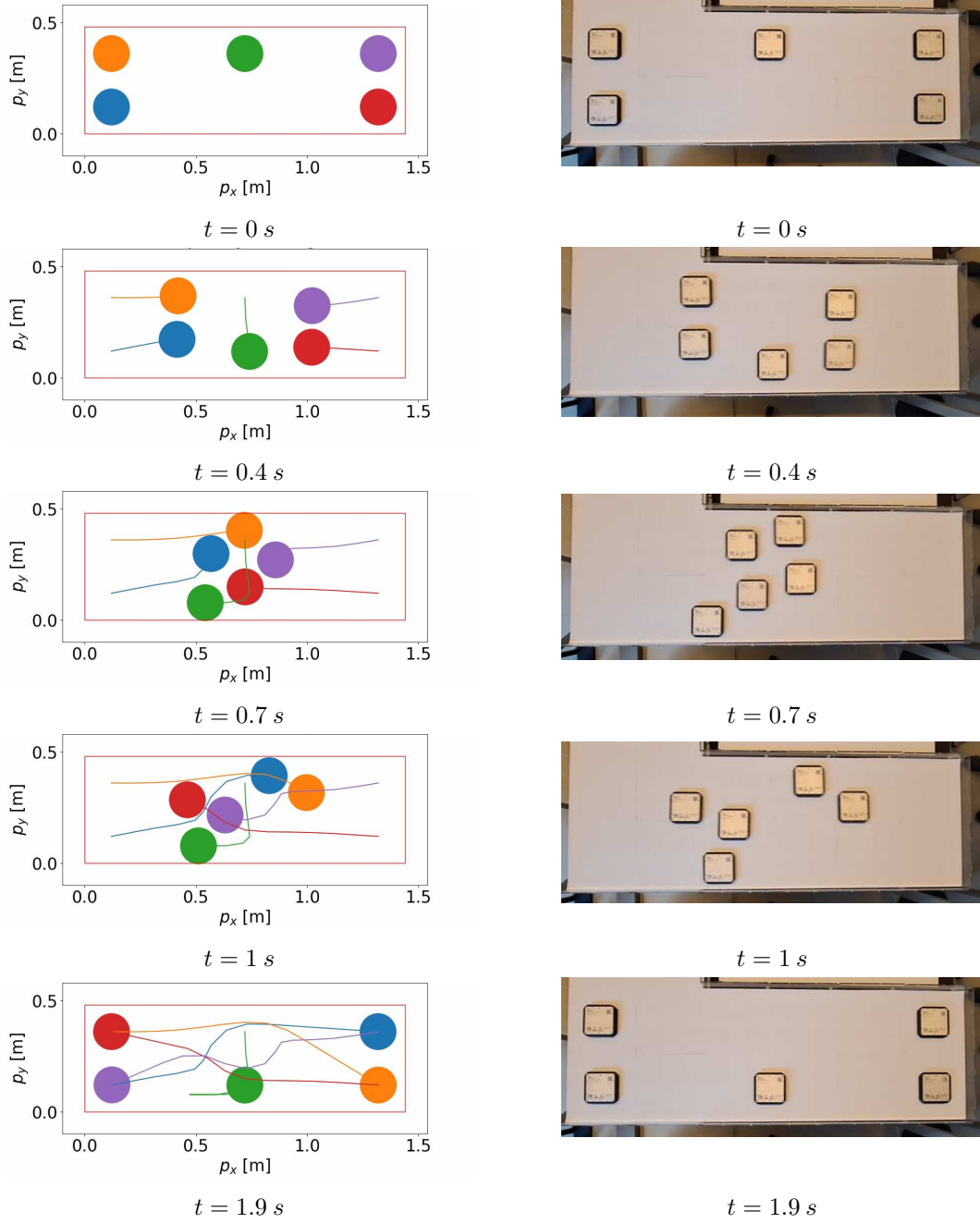


FIGURE 6.8: Comparison between simulation and experiment on the XPlanar, performed for 5 agents with centralised MPC.

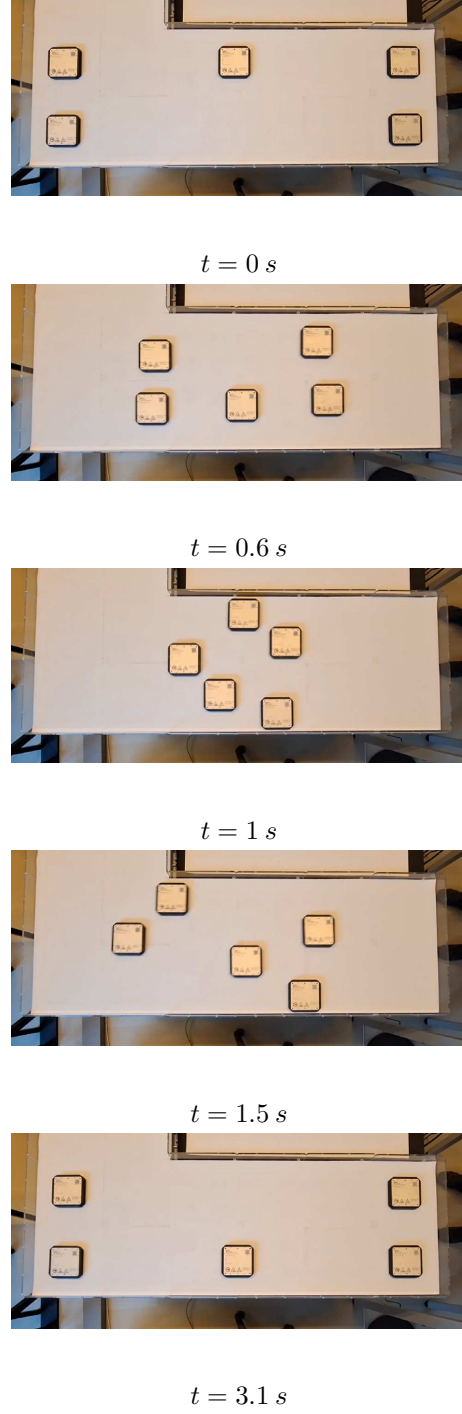
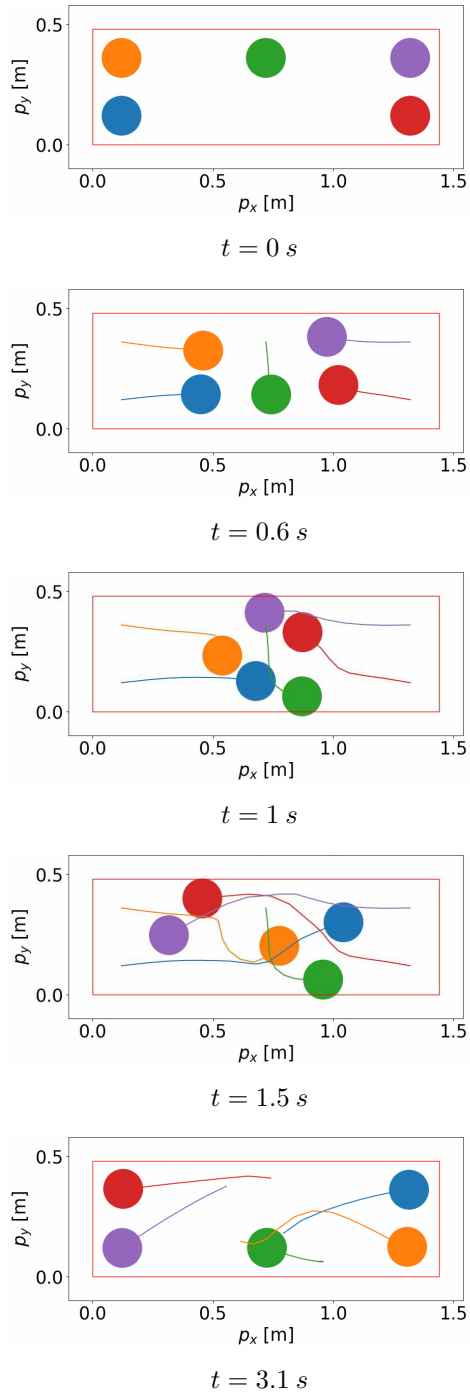


FIGURE 6.9: Comparison between simulation and experiment on the XPlanar, performed for 5 agents with ADMM-CBF performance mode.

6. RESULTS AND DISCUSSION

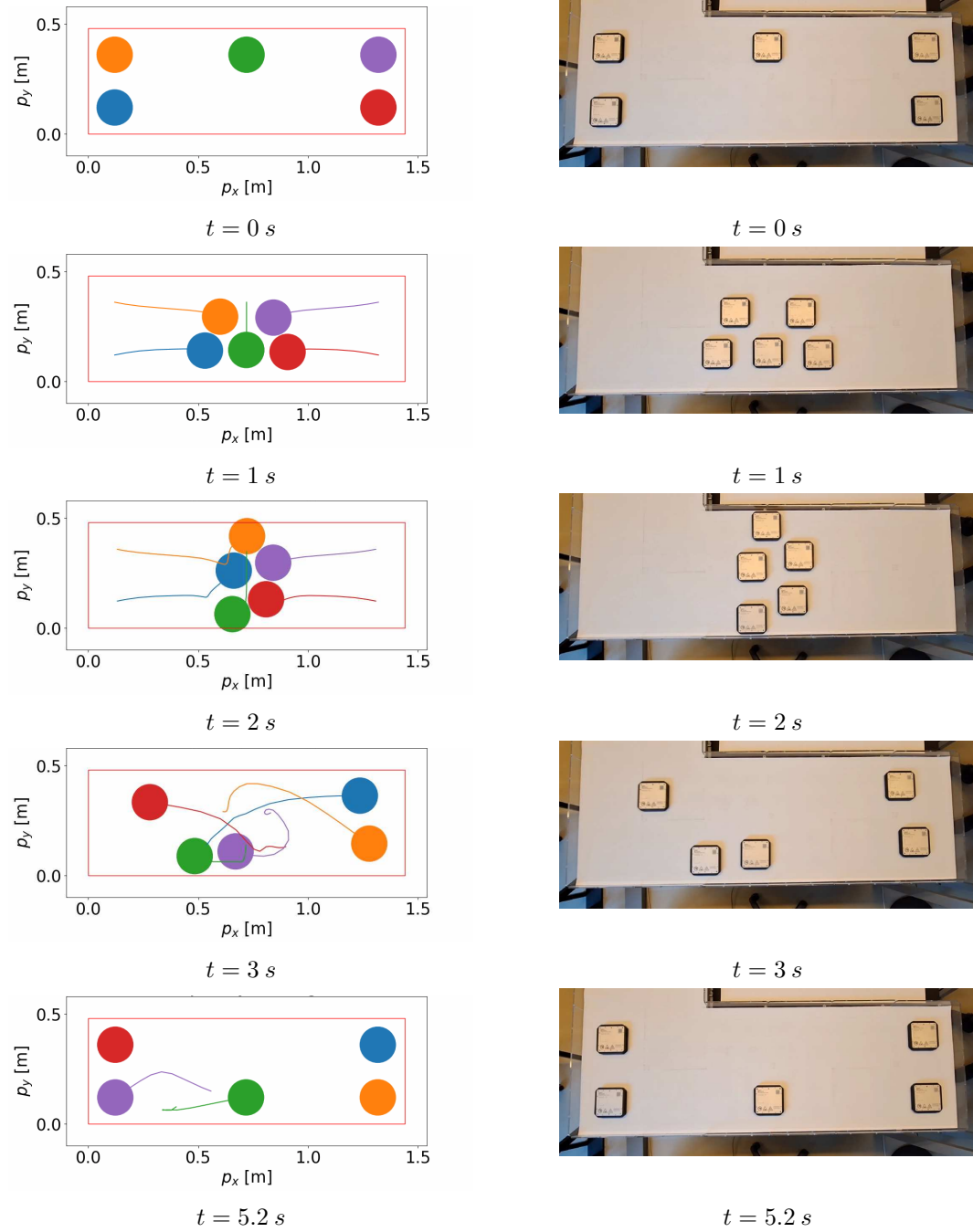


FIGURE 6.10: Comparison between simulation and experiment on the XPlanar, performed for 5 agents with ADMM-CBF fast mode.

Chapter 7

Conclusion

This thesis developed several optimal control based motion planning algorithms and provided implementation guidelines with tunings for each algorithm.

Although the free-time problem cannot be used for real-time purposes, it is useful to consider it as a benchmark for optimality. Centralised MPC, on the other hand, does allow motion planning in real-time. However, due to its scalability $\sim O(N^{2.96})$, we did not continue to pursue a real-time implementation, as we concluded that it is not suitable for multi-robot systems with more than a few agents.

Decentralising the problem using ADMM, still within a sequential scheme, reduces scaling at the cost of optimality and safety. The latter, which is of great importance in practical settings, is successfully recovered by adding the central QCQP 4.11 step using CBFs, resulting in a novel hybrid multi-agent motion planning method that scales with $\sim O(N^{1.82})$. To increase the efficiency and performance of the resulting trajectories, the number of ADMM iterations must often be significantly increased due to its unfavourable convergence properties for this non-convex problem. In these ADMM iterations, tests show that NLP 4.4 is the bottleneck, e.g. for 10 agents it can take up 92.8% of the total computation time.

After the scalability is improved, different steps are taken to accelerate computations. First, FATROP is used to solve OCP 4.3 resulting in a decrease in the computational time of 39.8% and 28.4% for 5 and 10 agents respectively. Secondly, the algorithm is translated from Python to C++, while still using the same serialised CasADi functions. This additionally decreased the computational time with 57.9% and 38.1% for 5 and 10 agents respectively. Lastly, the computation time is initially seen to decrease further with 60.8% for 5 agents by parallelising the implementation using multithreading. However, the influence of multithreading for medium-scale systems is not able to be fully established yet due to a solver mismatch, causing a scalability of $\sim O(N^{4.24})$. This results in a decrease in computational time of only 11.0% for 10 agents. In total, the computational time can be decreased by 90.1% for 5 agents and 60.5% for 10 agents.

ADMM-CBF is investigated using two cases: fast mode ($m = 1$, $\mu = 1$) and performance mode ($m = 20$, $\mu = 40$). From this, the strength of ADMM can be observed, namely that it allows the number of iterations per time step to be fixed, without a loss of safety guarantee. This ensures that the performance of the hybrid method can be linked to the hardware: a stronger computer will be able to do more ADMM iterations, and thus compute more optimal trajectories. The second advantage is that computational time becomes more predictable.

ADMM-CBF entails real-time capability for more agents than centralised MPC as its RCT factor can be lowered by decreasing the number of ADMM iterations, this is especially apparent with fast mode. ADMM-CBF fast mode can perform real-time computations up to 13 agents. This is accomplished using the sequential implementation because for $N = 13$, the scalability of using FATROP to solve the NLP turns out to be detrimental. However, for a lower number of agents, FATROP proves to be faster than IPOPT and even performance mode is able to perform up to 5 agents in real-time.

Finally, the experimental tests on the XPlanar provide a proof of concept for the practical suitability of the hybrid method.

7.1 Future research

Despite considerable improvements, mainly related to safety and scalability, challenges remain w.r.t. implementation and performance. This section provides an overview of the possibilities for future work.

To obtain **more efficient trajectories** without the need for an abundant number of ADMM iterations, there are several promising approaches that are yet to be explored:

- Incorporating ADMM acceleration techniques [15] to obtain similar convergence results with a reduced number of ADMM iterations.
- Researching various non-conservative reformulations of the collision avoidance constraints to obtain more favourable convergence properties.
- Employing a dynamic penalty parameter μ and researching methods to automatically adapt it over the trajectory.
- Employing dynamic HOCBF parameters (K_1 and K_2).

Moving to **real-time performance** in large-scale systems also requires some additional modifications. We provide four suggestions:

- Researching methods to make ADMM compatible with time-varying neighbourhood schemes. Or, finding conditions in which neighbourhoods can be kept constant.
- Incorporating learning techniques to non-safety-critical parts of the algorithm, to further reduce scaling without a loss of the safety guarantee. For example, NLP 4.4 is simultaneously the bottleneck w.r.t. computational cost and is not a safety-critical step since $\mathbf{z}_i$ and $\mathbf{z}_{i,j}$ are not directly applied to the system.
- Finding another more tailored solver for NLP 4.4 that is thread-safe. This would allow us to fully exploit the multithreaded implementation and increase its real-time capability.
- Investigating the limits of reducing the length of the planning horizon when using QCQP 4.11 for safety.

Lastly, the following steps could be taken to extend the algorithm's **practical suitability**:

- Setting up a communication network between a dedicated external computer to perform real-time computations externally and send the trajectories to the XPlanar computer.
- Pair the algorithm with a discrete motion planner for more efficient navigation from corridor to corridor in general multi-rectangular non-convex arenas.
- Using C++ class template barrier [1] for thread synchronisation to limit context switching of the threads.
- Designing a multithread scheme in coordination with a scheduler policy allowing for more efficient parallelisation.
- Designing a multiprocessing scheme which also allows to parallelise the hybrid method, while still using IPOPT.

Bibliography

- [1] C++ class template for `std::barrier`. <https://en.cppreference.com/w/cpp/thread/barrier.html>. Accessed: 02/06/2025.
- [2] Mumps home page. <https://mumps-solver.org/index.php?page=home>. Accessed: 02/06/2025.
- [3] A. D. Ames, S. Coogan, M. Egerstedt, G. Notomista, K. Sreenath, and P. Tabuada. Control barrier functions: Theory and applications. In *2019 18th European control conference (ECC)*, pages 3420–3431. Ieee, 2019.
- [4] J. A. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl. Casadi: a software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11:1–36, 2019.
- [5] B. Automation. Beckhoff information system. Technical report. https://infosys.beckhoff.com/index_en.htm, Accessed: 5/05/2025.
- [6] Beckhoff. Xplanar | planar motor system. <https://www.beckhoff.com/en-en/products/motion/xplanar-planar-motor-system/>, Accessed: 05/05/2025.
- [7] U. Borrmann, L. Wang, A. D. Ames, and M. Egerstedt. Control barrier certificates for safe swarm behavior. *IFAC-PapersOnLine*, 48(27):68–73, 2015.
- [8] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1):1–122, 2011.
- [9] M. Chahine, R. Firoozi, W. Xiao, M. Schwager, and D. Rus. Local non-cooperative games with principled player selection for scalable motion planning. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 880–887. IEEE, 2023.
- [10] J. Chen. A brief tutorial on consensus admm for distributed optimization with applications in robotics. *arXiv preprint arXiv:2410.03753*, 2024.
- [11] J. Choi. Introduction to control lyapunov functions and control barrier functions, 2021. Guest lecture at University of California, San Diego, https://www.youtube.com/watch?v=_Tkn_Hzo4AA.

- [12] W. Decr. Specialised lecture 2: Optimal control. Slides from optimization of mechatronic systems [H04U1C], KU Leuven.
- [13] H. J. Ferreau, C. Kirches, A. Potschka, H. G. Bock, and M. Diehl. qpOases: A parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation*, 6:327–363, 2014.
- [14] J. Gillis, B. Vandewal, G. Pipeleers, and J. Swevers. Effortless modeling of optimal control problems with rokit. In *39th Benelux Meeting on Systems and Control, Date: 2020/03/10-2020/03/12, Location: Elspeet, The Netherlands*, 2020.
- [15] T. Goldstein, B. O’Donoghue, S. Setzer, and R. Baraniuk. Fast alternating direction optimization methods. *SIAM Journal on Imaging Sciences*, 7(3):1588–1623, 2014.
- [16] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.
- [17] C. Jiang and Y. Guo. Incorporating control barrier functions in distributed model predictive control for multirobot coordinated control. *IEEE Transactions on Control of Network Systems*, 11(1):547–557, 2023.
- [18] Y. Koren, J. Borenstein, et al. Potential field methods and their inherent limitations for mobile robot navigation. In *Icra*, volume 2, pages 1398–1404, 1991.
- [19] S. LaValle. Rapidly-exploring random trees: A new tool for path planning. *Research Report 9811*, 1998.
- [20] C. Liu, C.-Y. Lin, and M. Tomizuka. The convex feasible set algorithm for real time optimization in motion planning. *SIAM Journal on Control and optimization*, 56(4):2712–2733, 2018.
- [21] C. E. Luis and A. P. Schoellig. Trajectory generation for multiagent point-to-point transitions via distributed model predictive control. *IEEE Robotics and Automation Letters*, 4(2):375–382, 2019.
- [22] T. Mercy, R. Van Parys, and G. Pipeleers. Spline-based motion planning for autonomous guided vehicles in a dynamic environment. *IEEE Transactions on Control Systems Technology*, 26(6):2182–2189, 2017.
- [23] P. Patrinos. Optimization. Lecture notes from Optimization [H03E3A], KU Leuven.
- [24] Z. Qin, K. Zhang, Y. Chen, J. Chen, and C. Fan. Learning safe multi-agent control with decentralized neural barrier certificates. *arXiv preprint arXiv:2101.05436*, 2021.

-
- [25] J. B. Rawlings, D. Q. Mayne, M. Diehl, et al. *Model predictive control: theory, computation, and design*, volume 2. Nob Hill Publishing Madison, WI, 2017.
 - [26] F. Rey, Z. Pan, A. Hauswirth, and J. Lygeros. Fully decentralized admm for coordination and collision avoidance. In *2018 European Control Conference (ECC)*, pages 825–830. IEEE, 2018.
 - [27] A. Sathya, P. Sopasakis, R. Van Parys, A. Themelis, G. Pipeleers, and P. Patrinos. Embedded nonlinear model predictive control for obstacle avoidance using panoc. In *2018 European Control Conference (ECC)*, pages 1523–1528, 2018.
 - [28] O. Shorinwa, T. Halsted, J. Yu, and M. Schwager. Distributed optimization methods for multi-robot systems: Part 1a tutorial [tutorial]. *IEEE Robotics & Automation Magazine*, 31(3):121–138, 2024.
 - [29] O. Shorinwa, T. Halsted, J. Yu, and M. Schwager. Distributed optimization methods for multi-robot systems: Part 2a survey. *IEEE Robotics & Automation Magazine*, 31(3):154–169, 2024.
 - [30] Z. Tian, Z. Zhang, and R. Jin. Distributed admm for time-varying communication networks. In *2022 IEEE 96th Vehicular Technology Conference (VTC2022-Fall)*, pages 1–5. IEEE, 2022.
 - [31] R. Van Parys and G. Pipeleers. Online distributed motion planning for multi-vehicle systems. In *2016 European Control Conference (ECC)*, pages 1580–1585. IEEE, 2016.
 - [32] R. Van Parys and G. Pipeleers. Distributed mpc for multi-vehicle systems moving in formation. *Robotics and Autonomous Systems*, 97:144–152, 2017.
 - [33] L. Vanroye, A. Sathya, J. De Schutter, and W. Decré. Fatrop: A fast constrained optimal control problem solver for robot trajectory optimization and control. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10036–10043. IEEE, 2023.
 - [34] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106:25–57, 2006.
 - [35] L. Wang, A. D. Ames, and M. Egerstedt. Safety barrier certificates for collisions-free multirobot systems. *IEEE Transactions on Robotics*, 33(3):661–674, 2017.
 - [36] W. Wang, W. Xiao, A. Gonzalez-Garcia, J. Swevers, C. Ratti, and D. Rus. Robust model predictive control with control barrier functions for autonomous surface vessels. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6089–6095. IEEE, 2024.
 - [37] W. Xiao and C. Belta. High-order control barrier functions. *IEEE Transactions on Automatic Control*, 67(7):3655–3662, 2021.

- [38] Y. Yang, X. Guan, Q.-S. Jia, L. Yu, B. Xu, and C. J. Spanos. A survey of admm variants for distributed optimization: Problems, algorithms and features. *arXiv preprint arXiv:2208.03700*, 2022.
- [39] X. Zhang, A. Liniger, and F. Borrelli. Optimization-based collision avoidance. *IEEE Transactions on Control Systems Technology*, 29(3):972–983, 2020.
- [40] H. Zhou and C. Liu. Distributed motion coordination using convex feasible set based model predictive control. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8330–8336. IEEE, 2021.
- [41] V. Zinage, A. Jha, R. Chandra, and E. Bakolas. Decentralized safe and scalable multi-agent control under limited actuation. *arXiv preprint arXiv:2409.09573*, 2024.