Non-linear Model Predictive Control for Aerial-Ground Cooperative Robotics

---

Nicola Lissandrini

Corso di laurea in Ingegneria dell'Automazione

Relatore Prof. Angelo Cenedese
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Padova, Padua, Italy

Correlatore Prof. Dimos Dimarogonas
Department of Automation and Control
KTH Royal Institute of Technology, Stockholm, Sweden

Supervisori Christos Verginis e Pedro Roque

8 Aprile 2019
Padova

KTH Royal Institute of Technology

Università degli Studi di Padova

Corso di Laurea in Ingegneria dell'Automazione

Non-linear Model Predictive Control for Aerial-Ground Cooperative Robotics

Nicola Lissandrini

Reviewers

Prof. Angelo Cenedese
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Padova, Padua, Italy

Prof. Dimos Dimarogonas
Department of Automation and Control
KTH Royal Institute of Technology, Stockholm, Sweden

Supervisors Christos Verginis and Pedro Roque

April 8, 2019

Nicola Lissandrini
Non-linear Model Predictive Control for Aerial-Ground Cooperative Robotics
Master Thesis, April 8, 2019
Reviewers: Prof. Dimos Dimarogonas and Prof. Angelo Cenedese
Supervisors: Christos Verginis and Pedro Roque

KTH Royal Institute of Technology
Department of Automation and Control
School of Electrical Engineering and Computer Science
Brinellvägen 8, 114 28 Stockholm, Sweden
SE-100 44 STOCKHOLM

Università degli Studi di Padova
Corso di laurea in Ingegneria dell'Automazione
Department of Information engineering
Via Giovanni Gradenigo, 6, Padua, Italy
35100

Abstract

Cooperative robotics is a trending topic in nowadays research as it makes possible a number of tasks that cannot be performed by individual robots, such as heavy payloads transportation or challenging manoveurs. In this thesis we address the problem of cooperative transportation by etherogeneous mobile robots. Specifically we consider a generic number of robotic agents simultaneously grasping an object, which is to be transported to a prescribed set point while avoiding obstacles. The procedure is based on a distributed leader-follower structure, where the designated leader agent is responsible of generating a trajectory compatible with its dynamics. The followers must compute a trajectory for their own manipulators that aims at minimizing the internal forces and torques that might be applied to the object by the different grippers.

The Model Predictive Control is well suited to solve such kind of problem for two reasons. First, it provides both an optimal control law and a technique to generate trajectories, that can be shared among the agents. The second reason is that the control comes from an optimization problem, so the solution to the above problem comes naturally with the definition of optimal control problem.

The proposed algorithm is implemented on a system of a ground and aerial robot. The former is composed of an omnidirectional mobile base equipped with a 4-dof manipulator while the latter of an hexarotor with a 2-dof arm mounted beneath the base. The two robots and the control algorithms will be tested with a Gazebo simulation, to which will follow a preliminary phase of experiments with the real robots.

Sommario

La robotica cooperativa è un ambito sempre più centrale nella ricerca scentifica. Con essa, molte attività che risultano impossibili con singoli robot diventano realizzabili, come ad esempio il trasporto di carichi pesanti o l'esecuzione di manovre complesse. In questa tesi viene affrontato il problema del trasporto cooperativo eseguito da robot eterogenei. Nello specifico, si considera un numero generico di agenti che stanno tenendo la presa sull'oggetto che dovrà essere trasportato in una posizione target desiderata evitando la collisione con gli ostacoli. La tecnica si basa su una struttura distribuita "leader-follower", in cui l'agente con ruolo di leader ha il compito di generare una traiettoria compatibile con la sua dinamica e che eviti gli ostacoli. I follower calcolano di conseguenza una traiettoria per i propri manipolatori con l'obiettivo di minimizzare le forze/momenti interni applicati sull'oggetto da parte dei vari agenti.

Il Model Predictive Control si adatta perfettamente a questo tipo di problemi per due ragioni. Come prima cosa, offre contemporaneamente una legge di controllo ottimale e una tecnica per generare traiettorie prive di collisioni, che possono facilmente essere condivise nella rete di agenti. Per secondo, il controllo deriva da un problema di ottimizzazione che dunque può allo stesso tempo risolvere il problema di minimizzazione delle forze assegnato ai follower.

L'algoritmo proposto è implementato su un sistema di robot a terra e aerei. Il primo è composto da una base omnidirezionale dotata di un manipolatore a quattro giunti mentre il secondo è un esacottero che monta un braccio a due gradi di libertà sotto la base. I due robot assieme agli algoritmi di controllo verranno testati in una simulazione Gazebo, alla quale seguirà una fase preliminare di esperimenti su robot reali.

Sammanfattning

Kollaborativ robotik är ett centralt ämne i dagens forskning eftersom den möjliggör en rad arbetsuppgifter som inte kan utföras av individuella robotar, såsom tunga lyft och komplexa manövrar. I den här uppsatsen kommer problemet med kooperativ transport av heterogena mobila robotar att undersökas. Närmare bestämt undersöks ett generiskt antal robotar, som samtliga greppar ett objekt som i sin tur skall transporteras till en bestämd punkt medan de undviker hinder. Tekniken är grundad på en fördelad struktur, en så kallad ”leader-follower”, vari ledarroboten har i uppgift att skapa kompatibel bana med hjälp av sin dynamiska samt undviker hindren. Follower-robotarna måste räkna ut en bana för deras egna ”manipolatori” för att minimera kraften som möjligen appliceras på objektet av de olika gripklorna.

Predictive Control modellen är väl lämpad att lösa dessa problem av två anledningar. För det första är den utrustad med både en ”optimal control law” och en teknisk metod för att framställa banor utan kollisioner, som kan delas mellan agenterna. För det andra kommer kontrollen från ett optimeringsproblem som löses naturligt från minimeringen av kraften när den tilldelas follower-robotarna.

Den förslagna algoritmen är implementerad på ett system av mark- och flygrobotar. Den första består av en ”omnidirectional mobile base” med en ”4-dof manipulator” medan den andra är en ”hexarotor” med en ”2-dof arm” monterad under basen. De två robotarna och kontrolalgoritmen kommer att testas i en Gazebo simulation, vilket kommer följas av en preliminär fas av experiment med riktiga robotar.

Contents

1 Introduction

> *Industry 4.0 is less of a fourth revolution and more of an evolution in many small steps that will truly change how manufacturing and industry does business*
>
> — **Paul Carreiro**
> EMEA at Infor

Automating repetitive task have always been an ambition for human beings. Nowadays robotics is a key factor in industry, as it makes possible to execute repetitive tasks with an efficiency that was impossible before the industrial revolution. However, the human contribution is still essential in the manufacturing processes as robots, until last years, have been always able to just reproduce an action programmed by a person. In this way, the efficiency of production have been able to increase exponentially.

With *Industry 4.0* the world of robotics is coming to a paradigm shift. From tasks of passive reproduction of scheduled movements, robots are becoming able to operate into unknown or partially known environments and smartly adapt to unexpected conditions. This was made possible by an extraordinary development that has been recently realized in all fields of Information Technology: from fast wireless communications to intelligent systems and control algorithms. Optimization, in particular, has a key role in this context: it is the fundamental concept behind machine learning and artificial intelligence techniques, as well as the most advaned control techniques, the nonlinear Model Predictive Control, which will be extensively used in this work.

However, even though those innovations are extremely promising, their implementation in the productive systems is slow as they have an important drawback: they often lack of reliability. Intelligent algorithms, for instance, are based on an heuristic that most of the times provides excellent results but it is impossible to predict how bad are they will behave the times that fail nor the reasons of errors. Furthermore, it is often uncertain *how* the given task will be accomplished. For instance, consider a mobile agent that has to compute a path to overcame an unknown obstacle: even if the algorithm

ensures that the task will be accomplished, it is in general impossible to predict how the agent will achieve the goal, whether to pass on the right or left side. This issue is particularly critical in applications where multiple robots are assigned to collaborate to achieve the same task.

At the same time, multi-agent systems are a decisive ingredient of the new industrial revolution. It is not difficult to imagine applications that must involve more than one agent to be accomplished, like handling heavy payloads or large areas patrolling. Nevertheless, even if a task could be performed by a single agent, it might be more efficient or even cheaper to implement it with multiple, smaller robots.

Summing up, coordination among multiple agents has fundamental importance in the new productive systems. In this thesis, we will exploit the latest technologies of optimization and control to address a particular instance of this problem: the cooperative manipulation. Specifically, the problem will involve etherogeneous robots, ground and flying, trying to take the best from the two worlds: the strenght of ground robots allowing to manipulate heavy payloads and the mobility of flying vehicles that permit to reach configurations that are otherwise impossible.

1.1 State of the art

Multi-agent robotics is a trending topic in the recent research activities. Strong results have been demonstrated on the control of single and multiple flying systems [16, 7]. Recently, a lot of effort have been made to allow phyiscal interaction among these systems and the environment, i.e. aerial manipulation, [12, 8, 24]. Essential is also the development of estimation techniques to allow for robust control, for the vehicle it self as well as the payload [3]. Cooperative transportation among aerial agents, either with cables or with manipulators, have been studed in multiple works [13, 15, 17, 14].

As regards ground and aerial cooperation, [19] provides for fundamental mathematical definitions. Further tests have been made in [23], where the ground vehicle is tasked to deploy the object to a certain position and the aerial vehicle adjusts its inclination, or in [18] where a team of ground robots is used to stabilize the aerial vehicle.

1.2 Thesis Structure

After this introductive chapter which includes a brief presentation of the MPC principles, in Chapter 2, we provide the mathematical formulation and solution to the problem: first, we define the problem settings for a generic number of agents. Then, after introducing the general models for the considered agents to be employed in the MPC formulation, we define the multi-agent algorithm, based on [20] and improved to allow for an online implementation. A robust collision avoidance technique, alternative to that proposed in [20], is then presented.

In chapter 3 we define the differential kinematics models for the case-study implementation of the generic algorithm, involving a ground and an aerial robot. After that, a pure kinematic simulation environment with Matlab/Simulink is provided to test the MPC technique. First the agents end-effector is individually controlled in both position and orientation and then the cooperative algorithm is simulated, along with the collision avoidance technique.

Chapter 3 is about the definitions of the Gazebo simulation environment. After a short introduction to the basic ROS and Gazebo concepts, we report the procedure according to which the models for the robot have been designed in the simulator. Furthermore, the realizion of a Gazebo plugin that provide an easy simulation of the grasp is presented.

Chapter 4 contains one of the major contribution of the work: a flexible tool that allows for the implementation of the MPC control by means of a ROS node capable to deal with any problem definition and dynamical constraints, based on the ACADO code generation tool [9, 1, 10]. First the essential tools from ACADO to define the mathematical problem are outlined. Then the implementation of the MPC tool is presented. The MPC tool is designed to work in two modes: synchronous and asynchronous: the former allows for handling input packets loss by exploiting the MPC predictions, whereas the latter provides a simpler implementation.

In chapter 6 the MPC Wrapper tool is employed in the case study and interfaced with the rest of the ROS network. Also, a robust low-level control technique for the aerial control is provided. After that, the main cooperative

algorithm implementation is presented, that includes a preliminary phase of agents rendez-vous. Finally, the Gazebo simulation results of the entire experiment are reported, along with a preliminary phase of experiments with real robots.

1.3 Model Predictive Control

Model Predictive Control (MPC) is an advanced control technique that can be formulated as the repeated solution of a finite horizon open-loop optimal control problem subject to system dynamics and input and state constraints [6].

The basic concept is to use a dynamic model to predict the system evolution and to compute the control inputs that optimize a given cost function. Formally, consider a generic continuous time system:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) \tag{1.1}$$

Ideally, the optimal control problem is about finding $\boldsymbol{u}(\cdot)$ that minimizes the following cost function defined over an infinite horizon:

$$J_\infty(\boldsymbol{x}, \boldsymbol{u}(\cdot)) = \int_0^\infty V(\boldsymbol{x}(t), \boldsymbol{u}(t))\, \mathrm{d}t \tag{1.2}$$

given the initial condition $\boldsymbol{x}(0) = \boldsymbol{x}_0$. If $\bar{\boldsymbol{u}}(\cdot)$ is a solution to this optimal problem, it is possible to prove that if $V(\cdot)$ is positive definite and both $\boldsymbol{f}$ and V are regular enough, then $\bar{\boldsymbol{u}}(\cdot)$ stabilizes the origin of the system $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \bar{\boldsymbol{u}})$, i.e.:

$$\lim_{t \to \infty} \boldsymbol{x}(t) = 0 \tag{1.3}$$

for initial conditions in a neighborhood of $\boldsymbol{x}_0$. In principle, one may look for a closed solution, but this in general does not exist. On the other hand, this specific kind of problem is not suitable to be solved numerically, because first the solution space $\mathcal{U} = \{\boldsymbol{u}(\cdot)\}$ is infinite dimensional and, second, the time interval of interest $[0, \infty)$ is also infinite. Also, the computed control input is open loop and represent a solution to the exact model defined by $\boldsymbol{f}$, which is always an inexact representation of the actual system. On the contrary, we are looking for a closed loop system that is robust to model errors.

To this aim, we will simplify the problem allowing for an approximated numerical solution. First, the system (1.1) needs to be discretized to a corresponding sampled time system

$$\boldsymbol{x}_{k+1} = \boldsymbol{f}_d(\boldsymbol{x}_k, \boldsymbol{u}_k) \tag{1.4}$$

Then, we restrict the interval of interest to a finite set of samples of length N, referred to as *control horizon*. To address the third issue, at each sample time the cost is defined over the interval from time k to time $k+N$ and the system is initialized with the last measured state $\hat{\boldsymbol{x}}_k$. The optimal problem is then resorted to:

$$\begin{aligned} \min_{\boldsymbol{u}_0,\dots,\boldsymbol{u}_N} \quad & \sum_{k=0}^{N-1} V(\boldsymbol{x}_k, \boldsymbol{u}_k) + V_f(\boldsymbol{x}(N)) \\ \text{subject to:} \quad & \boldsymbol{x}_{k+1} = \boldsymbol{f}_d(\boldsymbol{x}_k, \boldsymbol{u}_k) \\ & \boldsymbol{x}_0 = \hat{\boldsymbol{x}}_0 \end{aligned} \tag{1.5}$$

where the term $V_f(\cdot)$ penalizes the termial state of the considered window is referred to as *terminal cost*, whereas $V(\cdot)$ is called *running cost* in this context. In this way the optimal problem is defined over a finite dimensional set $\boldsymbol{u} \in \mathbb{R}^N$ so that it is suitable to be solved numerically. Also, it is important to point out that each sample time a different control problem is solved, accounting for the last time measurement and thus making the control scheme closed-loop. Most of the solvers are also able to solve constrained problems, allowing to consider state and input constraints of the type:

$$\begin{aligned} \mathcal{U} &= \{\boldsymbol{u} \in \mathbb{R}^m \ : \ \boldsymbol{u}_{min} \leq \boldsymbol{u} \leq \boldsymbol{u}_{max}\} \\ \mathcal{X} &= \{\boldsymbol{x} \in \mathbb{R}^n \ : \ \boldsymbol{x}_{min} \leq \boldsymbol{x} \leq \boldsymbol{x}_{max}\} \end{aligned} \tag{1.6}$$

Summarizing, the MPC scheme works as follows:

1. Obtain state estimate
2. Compute the optimal input by minimizing the cost function
3. Apply the first part of the optimal input until the next sample time

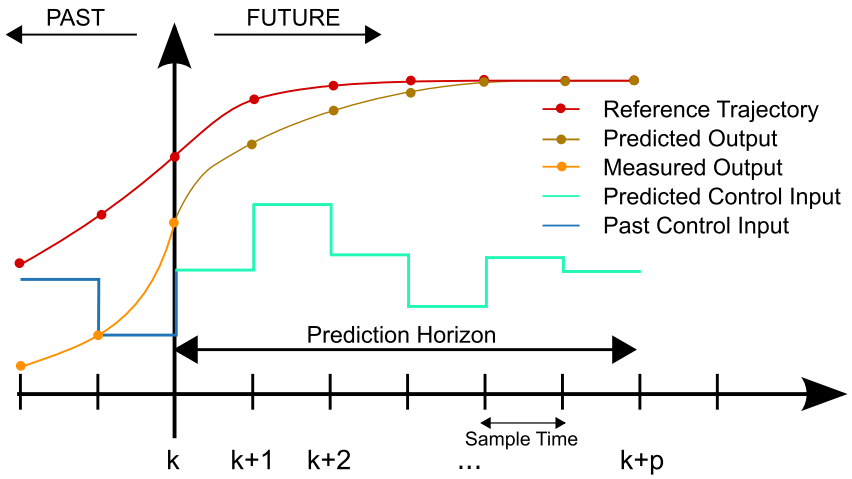


Figure 1.1: Principle of MPC, [2]

2 The Cooperative Manipulation Algorithm

In this section we will provide a mathematical description of the considered setup. First, we define the general setting with general assumptions regarding the number and the geometry of each robot. Next, we will introduce a geometric control technique of the end-effector on $\mathbb{SE}(3)$ based on the theory of nonlinear model-predictive control. Finally, we derive the multi-agent algorithm, along with a technique which exploits the power of MPC to perform collision avoidance.

2.1 Problem Formulation

In this section we consider a generic setup where N robotic agents are grasping the same object. The robots are composed of a mobile base and a manipulator, which can both have an arbitrary number of degrees of freedom. The base can be either a ground vehicle, e.g. fully actuated with holonomic wheels with 2 d.o.f., or a UAV. The aim of the agents is to transport the object along a reference trajectory minimizing the internal forces and avoiding obstacles.

In the following, we denote by $\boldsymbol{p}_{v,i},\ \boldsymbol{p}_{e,i} \in \mathbb{R}^3$ the position of the base and of the end effector, while. Let $\boldsymbol{q}_i \in \mathbb{R}^{n_i}$ be the vector of joints variables describing the configuration of each manipulator, where n_i is the corresponding number of degrees of freedom. We assume to be able to supply velocity inputs to the system, and then the kinematics of each end effector, which is the objective of the control algorithm, is described by the following non-linear system:

$$\text{Agent } i\text{:} \quad \begin{cases} \dot{\boldsymbol{p}}_{e,i} &= \boldsymbol{A}_p \boldsymbol{u}_{v,i} + \boldsymbol{J}_{P,i}(\boldsymbol{q}_i)\boldsymbol{u}_{q,i} \\ \boldsymbol{\omega}_{e,i} &= \boldsymbol{A}_\omega \boldsymbol{u}_{\omega,i} + \boldsymbol{J}_{O,i}(\boldsymbol{q}_i)\boldsymbol{u}_{q,i} \\ \dot{\boldsymbol{R}}_{e,i} &= \boldsymbol{S}(\boldsymbol{\omega}_{e,i})\boldsymbol{R}_{e,i} \\ \dot{\boldsymbol{q}}_i &= \boldsymbol{u}_{q,i} \\ \dot{\boldsymbol{p}}_{v,i} &= \boldsymbol{A}_p \boldsymbol{u}_{v,i} \\ \dot{\boldsymbol{R}}_{v,i} &= \boldsymbol{S}(\boldsymbol{A}_\omega \boldsymbol{u}_{\omega,i})\boldsymbol{R}_{v,i} \end{cases} \tag{2.1}$$

where:

- $\boldsymbol{p}_{v,i},\ \boldsymbol{p}_{e,i} \in \mathbb{R}^3$ are the position of the base and the end-effector of the i-th agent, respectively.
- $\boldsymbol{R}_{v,i},\ \boldsymbol{R}_{e_i} \in \mathbb{SO}(3)$ refer to the corresponding rotation matrices relative to a fixed world frame $\{W\}$.
- $\boldsymbol{u}_{v,i},\ \boldsymbol{u}_{\omega,i} \in \mathbb{R}^3$ are the linear and angular input velocities applied at the base.
- $\boldsymbol{u}_{q,i}$ are the joint velocities, which we assume to be able to control with zero error.
- $\boldsymbol{A}_p,\ \boldsymbol{A}_\omega \in \mathbb{R}^{3\times3}$ allow to model constraints on the input velocity (e.g. reference frame transform or anholonomic constraints)
- $\boldsymbol{\omega}_{e,i}$ is the angular velocity of the end effector in the world frame.
- $\boldsymbol{J}_{P,i},\ \boldsymbol{J}_{O,i} \in \mathbb{R}^{3\times n_i}$ are, respectively, the position and orientation Jacobian matrices, which depend on the structure of the manipulator.

Finally, for compactness of notation, we define the full state and input:

$$\boldsymbol{x}_i = \begin{bmatrix} \boldsymbol{p}_{e,i} & \boldsymbol{r}_{e,i} & \boldsymbol{p}_{v,i} & \boldsymbol{q}_i \end{bmatrix}^\top \in \mathbb{R}^{n_i}, \quad \boldsymbol{u}_i = \begin{bmatrix} \boldsymbol{u}_{q,i} & \boldsymbol{u}_{v,i} & \boldsymbol{u}_{\omega,i} \end{bmatrix}^\top \in \mathbb{R}^{p_i} \tag{2.2}$$

where the lower case $\boldsymbol{r}_{e,i}$ refers to a column-major vector representation of the rotation matrix $\boldsymbol{R}_{e,i}$, s.t. $[\boldsymbol{R}_{e,i}]_{(h,k)} = [\boldsymbol{r}_{e,i}]_{(3k+h)}$. We can then rewrite (2.1) with the compact notation:

$$\dot{\boldsymbol{x}}_i = \mathcal{A}_i(\boldsymbol{x}_i)\boldsymbol{u}_i \tag{2.3}$$

where $\mathcal{A}_i(\boldsymbol{x}_i)$ collects all the control-affine terms and can be easily inferred by the equation. Then, the tracking error is defined as:

$$\boldsymbol{e}_i = \beta_p \left\| \boldsymbol{p}_{e,i} - \boldsymbol{p}_{e,i,des} \right\|^2 + \beta_o \, \mathrm{d}^2_{\mathbb{SO}(3)}(\boldsymbol{R}_{e,i}, \boldsymbol{R}_{e,i,des}) \tag{2.4}$$

where $\mathrm{d}(\cdot,\cdot) : \mathbb{SO}(3) \times \mathbb{SO}(3) \to \mathbb{R}^+$ is a suitable distance in $\mathbb{SO}(3)$ and $\beta_o,\ \beta_p$ are weight factors that we may want to consider.

In this work we address two problems: the first is to derive a control law to track a reference trajectory lying in $\mathbb{SE}(3)$ for each agent separately. The second is to define an algorithm that allows multi-agent coordination minimizing

the internal forces applied on the object. We provide a formal definition in the following:

Problem 2.1 For each agent, find a feedback control law $\boldsymbol{u}_i = \boldsymbol{f}_i(\boldsymbol{x}_i)$ such that the tracking error dynamics is locally asymptotically stable:

$$\lim_{t\to\infty} \boldsymbol{e}_i(t) = 0, \qquad \text{for } \boldsymbol{e}_i(0) \in \mathscr{R}_i \tag{2.5}$$

with $\mathscr{R}_i \subseteq \mathbb{R}^{n_i}$ an open set representing the region of attraction.

Assuming to be able to control $\boldsymbol{T}_{e,i} = (\boldsymbol{p}_{e,i}, \boldsymbol{R}_{e,i}) \in \mathcal{M}_i \subseteq \mathbb{SE}(3)$, where $\mathcal{M}_i$ is the reachable submanifold of $\mathbb{SE}(3)$ according to the robot structure, we define the following coordination problem.

Problem 2.2 Given a reference trajectory for the object:

$$\{\boldsymbol{\xi}_{o,ref}(t) : \mathbb{R} \to \mathbb{SE}(3) : t \mapsto \boldsymbol{\xi}(t) = \left(\boldsymbol{p}_{o,ref}(t), \boldsymbol{R}_{o,ref}(t)\right),\ t > 0\}, \tag{2.6}$$

define a multi-agent algorithm, based on full and constant communication graph with reliable transmission, that tracks such trajectory and it is such that each end-effector maintain a pose relative to the object with minimum error, i.e. the following quantity:

$$\mathcal{E}(t) = \sum_{i=0}^{N-1} \mathrm{d}^2_{\mathbb{SE}(3)} \left(\boldsymbol{T}_{e,i}(t)\hat{\boldsymbol{T}}_o^{e,i}(t), \boldsymbol{\xi}_o(t)\right), \tag{2.7}$$

with the usual products defined on the group $\mathbb{SE}(3)$ and:

$$\mathrm{d}^2_{\mathbb{SE}(3)}[(\boldsymbol{p}_1, \boldsymbol{R}_1), (\boldsymbol{p}_2, \boldsymbol{R}_2)] = \beta_p \|\boldsymbol{p}_1 - \boldsymbol{p}_2\|^2 + \beta_o \, \mathrm{d}^2_{\mathbb{SO}(3)}(\boldsymbol{R}_1, \boldsymbol{R}_2) \tag{2.8}$$

and where $\boldsymbol{\xi}_o(t)$ is the actual path tracked by the object and $\hat{\boldsymbol{T}}_o^{e,i}(t) \in \mathbb{SE}(3)$ is the estimated relative transform from each agent's end-effector to the object.

A few remarks on the latter problem:

- At $t = 0$ we assume that the objects and the agents are still, and we are able to know all the initial relative positions $\hat{\boldsymbol{T}}_o^{e,i}(0)$. In this configuration $\hat{\boldsymbol{T}}_o^{e,i}(0)$ expresses exactly the physical transformation between the gripper and the object.

- By this assumption, it holds:

$$\hat{\boldsymbol{T}}_o^{e,i}(0) = \boldsymbol{T}_o^{e,i} \tag{2.9}$$

- For $t > 0$, $\hat{\boldsymbol{T}}_o^{e,i}(t)$ must be modelled in an appropriate way. As an example, if we assume that the grasp from each agent is rigid, then the relative position is constant $\hat{\boldsymbol{T}}_o^{e,i} = \hat{\boldsymbol{T}}_o^{e,i}(t) = \hat{\boldsymbol{T}}_o^{e,i}(0), \ \ \forall t \in \mathbb{R}^+$. However, this assumption is critical and strongly depends on the implementation of the system.

- The actual quantities we're able to control are not $(\boldsymbol{p}_o, \boldsymbol{R}_o)$ but rather $(\boldsymbol{p}_{e,i}, \boldsymbol{R}_{e,i})$, so for each agent we shall consider a reference trajectory transformed in its reference frame, as follows:

$$\boldsymbol{\xi}_{e,i}(t) = \boldsymbol{\xi}_o(t) \left(\hat{\boldsymbol{T}}_o^{e,i}(t)\right)^{-1} \tag{2.10}$$

2.2 MPC for individual geometric control

In this section we setup the MPC controller implemented with the ACADO library, which solves discrete time nonlinear model predictive control of the following form:

$$\begin{aligned}
\min_{\substack{\boldsymbol{x}_0, \dots, \boldsymbol{x}_N \\ \boldsymbol{u}_0, \dots, \boldsymbol{u}_{N-1}}} \quad & \sum_{k=0}^{N-1} \left\| \boldsymbol{h}(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{d}_k) - \boldsymbol{y}_{ref,k} \right\|_{\boldsymbol{W}_k}^2 + \left\| \boldsymbol{h}_N(\boldsymbol{x}_N) - \boldsymbol{y}_{ref,N} \right\|_{\boldsymbol{W}_N}^2 \\
\text{subject to:} \quad & \boldsymbol{x}_0 = \hat{\boldsymbol{x}}_0 \\
& \boldsymbol{x}_{k+1} = \boldsymbol{F}(\boldsymbol{x}_k, \boldsymbol{u}_k), \quad k = 0, \dots, N-1 \\
& \boldsymbol{x}_{k,min} \leq \boldsymbol{x}_k \leq \boldsymbol{x}_{k,max}, \quad k = 0, \dots, N \\
& \boldsymbol{u}_{k,min} \leq \boldsymbol{u}_k \leq \boldsymbol{u}_{k,max}, \quad k = 0, \dots, N
\end{aligned} \tag{2.11}$$

where: N is the number of samples of the prediction horizon, $\boldsymbol{x}_k \in \mathbb{R}^{n_x}$ denotes the state vector, $\boldsymbol{u}_k \in \mathbb{R}^{n_u}$ the control input, $\boldsymbol{d}_k \in \mathbb{R}^{n_{od}}$ representes a general purpose online data, $\boldsymbol{F}(\cdot, \cdot)$ is a discretized version of the continuous time model of the case, $\hat{\boldsymbol{x}}_0$ is the currently measured state and $\boldsymbol{h}(\cdot, \cdot, \cdot) \in \mathbb{R}^{n_y}$ is the output function of the system we want to consider. Note that all the quantities involved have to be defined over the whole time window.

Definition of the Cost function

The most critical element to consider in (2.11) is the output function $\boldsymbol{h}(\cdot,\cdot,\cdot)$ along with $\boldsymbol{y}_{ref}$. Recalling that the goal is to control the position and orientation, we want to find the minimum norm linear error between the current and the goal position. We recall the definition of state and goal trajectory:

$$\begin{aligned}\boldsymbol{x}_i(t) &= \begin{bmatrix}\boldsymbol{p}_{e,i} & \boldsymbol{r}_{e,i} & \boldsymbol{p}_{v,i} & \boldsymbol{q}_i\end{bmatrix}^\top \in \mathbb{R}^{n_x} \\ \boldsymbol{\xi}_i(t) &= (\boldsymbol{p}_{e,des}(t), \boldsymbol{R}_{e,des}(t)) \in \mathbb{SE}(3)\end{aligned} \tag{2.12}$$

In the following, we will drop the index i, as everything is valid for both agents in the exact same way. Also, we pass straight to sampled time quantities. The trajectory must be sampled and considered within the prediction horizon, i.e. $\boldsymbol{\xi}_k = \boldsymbol{\xi}(t_0 + kT_s)$, $k = 1,\dots,N$. The first part of the output function is straightforward to model:

$$\boldsymbol{h}_{1:3}(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{d}_k) = \boldsymbol{p}_e(k) - \boldsymbol{p}_{e,des}(k) \tag{2.13}$$

To define the orientational error, we must choose a proper error metrics on $\mathbb{SO}(3)$. One may be tempted to consider the Riemaniann geodesic metrics:

$$\mathrm{d}_R^2(\boldsymbol{R}_e, \boldsymbol{R}_{des}) := \left\|\log(\boldsymbol{R}_e^\top \boldsymbol{R}_{des})\right\|_F^2 \tag{2.14}$$

where $\|\cdot\|_F$ denotes the Frobenius matrix norm. However, computing the matrix logarithm is not straightforward and thence increments the calculus load for each computation step. Also, the optimiziation is numerical, so we are not exploiting the smoothness properties that comes from working in the Riemaniann tangent manifold. We then consider the first order Taylor expansion of the matrix logarithm, which also has the meaning of deviation of the rotation difference from identity:

$$\mathrm{d}_I^2(\boldsymbol{R}_e, \boldsymbol{R}_{des}) := \|\boldsymbol{R}_e^\top \boldsymbol{R}_{des} - \boldsymbol{I}\|_F^2 \tag{2.15}$$

Since this relation cannot be expressed as a regular difference, we employ the online data vector to represent the desired rotation matrix:

$$[\boldsymbol{d}_k]_{3k+h} = [\boldsymbol{R}_{des}]_{h,k} \tag{2.16}$$

and, if we denote the 3×3 corresponding matrix $\boldsymbol{D}_k$

$$\begin{aligned} \boldsymbol{h}_4(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{d}_k) &= \sum_{h=1}^{3}\sum_{k=1}^{3}\left[\boldsymbol{R}_{e,k}^{\top}\boldsymbol{D}_k - \boldsymbol{I}\right]_{h,k}^{2} \\ [\boldsymbol{y}_{k,ref}]_4 &= 0 \end{aligned} \tag{2.17}$$

Then, the remaining components of $\boldsymbol{h}$ account for the minimization of the input energy, which grants the stability of the controller:

$$\begin{aligned} \boldsymbol{h}_{5:5+p}(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{d}_k) &= \boldsymbol{u}_k \\ [\boldsymbol{y}_{k,ref}]_{5:5+p} &= \boldsymbol{0}_p \end{aligned} \tag{2.18}$$

where p is the number of the control inputs for the specific robot.

2.3 MPC for cooperative manipulation: a robust approach

In the last section we solved the first of the two problems. Here we address the latter, whose goal is to control the object to a prescribed pose in a cooperative way among multiple agents. This could be solved in a centralized way [20] by deriving a coupled model of the agents. This would need to make strong assumptions about the rigidity since the coupled MPC needs the full model to make predictions. Here we present a technique which has two advantages:

- It is decentralized: each agent computes its own inputs to control the end effector with separate MPCs. This allows for lighter models, which would become exponentially more complex as the number of agents increases, and it is naturally suited for distributed computing.
- It does not require strong assumption on the rigidity of the grasps: as we will see, we want to minimize the internal forces and torques but we cannot require that they are exactly zero. We will rather achieve this goal by defining a minimization problem. In this way, even if there is small difference in the grasps w.r.t. the rest condition the algorithm will converge to the minimum displacement physically realizable.

The proposed is a leader-follower strategy in which one agent, that takes the role of the leader is designed to generate the trajectory all the other agents will follow. Recall the definitions:

- $\boldsymbol{T}_{e,i}$, $\boldsymbol{T}_o$ refer to the pose of the i-th end effector and the object, respectively.
- $\boldsymbol{T}_o^{e,i}(t)$ is the actual relative transform from the end-effector of agent i and the object pose at time t.
- $\hat{\boldsymbol{T}}_o^{e,i}(t)$ is the relative pose of the object w.r.t. the end-effector as expected from agent i, with the property:

$$\hat{\boldsymbol{T}}_o^{e,i}(0) = \boldsymbol{T}_o^{e,i} \tag{2.19}$$

We will also refer to the leader agent with the index $\ell \in [1, \dots, N]$, and the other followers with $j \neq \ell$. Specifically, the algorithm is schematized as follows:

1. **Handshake**

   Before the cooperative task is started, all agents perform a handshake in which they share information regarding the relative pose of each end effector w.r.t. the object. Based on (2.19), we can retrieve the the relative transform from the object to each agent.

$$\hat{\boldsymbol{T}}_o^{e,i}(0) = \boldsymbol{T}_o \left(\boldsymbol{T}_{e,i}\right)^{-1} \tag{2.20}$$

2. **Transform estimation**

   The trajectory of the object $\boldsymbol{\xi}_o(t) \in \mathbb{SE}(3)$ must be converted to a trajectory of the leader end effector. The conversion is performed via the estimates $\hat{\boldsymbol{T}}_o^{e,\ell}(t)$. It is now needed to make some assumptions on that transform: we don't want to assume rigidity assumption since it is unrealistic and does not allow the rolling-pitching rotations of the flying agents. However, we will assume that the displacement from the initial position is bounded in norm, i.e.:

$$\mathrm{d}_{\mathbb{SE}(3)} \left(\hat{\boldsymbol{T}}_o^{e,i}(t), \hat{\boldsymbol{T}}_o^{e,i}(0)\right) \leq \kappa_i \tag{2.21}$$

   If κ_i are small, we assume that the internal forces and torques are correspondingly small and below a tolerance threshold, although we cannot model this relation explicitly as the models account only for the first order kineamtic. This is certainly the case when the overall dynamics is slow. In this case, in particular the rolling and pitching

movements of the drones (which, we recall, are not taken in to account in the model) produce internal torques that are tolerable. In view of (2.21) the best estimate $\hat{\boldsymbol{T}}_o^{e,i}(t)$ is:

$$\hat{\boldsymbol{T}}_o^{e,i}(t) = \hat{\boldsymbol{T}}_o^{e,i}(0), \quad \text{for } t > 0, \quad i1, \dots, N \tag{2.22}$$

We will then drop the dependance of t.

3. **Trajectory conversion**

   Under those assumption, the desired trajectory for the object, which must be converted into the corresponding desired trajectory for the end effector, is given by:

$$\boldsymbol{\xi}_{e,\ell}(t) = \boldsymbol{\xi}_o(t) \left(\hat{\boldsymbol{T}}_o^{e,\ell}\right)^{-1} \tag{2.23}$$

4. **Leader control and prediction**

   For each time step, the leader will exploit the MPC to track the trajectory. Along the control input $\boldsymbol{u}_\ell(t_k)$, it will compute the corresponding predicted trajectory within the time windw NT_s:

$$\{\hat{\boldsymbol{\xi}}_{e,\ell}(t_k),\ t_k = kT_s,\ k = 0, \dots, N\} \tag{2.24}$$

5. **Follower tracking**

   The followers must now guarantee that the displacement error, defined as in (2.21), is minimized. First, the predicted trajectory is converted to the trajectory of the corresoponding follower end effector, which under the considered assumpitons can be derived as:

$$\begin{aligned} \boldsymbol{\xi}_{e,j}(t) &= \hat{\boldsymbol{\xi}}_{e,\ell}(t) \hat{\boldsymbol{T}}_{e,j}^{e,\ell} \\ &= \hat{\boldsymbol{\xi}}_{e,\ell}(t) \hat{\boldsymbol{T}}_o^{e,\ell} \left(\hat{\boldsymbol{T}}_o^{e,j}\right)^{-1} \end{aligned} \tag{2.25}$$

   Note that, because of (2.9), it holds:

$$\boldsymbol{T}_{e,j}(0) = \boldsymbol{\xi}_{e,j}(0) \tag{2.26}$$

One first solution, proposed in [20], is to force (2.26) to hold over the whole prediction window with an additional constraint on the MPC formulation of the agents, which would then be of the form, in a continuous time formulation:

$$\begin{aligned}
\min_{\boldsymbol{x}_j(\cdot),\,\boldsymbol{u}_j(\cdot)} & \int_0^T \boldsymbol{x}_j(t)^\top \boldsymbol{W}_{\boldsymbol{x}} \boldsymbol{x}_j(t) + \boldsymbol{u}_j(t)^\top \boldsymbol{W}_{\boldsymbol{u}} \boldsymbol{u}_j(t)\,\mathrm{d}t \\
& + \boldsymbol{x}_j^\top(T) \boldsymbol{W}_T \boldsymbol{x}_j(T) \\
\text{subject to: } & \boldsymbol{x}_j(0) = \hat{\boldsymbol{x}}_j(0) \\
& \dot{\boldsymbol{x}}_j = \mathcal{A}_j(\boldsymbol{x}_j)\boldsymbol{u}_j \\
& \vdots \\
& (\boldsymbol{p}_{e,j}(t), \boldsymbol{R}_{e,j}(t)) = \boldsymbol{\xi}_{e,j}(t), \quad \forall t \in [0, T]
\end{aligned} \tag{2.27}$$

The problem with the last constraint is mathematically feasible in the continuous formulation due to (2.26). However, in the discrete time implementation it must be converted into a small threshold, for each component. However, this has several problems:

- The threshold should be large enough to allow errors due to noisy measurement and not perfect tracking of the low-level controller, but a larger threshold entails worse performances in terms of displacement errors and the consequent internal forces/torques.

- Furthermore, the behavior within the threshold is defined by the cost function, which is necessary to grant stability but it does not aim at minimizing the error and, in general, is unclear. As instance, in many experiments the error turned out to tend to the boundary of the tolerance region. Although this is not proven to happen anytimes, this is certainly an undesirable behavior.

- On the other hand a small threshold may result in exponentially high computational load, which, for real time applications, cannot be accepted.

- In the case that the error is higher than the threshold, the dynamics is also undefined. This could happen not only due to possible phyisical disturbances, in which case the optimization problem is unsolvable, but even if it is not the case the solver may happen to find a suboptimal solution which does not meet that constraint. In such circumstance, which happens expecially in a limited-time

context, the result is unpredictable and in many experiments the produced solution had a really high error.

To overcome these issues, we will formulate the following task as a minimization problem, rather than a constraint. The leader MPC, in the previous step, predicted the trajectory $\hat{\boldsymbol{\xi}}_o(t) = \hat{\boldsymbol{\xi}}_{e,\ell}(t)\hat{\boldsymbol{T}}_o^{e,\ell}$ that approach the reference $\boldsymbol{\xi}_{o,ref}(t)$ according to its dynamic constraints and weights. Then, according to the predicted trajectory, the cooperative task is accomplished by minimizing

$$\begin{aligned}\hat{\mathcal{E}}(t) &= \sum_{i=0}^{N-1} \mathrm{d}^2_{\mathbb{SE}(3)}\left(\boldsymbol{T}_{e,i}(t)\hat{\boldsymbol{T}}_o^{e,i}(t), \hat{\boldsymbol{\xi}}_o(t)\right) \\ &= \mathrm{d}^2_{\mathbb{SE}(3)}\left(\boldsymbol{T}_{e,\ell}(t)\hat{\boldsymbol{T}}_o^{e,\ell}(t), \hat{\boldsymbol{\xi}}_o(t)\right) \\ &+ \sum_{j\neq\ell} \mathrm{d}^2_{\mathbb{SE}(3)}\left(\boldsymbol{T}_{e,i}(t)\hat{\boldsymbol{T}}_o^{e,i}(t), \hat{\boldsymbol{\xi}}_o(t)\right)\end{aligned} \tag{2.28}$$

Under the assumption $\mathrm{d}_{\mathbb{SE}(3)}(\boldsymbol{T}_o^{e,\ell}(t), \boldsymbol{T}_o^{e,\ell}) \leq \kappa_\ell$, the first term is a positive quantity smaller than κ_ℓ, so we will find a suboptimal solution by individually minimizing the terms in the sum:

$$\mathrm{d}^2_{\mathbb{SE}(3)}\left(\boldsymbol{T}_{e,i}(t)\hat{\boldsymbol{T}}_o^{e,i}, \hat{\boldsymbol{\xi}}_o(t)\right) \tag{2.29}$$

in which we exploited again the bound given by (2.21). By suboptimal we mean that the solution found may differ from the optimal by a quantity proportional to a linear combination of κ_i. We can rewrite:

$$\begin{aligned}\mathrm{d}^2_{\mathbb{SE}(3)}\left(\boldsymbol{T}_{e,i}(t)\hat{\boldsymbol{T}}_o^{e,i}, \hat{\boldsymbol{\xi}}_o(t)\right) &= \mathrm{d}^2_{\mathbb{SE}(3)}\left(\boldsymbol{T}_{e,i}(t), \hat{\boldsymbol{\xi}}_o(t)\left(\hat{\boldsymbol{T}}_o^{e,i}\right)^{-1}\right) \\ &= \mathrm{d}^2_{\mathbb{SE}(3)}\left(\boldsymbol{T}_{e,i}(t), \hat{\boldsymbol{\xi}}_{e,\ell}(t)\left(\hat{\boldsymbol{T}}_o^{e,i}\right)^{-1}\right)\end{aligned} \tag{2.30}$$

Then, the task is formally achieved by finding control input for the followers that minimzes the total error over the whole prediction window, i.e.:

$$\begin{aligned}&\min_{\boldsymbol{x}_j(\cdot),\ j\neq i} \int_t^{t+T} \sum_{j\neq i} \mathrm{d}^2_{\mathbb{SE}(3)}\left(\boldsymbol{T}_{e,i}(t), \hat{\boldsymbol{\xi}}_{e,\ell}(t)\hat{\boldsymbol{T}}_o^{e,\ell}\hat{\boldsymbol{T}}_{e,j}^{e,\ell}\right) \mathrm{d}t \\ &\text{subject to: } \dot{\boldsymbol{x}}_j = \mathcal{A}_j(\boldsymbol{x}_j)\boldsymbol{u}_j, \quad \forall j \neq i\end{aligned} \tag{2.31}$$

By the linearity of the integrator it is easy to show that it is equivalent to solve the minimization problems separately for each agent. By the definition of $\mathrm{d}_{\mathbb{SE}(3)}$ we considered, it is immediate to see that this is an optimal control problem whose discretized version can be solved each

time step via the MPC controller we introduced in the previous section (2.11) for each agent, once defined the reference trajectory in (2.12) as:

$$\boldsymbol{\xi}_j(t) = \boldsymbol{\xi}_{e,\ell}(t)\hat{\boldsymbol{T}}_{e,j}^{e,\ell} \tag{2.32}$$

6. **Control actuation**

   Every agent wait until all of them have finished the computations to apply synchronously the control input $\boldsymbol{u}_i(t)$. In this way, for real time applications it is needed that the

$$T_s > T_{c,\ell} + \max_{j \neq \ell} T_{c,j} \tag{2.33}$$

   where $T_{c,i}$ is the computation time of each agent.

Remarks The result from (2.31) that the problem is solvable separately is not trivial and proves that the problem can be actually solved with a distributed scheme. The solution provided is also inherently robust: suppose that all the correct initial transformations $\hat{\boldsymbol{T}}_o^{e,i}$ are known. Then the convergence to the minimum error $\mathcal{E}(t)$ is ensured by the MPC for any initial configuration $\boldsymbol{T}_{e,i}(t_0)$, rather than for those into the small threshold above mentioned. This is important especially when the robotic system is etherogeneous and composed by robots with different dynamical properties, like those, in particular, composed by underactuated flying vehicles, which non-negligible errors in the low-level controller are structural.

2.4 Control strategy for obstacle avoidance

In most practical cases agents are moving in presence of obstacles, so we may require that the generated trajectory is such to avoid collision among agents with obstacles and agents with each other. By modifying the model in the MPC formulation, as shown in this section, it is easy to implement such feature. However, the individual nature of the MPC in the distributed algorithm, makes some issues arise. Assume that the leader generates a trajectory which avoid itself to collide with the obstacles. It is possible that

such path leads to unavoidable collision for some followers, since it has no information about the dynamics of the followers and it cannot make prediction on them. Designing an algorithm that accounts for the collision of all the agents requires a structural revision of the whole multi-agent procedure, and it is saved for future works. For now we concentrate on the design of an MPC which grants individual collision avoidance, and assume that the described condition does not occur.

2.4.1 Mathematical definition

We consider a set of M obstacles described by its bounding ellipsoid, identified by the pair:

$$\mathcal{O}_i = (\boldsymbol{c}, A) \in \mathbb{R}^3 \times \mathbb{R}^{3\times 3} \tag{2.34}$$

where $\boldsymbol{c} \in \mathbb{R}^3$ is the center of the ellipsoid and A is a matrix whose eigenvectors represent the principal axes of the ellipsoid $\boldsymbol{e}_1$, $\boldsymbol{e}_2$, $\boldsymbol{e}_3$ and whose eigenvalues are the inverse squared length of each axis. In general, A can be always written as:

$$A = \begin{bmatrix} \boldsymbol{e}_1 & \boldsymbol{e}_2 & \boldsymbol{e}_3 \end{bmatrix}^\top \begin{bmatrix} 1/a^2 & & \\ & 1/b^2 & \\ & & 1/c^2 \end{bmatrix} \begin{bmatrix} \boldsymbol{e}_1 & \boldsymbol{e}_2 & \boldsymbol{e}_3 \end{bmatrix} \tag{2.35}$$

Then the equation describing the boundary of the ellipse is:

$$g_A(\boldsymbol{x}) = (\boldsymbol{x} - \boldsymbol{c})^\top A (\boldsymbol{x} - \boldsymbol{c}) - 1 = 0 \tag{2.36}$$

We could also define bounding ellipsoid for each link and phyisical element of the robots and then define the collision avoidance with the distance between ellipsoid. However, computing such distance is not trivial, and needs to be performed each optimization step, resulting in very high computational burden. What it is immediate to derive is to check whether a point $\boldsymbol{x}_0 \in \mathbb{R}^3$ is outside the ellipsoid, which occurs if and only if:

$$g_A(\boldsymbol{x}_0) > 0 \tag{2.37}$$

Exploiting this relation, we consider a set of *keypoints* on each agent:

$$\mathcal{K} = \{\boldsymbol{k}_i \in \mathbb{R}^3, i = 1, \dots, K\} \tag{2.38}$$

Then, we enlarge the obstacles isotropically by a factor r, i.e. $a' = a + r$, $b' = b + r$, $c' = c + r$, and consider the ellipsoid described by A' with the same eigenvector as A and eigenvalues a', b', c'. In this way the condition:

$$g'_A(\boldsymbol{x}_0) = 0 \tag{2.39}$$

means that $\boldsymbol{x}_0$ is at distance r from the original ellipsoid $(\boldsymbol{c}, A)$. In this way we are considering a sphere of radius r around $\boldsymbol{k}_i$ which must not intersect the obstacle ellipsoid. We choose the set of keypoints $\mathcal{K}$ to be sufficiently spread over the volume of each agent, i.e. the union of such spheres should approximately cover the whole robot, if we want to ensure the obstacle avoidance. However, unless pathological cases, it can be sufficient to place the keypoints into a few points: the wheels or the fans, each joint and the end effector. In case of very long links it could be appropriate to insert additional keypoint equally spaced throughout the link. In the specific case of the small ground and aerial robots we are considering, though, the keypoints, for the base, joints and end-effector are sufficient.

2.4.2 Robust obstacle avoidance with MPC

The simplest approach is to add the collision avoidance constraint in the MPC, i.e. add to (2.11) a constraint of the form:

$$\begin{aligned} &\text{subject to: } [\dots] \\ &\qquad\qquad g_i(\boldsymbol{k}_j) > 0, \quad \text{for } i = 1, \dots, M \text{ and } j = 1, \dots, K \end{aligned} \tag{2.40}$$

However, usually the optimal trajectory lies on the boundary of such constraint. If for a drift in the actual model or a small external disturbance one keypoint ends up inside the enlarged obstacle, the constraints are not met for the initial condition of the next MPC step, in which the problem becomes unsolvable. We then avoid such constraint by adding a cost term in the minimization function. Indeed, note that the function $\boldsymbol{g}_i(\boldsymbol{x})$ is a quadratic function in $\boldsymbol{x}$, and it is positive definite since the eigenvalues of A are positive. Then, it is monotonically decreasing for $\boldsymbol{x}$ approaching the $\boldsymbol{c}$ and positive

outside the boundary of the ellipsoid. We can then define, for each obstacle $i = 1, \dots, M$ the following term:

$$\chi_i(\boldsymbol{x}) = C_i e^{-\lambda_i \sqrt{g_i(\boldsymbol{x})}} \tag{2.41}$$

where C_i is the desired value of the cost at the boundary ofthe ellipsoid, λ_i are decay factors which is convenient to define as:

$$\lambda_i = -\log(\varepsilon)/m_i \tag{2.42}$$

with $\varepsilon > 0$ small and $m_i > 0$ so that the exponential function takes the value ε when $g_i(\boldsymbol{x}) = m_i$, so that m_i defines the margin around the obstacle where the cost becomes negligible.

As an example, consider the simplest case in which $A = \boldsymbol{I}_3$, so that the obstacle is a unitary sphere centered in $\boldsymbol{c}$. Then, consider the enlarged obstacle with $A' = \boldsymbol{I}_3 \frac{1}{(1+r)^2}$, then the quadratic function corresponds to:

$$g_{A'}(\boldsymbol{x}) = \frac{(x_1 - c_1)^2 + (x_2 - c_2)^2 + (x_3 - c_3)^2}{(1+r)^2} - 1 \tag{2.43}$$

It is clear that the boundary $g(\boldsymbol{x}) = 0$ describes a sphere of radius $1 + r$, then it becomes:

$$\chi(\boldsymbol{x}) = Ce^{-\lambda/(1+r)(\|\boldsymbol{x}-\boldsymbol{c}\|-r)} \tag{2.44}$$

From (2.44) we understand that function χ is an exponential that decays with the distance of $\boldsymbol{x}$ from $\boldsymbol{c}$, with a margin r. Figure 2.1 provides an example of the trend of χ as a function of the distance $d := \|\boldsymbol{x} - \boldsymbol{c}\|$.

Then, we can add to the MPC problem in (2.11) a term that accounts for each keypoint $\boldsymbol{k}_j$, $j = 1, \dots, K$ a cost proportionalto the distance to each obstacle $i = 1, \dots, M$:

$$\boldsymbol{h}_{q+1}(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{d}_k) = \sum_{i=1}^{M} \sum_{j=1}^{K} \chi_i(\boldsymbol{k}_j) \tag{2.45}$$

where q is the size of the output function $\boldsymbol{h}$ previously considered and the information about the obstacles are encoded in the online data $\boldsymbol{d}_k$. The keypoints values can be either computed online as a function of the joints or can be supplied externally as online data $\boldsymbol{d}_k$.

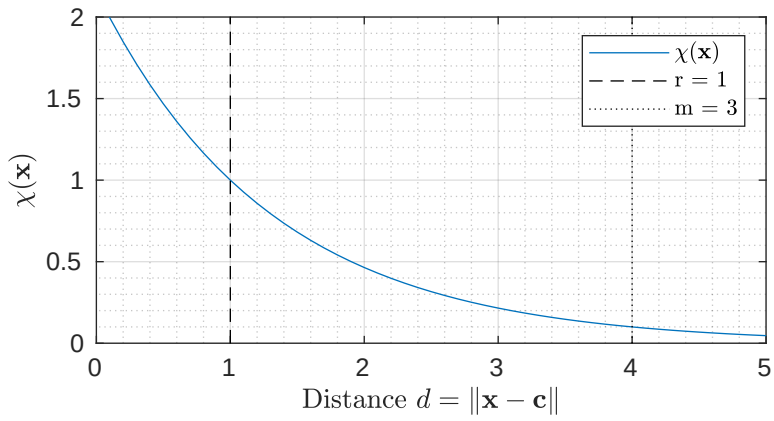


Figure 2.1: Example of $\chi(d)$ with $r = 1$, $m = 3$, $\varepsilon = 0.1$

As an example, consider the case of one obstacle. The cost function resulting from the weighted sum of the contribute for obstacle avoidance and for the distance to the reference tracking, considered on the plane xy, looks like in figure 2.2

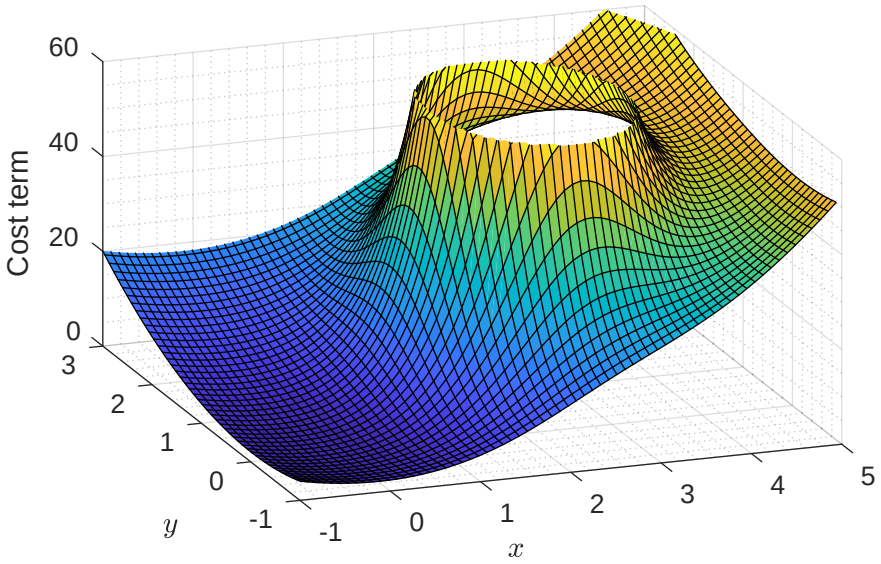


Figure 2.2: Example surface plot of the cost function of generic points on a plane considering both the cost for reference tracking and obstacle avoidance.

3 Kinematic Robots Modeling and Simulation

In this work we will consider a specific implementation with $N = 2$ heterogeneous agents: one ground and one aerial. As anticipated in the introduction, the ground robot is composed of an omnidirectional base, which is fully actuated on the plane, equipped with a 4 d.o.f. arm. The aerial robot is an underactuated hexacpoter comes with a 2 d.o.f. arm. In the following we will provide a model of the kinematics of the two robots. Since the purpose is to adopt it in an MPC, we choose to provide a first order model, as the full second-order dynamics would require an high control rate and a high calculation complexity at the same time, which would require an exponentially more computational power. For notation convenience, hereafter instead of the indexed notation for the agents $i = 1, 2$, we will refer to a quantity relative to the ground robot with a subscript g and to the aerial robot with a subscript a

3.1 Ground Robot Modeling

We start by defining the constraints on the inputs and the involved quantities:

- $\boldsymbol{R}_{v,g} \in \mathbb{SO}(3)$, the orientation of the vehicle w.r.t. the world frame, constrained in velocity by:

$$\boldsymbol{\omega}_{v,g} = u_{\omega,\psi,g}\boldsymbol{z}_g \tag{3.1}$$

  where $\boldsymbol{z}_g = \boldsymbol{z}$ is the z-axis of the vehicle frame, which correspond to the same axis of world frame: $\boldsymbol{z}_g = [0,\ 0,\ 1]^\top$. This means that only one angle is enough to representate $\boldsymbol{R}_{v,g}$, so we will consider without representation issues a dynamics on the angle:

$$\dot{\psi}_{v,g} = u_{\omega,\psi,g} \tag{3.2}$$

and $\boldsymbol{R}_{v,g} = \boldsymbol{R}_z(\psi_{v,g})$, with $\boldsymbol{R}_z$ indicating a rotation on z. Also, in the model (2.1), we have the linear velocity input as $\boldsymbol{\omega}_{v,u} = [0, 0, u_{\omega,\psi,g}]^\top$.

- $\boldsymbol{p}_{v,g} \in \mathbb{R}$, the position of the ground vehicle w.r.t. the world frame, constrained in velocity by:

$$\dot{\boldsymbol{p}}_{v,g} = u_{v,x,g}\boldsymbol{x}_g + u_{v,y,g}\boldsymbol{y}_g, \quad \boldsymbol{u}_{v,g} = \begin{bmatrix} u_{v,x,g}, u_{v,y,g} \end{bmatrix}^\top \tag{3.3}$$

where $\boldsymbol{x}_g$ and $\boldsymbol{y}_g$ are the versor of x and y axis in the ground vehicle frame, that are the first two columns of the matrix $\boldsymbol{R}_{v,g}$, i.e:

$$\boldsymbol{x}_g = \begin{bmatrix} \cos(\psi_{v,g}) \\ \sin(\psi_{v,g}) \\ 0 \end{bmatrix}, \qquad \boldsymbol{y}_g = \begin{bmatrix} -\sin(\psi_{v,g}) \\ \cos(\psi_{v,g}) \\ 0 \end{bmatrix} \tag{3.4}$$

The dynamics on the z component of $\boldsymbol{p}_{v,g}$ is trivially zero, so it results $p_{v,g,x} = z_g$, where z_g is a fixed parameter wich depends on the geometry and on the world reference fame. We then will consider $z_g = 0$.

- $\boldsymbol{q}_g = \begin{bmatrix} \vartheta_1, & \vartheta_2, & \vartheta_3, & \vartheta_4 \end{bmatrix}^\top \in \mathbb{R}^4$ is the vector of joints variable. For the considered manipulator, they are 4 angles representing the position of revolute joints.

From the constraints we can define the constraint matrices from (2.1) as:

$$\begin{aligned} \boldsymbol{A}_{p,g} &= \begin{bmatrix} \boldsymbol{x}_g & \boldsymbol{y}_g & \boldsymbol{0} \end{bmatrix} \\ \boldsymbol{A}_{o,g} &= \begin{bmatrix} \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{z}_g \end{bmatrix} \end{aligned} \tag{3.5}$$

In order to derive the expression of the position and orientation jacobians, we need to first compute the direct kinematics. In the next sections we will drop the g subscript, as we only deal with the ground robot.

3.1.1 Direct Kinematics

To find the transform from the world frame to the end-effector we consider the kinematic chain given by the product of relative transformation between the successive links:

$$\boldsymbol{T}_e^w = \boldsymbol{T}_b^w(\boldsymbol{p}_v, \psi_v)\boldsymbol{T}_0^b\boldsymbol{T}_1^0(\vartheta_1)\boldsymbol{T}_2^1(\vartheta_2)\boldsymbol{T}_3^2(\vartheta_3)\boldsymbol{T}_e^3(\vartheta_4) \tag{3.6}$$

where ϑ_i are the joint variables, s.t. $\boldsymbol{q} = [\vartheta_1, \dots, \vartheta_4]^\top$ and:

$$\boldsymbol{T}_b^w(\boldsymbol{p}_v, \psi_v) = \begin{bmatrix} \boldsymbol{R}_v(\psi_v) & \boldsymbol{p}_v \\ \boldsymbol{0} & 1 \end{bmatrix} \tag{3.7}$$

The reference frames are set according to the *Denavit-Hartenberg* convention and the transformations can be expressed as follows:

- The first transform accounts for possible position and rotation offset between the body frame and the first link of the manipulator. In the specific case, we consider no rotation offset, and:

$$\boldsymbol{R}_0^b = \boldsymbol{I}_3, \quad \boldsymbol{o}_0^b = \begin{bmatrix} \ell_{b,x} \\ \ell_{b,y} \\ \ell_{b,z} \end{bmatrix}, \quad \boldsymbol{T}_0^b = \begin{bmatrix} \boldsymbol{I}_3 & \boldsymbol{o}_0^b \\ \boldsymbol{0} & 1 \end{bmatrix} \tag{3.8}$$

  where $\ell_{b,x}$, $\ell_{b,y}$, $\ell_{b,z}$ are known constants, reported in appendix.

- The second transform defines the rotation of the shoulder joint about the axis z of the body frame. There is no displacement between the two links and we consider a rotation about the x axis to align the frame to the next link.

$$\boldsymbol{R}_1^0 = \boldsymbol{R}_z(\vartheta_1)\boldsymbol{R}_x(\tfrac{\pi}{2}), \quad \boldsymbol{o}_1^0 = \boldsymbol{0}, \quad \boldsymbol{T}_1^0 = \begin{bmatrix} \boldsymbol{R}_1^0 & \boldsymbol{o}_1^0 \\ \boldsymbol{0} & 1 \end{bmatrix} \tag{3.9}$$

- The transformation regarding the first non-zero link is splitted in two parts, due to its "L" shape:

$$\begin{aligned} \boldsymbol{R}_{2'}^1 &= \boldsymbol{R}_z(\vartheta_2), \quad \boldsymbol{o}_{2'}^1 = \ell_{1,a}\boldsymbol{R}_{2'}^1\boldsymbol{x} \\ \boldsymbol{R}_2^{2'} &= \boldsymbol{R}_z(-\tfrac{\pi}{2}), \quad \boldsymbol{o}_2^{2'} = \ell_{1,b}\boldsymbol{R}_2^{2'}\boldsymbol{x} \\ \boldsymbol{T}_2^1 &= \begin{bmatrix} \boldsymbol{R}_{2'}^1\boldsymbol{R}_2^{2'} & \boldsymbol{o}_{2'}^1 + \boldsymbol{R}_{2'}^1\boldsymbol{o}_2^{2'} \\ \boldsymbol{0} & 1 \end{bmatrix} \end{aligned} \tag{3.10}$$

  where $\ell_{1,a}$ and $\ell_{2,b}$ are the lenghts of the longest and the shortest segment, respectively.

- The derivations for the last two links are straightforward:

$$\begin{aligned} \boldsymbol{R}_3^2 &= \boldsymbol{R}_z(\vartheta_3), \quad \boldsymbol{o}_3^2 = \ell_2 \boldsymbol{R}_3^2 \boldsymbol{x}, \quad \boldsymbol{T}_3^2 = \begin{bmatrix} \boldsymbol{R}_3^2 & \ell_2 \boldsymbol{R}_3^2 \boldsymbol{x} \\ \boldsymbol{0} & 1 \end{bmatrix} \\ \boldsymbol{R}_e^3 &= \boldsymbol{R}_z(\vartheta_4), \quad \boldsymbol{o}_e^3 = \ell_2 \boldsymbol{R}_e^3 \boldsymbol{x}, \quad \boldsymbol{T}_e^3 = \begin{bmatrix} \boldsymbol{R}_e^3 & \ell_3 \boldsymbol{R}_e^3 \boldsymbol{x} \\ \boldsymbol{0} & 1 \end{bmatrix} \end{aligned} \tag{3.11}$$

  where ℓ_2, ℓ_3 are the known lenghts of the last two links.

From the kinematic chain we can then find a closed form definition for $\boldsymbol{p}_e^b$ and $\boldsymbol{R}_e^b$, whose expression is cumbersome and it is not reported.

3.1.2 Differential Kinematics

After having derived an expression for $\boldsymbol{p}_e(\boldsymbol{q})$, we can find the position and orientation Jacobians, defined as:

$$\begin{aligned} \boldsymbol{J}_O &= \begin{bmatrix} \dfrac{\partial \boldsymbol{p}_e}{\partial \vartheta_1}, & \ldots, & \dfrac{\partial \boldsymbol{p}_e}{\partial \vartheta_4} \end{bmatrix} \\ \boldsymbol{J}_P &= \begin{bmatrix} \boldsymbol{\omega}_{0,1}, & \ldots, & \boldsymbol{\omega}_{3,e} \end{bmatrix} \end{aligned} \tag{3.12}$$

where $\omega_{i-1,i}$ are the relative angular velocities between links.

If we denote the by $\boldsymbol{J}_{P_h}$ the h-th column of the positional Jacobian, since all joints are revolutes, it holds:

$$\begin{aligned} \boldsymbol{J}_{P_1} &= \boldsymbol{z}_b \times (\boldsymbol{p}_e - \boldsymbol{p}_v) \\ \boldsymbol{J}_{P_2} &= \boldsymbol{z}_1 \times (\boldsymbol{p}_e - \boldsymbol{p}_0) \\ \boldsymbol{J}_{P_3} &= \boldsymbol{z}_2 \times (\boldsymbol{p}_e - \boldsymbol{p}_1) \\ \boldsymbol{J}_{P_4} &= \boldsymbol{z}_3 \times (\boldsymbol{p}_e - \boldsymbol{p}_2) \end{aligned} \tag{3.13}$$

where $\boldsymbol{z}_h = \boldsymbol{R}_h \boldsymbol{z}$ is the z-axis versor of the h-th reference frame, defined by $(\boldsymbol{p}_h,\ \boldsymbol{R}_h)$ which can be easily derived from (3.6) by stopping the multiplication at the h-th term.

Deriving the orientation Jacobian is straightforward:

$$\boldsymbol{J}_O = \begin{bmatrix} \boldsymbol{z}_b, & \boldsymbol{z}_1, & \boldsymbol{z}_2, & \boldsymbol{z}_3 \end{bmatrix} \tag{3.14}$$

We now have provided a closed form definition of every symbol in (2.1) so the ground agent model is well defined by:

$$\dot{\boldsymbol{x}}_g = \mathcal{A}_g(\boldsymbol{x}_g)\boldsymbol{u}_g \tag{3.15}$$

with the notation already introduced in the previous section.

3.2 Aerial Robot Modeling

As for the ground robot, we will model the first order kinematic of the UAV. However, the assumption to be able to control the velocity is more critical since the vehicle is underactuated: this implies that not only the velocity input is not tracked with zero error, but to achieve that movement there is an additional dynamics on the vehicle roll and pitch attitude which may introduce significant errors on the tracking of the end effector. This behavior is impossible to model with a first order differential relation as it depends on the low-level dynamics of the vehicle. On the other hand, with a full dynamics modeling it would be hard for the MPC to compute the control within the even shorter time step a flying robot requires. In this implementation we will assume that the overall dynamics is slow, so that these errors are negligible and we can model the vehicle with 4 un coupled integrators. Moreover, having a good velocity controller is crucial and not as easy to tune as for the ground robot. We will in any case employ a first order model in the MPC and try to integrate it with the low-level controller in a smart way which will overcome both issues. We will discuss about this in the subsequent sections.

We proceed by defining the variables and the constraints affecting the aerial robot:

- $\boldsymbol{R}_{v,a} \in \mathbb{SO}(3)$, the orientation of the vehicle in the world frame. We model the constraint on the angular velocity as:

$$\boldsymbol{\omega}_a = u_{\omega,\psi,a}\boldsymbol{z}_a \tag{3.16}$$

  where $\boldsymbol{z}_a$ is the versor of the z-axis of the vehicle frame. We stress the fact that, despite this modeling, the low-level controller will provide angular velocities components to roll and pitch. The errors caused by breaking this assumptions will be assessed experimentally. As with

the ground vehicle, we will represent this rotation and its dynamics through the yaw angle only, obtaining:

$$\dot{\psi}_{v,a} = u_{\omega,\psi,a} \tag{3.17}$$

being $\boldsymbol{R}_{v,a} = \boldsymbol{R}_z(\psi_{v,a})$

- $\boldsymbol{p}v, a \in \mathbb{R}^3$, the position of the aerial vehicle in the world frame, constrained in velocity according to:

$$\dot{\boldsymbol{p}}_{v,a} = u_{v,x,g}\boldsymbol{x}_a + u_{v,y,g}\boldsymbol{y}_a + u_{v,y,g}\boldsymbol{z}_a \tag{3.18}$$

where $\begin{bmatrix}\boldsymbol{x}_a, & \boldsymbol{y}_a, & \boldsymbol{z}_a\end{bmatrix} = \boldsymbol{R}_{v,a}$.

- $\boldsymbol{q}_a = \begin{bmatrix}\vartheta_{1,a}, & \vartheta_{2,a}\end{bmatrix}$ is the vector of variables representing the angular position of the two revolute joints.

The constraint matrices are then:

$$\begin{aligned}\boldsymbol{A}_{p,a} &= \boldsymbol{R}_{v,a} \\ \boldsymbol{A}_{\omega,a} &= \begin{bmatrix}\boldsymbol{0} & \boldsymbol{0} & \boldsymbol{z}_a\end{bmatrix}\end{aligned} \tag{3.19}$$

In the next sections we will drop the a subscript, as we only deal with the aerial robot.

3.2.1 Direct Kinematics

In the same way as for the ground manipulator, we consider a chain of reference transformations:

$$\boldsymbol{T}_e^w = \boldsymbol{T}_b^w(\boldsymbol{p}_v, \psi_v)\boldsymbol{T}_0^b\boldsymbol{T}_1^0(\vartheta_1)\boldsymbol{T}_e^1(\vartheta_2) \tag{3.20}$$

where:

$$\boldsymbol{T}_e^w(\boldsymbol{p}_v, \psi_v) = \begin{bmatrix}\boldsymbol{R}_v(\psi_v) & \boldsymbol{p}_v \\ \boldsymbol{0} & 1\end{bmatrix} \tag{3.21}$$

where the transformation are defined as follows:

- The first transformation is to align z-axis with the axis of rotation of the first joint, accounting also for the in an offset:

$$\boldsymbol{R}_0^b = \boldsymbol{R}_x(\tfrac{\pi}{2})\boldsymbol{R}_z(-\tfrac{\pi}{2}), \quad \boldsymbol{o}_0^b = \begin{bmatrix} \ell_{b,x} \\ \ell_{b,y} \\ \ell_{b,z} \end{bmatrix}, \quad \boldsymbol{T}_0^b = \begin{bmatrix} \boldsymbol{R}_0^b & \boldsymbol{o}_0^b \\ \boldsymbol{0} & 1 \end{bmatrix} \tag{3.22}$$

  where $\ell_{b,x}, \ell_{b,y}, \ell_{b,z}$ are known constant parameters.

- The second transformation, alike the corresponding one in the ground robot, is composed into two parts to account for the "L"-shaped link:

$$\begin{aligned} \boldsymbol{R}_{1'}^0 &= \boldsymbol{R}_z(\vartheta_1), \quad \boldsymbol{o}_{1'}^0 = \ell_{1,a}\boldsymbol{R}_{1'}^0\boldsymbol{x} \\ \boldsymbol{R}_1^{1'} &= \boldsymbol{R}_z(\tfrac{\pi}{2}), \quad \boldsymbol{o}_1^{1'} = \ell_{1,b}\boldsymbol{R}_1^{1'}\boldsymbol{x} \\ \boldsymbol{T}_1^0 &= \begin{bmatrix} \boldsymbol{R}_{1'}^0\boldsymbol{R}_1^{1'} & \boldsymbol{o}_{1'}^0 + \boldsymbol{R}_{1'}^0\boldsymbol{o}_1^{1'} \\ \boldsymbol{0} & 1 \end{bmatrix} \end{aligned} \tag{3.23}$$

  where $\ell_{1,a}$ and $\ell_{2,b}$ are the lengths of the longest and the shortest segment, respectively.

The closed form definition for $\boldsymbol{p}_e$, can be derived from the kinematic chain just like in the previous case.

3.2.2 Differential Kinematics

The expressions for the positional and orientational Jacobian are straightforward:

$$\begin{aligned} \boldsymbol{J}_{P_1} &= \boldsymbol{z}_0 \times (\boldsymbol{p}_e - \boldsymbol{p}_v) \\ \boldsymbol{J}_{P_2} &= \boldsymbol{z}_1 \times (\boldsymbol{p}_e - \boldsymbol{p}_1) \end{aligned} \tag{3.24}$$

where $\boldsymbol{z}_h$, $\boldsymbol{p}_1^b$ can be easily retrieved from the chain of transformation, properly truncated. We now have defined all elements of (2.1), so we have the aerial vehicle kinematic model:

$$\dot{\boldsymbol{x}}_a = \mathcal{A}_a(\boldsymbol{x}_a)\boldsymbol{u}_a \tag{3.25}$$

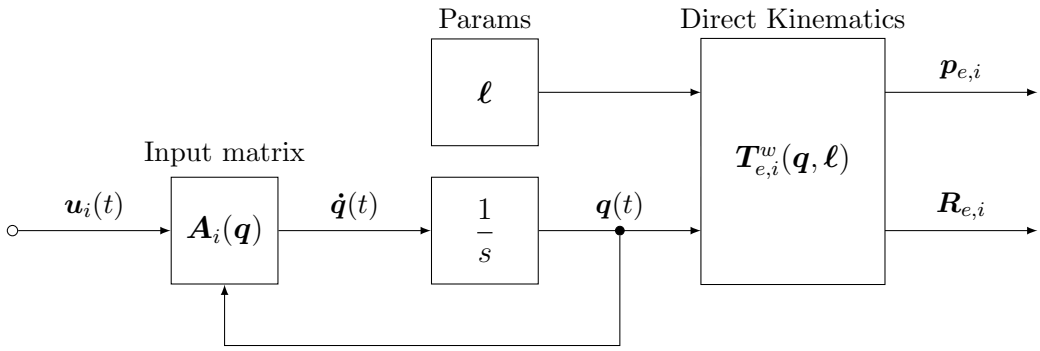


Figure 3.1: Kinematic simulation block diagram.

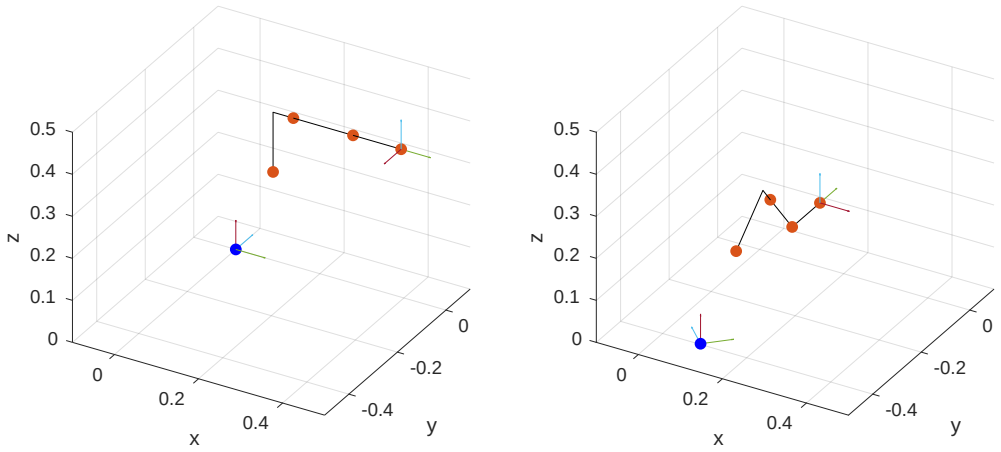
3.3 Pure kinematic simulation

In order to quickly validate the algorithm we set up a simulation environment in Simulink. To do so, we employ the kinematics model of the robots, which means that we are simulating the ideal case in which the robots are able to track velocity inputs with zero error and there are no second order dynamics effects on the structure. This is certainly a hard assumption and we are not able to predict how the results will differ from the reality and this means that such simulation is not enough to assess the performance of the algorithm. Nevertheless, this is useful to test the MPC for individual control and make some preliminary considerations on the cooperative algorithm.

3.3.1 Individual control

In Fig. 3.1 the simulink block diagram of the kinematic simulation for a generic agent i is reported. For the numerical computation, the integrator is discretized by Simulink via Euler method. Note that for the robot simulation we are using the direct kinematics, while in the MPC the Jacobian is used. This allows for an additional consistency check between the two.

Regarding the implementation of the MPC solver, ACADO generated code is wrapped in an `s-function` which provides a ready-to-use simulink block which is then inserted in a classical feedback loop by stacking the output of the simulation in Fig. 3.1, $\boldsymbol{p}_{e,i}$, $\boldsymbol{R}_{e,i}$ and $\boldsymbol{q}_i$ in a single vector to obtain $\boldsymbol{x}_i \in \mathbb{R}^{19}$. The ACADO code generation is described in detail in the next chapter.



(a) Initial **(b)** Final

Figure 3.2: Ground robot representation

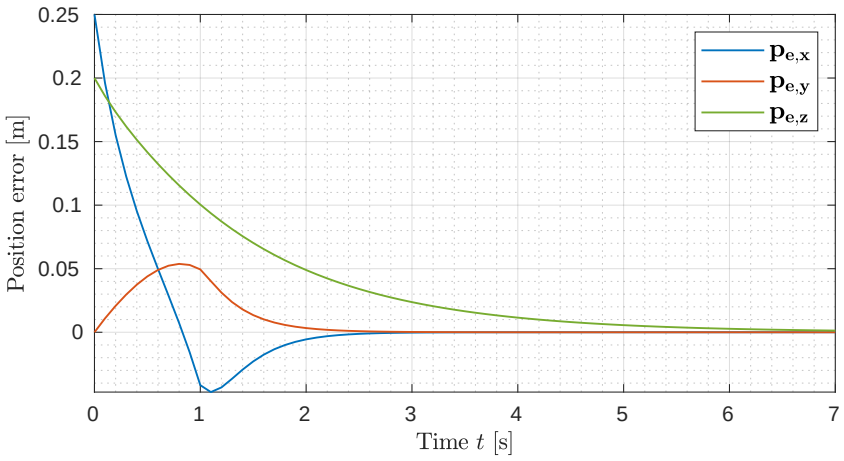
Ground Robot Control

In this section we briefly present the pure kinematics result for the ground robot, whose model derived in the previous section is used for the MPC, so that to control the pose of the end effector. We consider a starting position defined by $\boldsymbol{p}_v v, 0 = \mathbf{0}$, $\boldsymbol{q}_{g,0} = \left[0, \frac{\pi}{2}, 0, 0\right]$ which is represented in Fig. 3.2a. Then, we set a constant reference pose with a small position displacement and a rotation of $\frac{\pi}{2}$ about the z axis of world frame, which, in the frame of end effector, is represented by the pose:

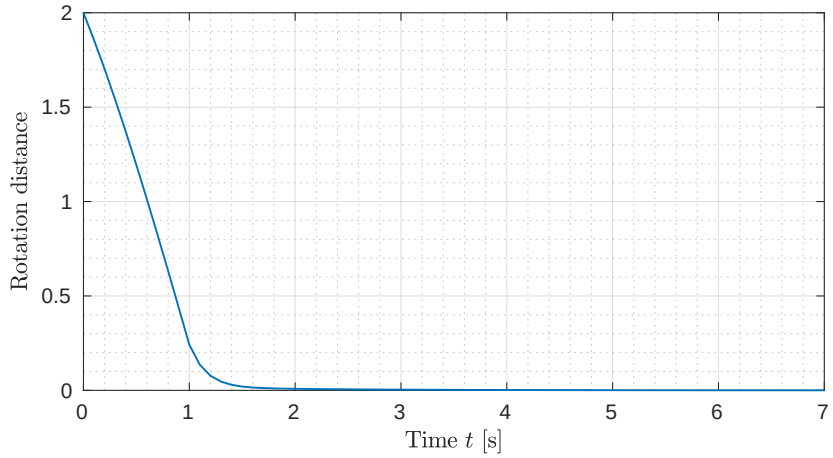
$$\begin{aligned} \boldsymbol{p}_{e,g,des} &= \boldsymbol{p}_{e,g,0} + [-0.25, 0, -0.2] \\ \boldsymbol{R}_{e,g,des} &= \boldsymbol{R}_{e,g,0} \boldsymbol{R}_y(\tfrac{\pi}{2}) \end{aligned} \tag{3.26}$$

where $(\boldsymbol{p}_{e,g,0}, \boldsymbol{R}_{e,g,0}) = \boldsymbol{T}^w_{e,g}(\boldsymbol{q}_{g,0})$. Note that a rotation about z in the world frame correspond to a rotation about y in the initial end-effector frame.

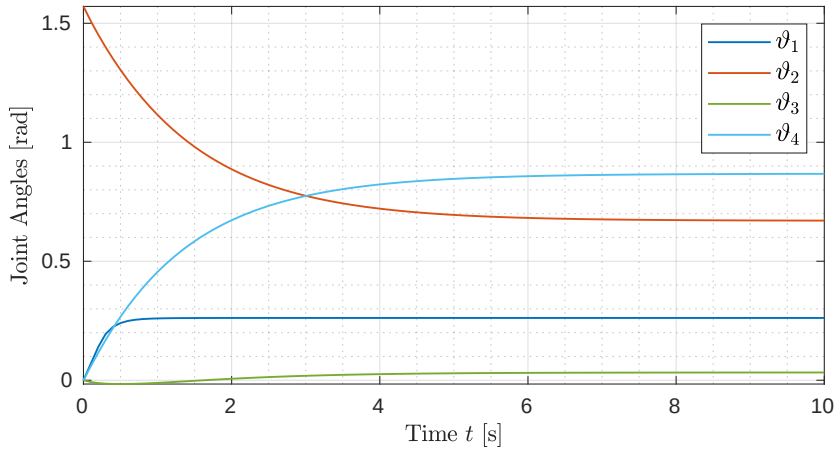
Since we cannot assess the performances and we are only interested in testing the stability of the controller, the choice of the cost matrices is not relevant and we set an unitary weight to the pose error penalty while 0.1 for the cost on the controls. As regards the sample time and prediction horizon, the computing time does not represent a limitation in Simulink as the simulation is not real time, and the computation can be as long as needed. On the other hand, it appeared during the experiments that a wider time window



(a) Position error



(b) Orientation error norm $\mathrm{d}_{\mathbb{SE}(3)}$



(c) Joints values

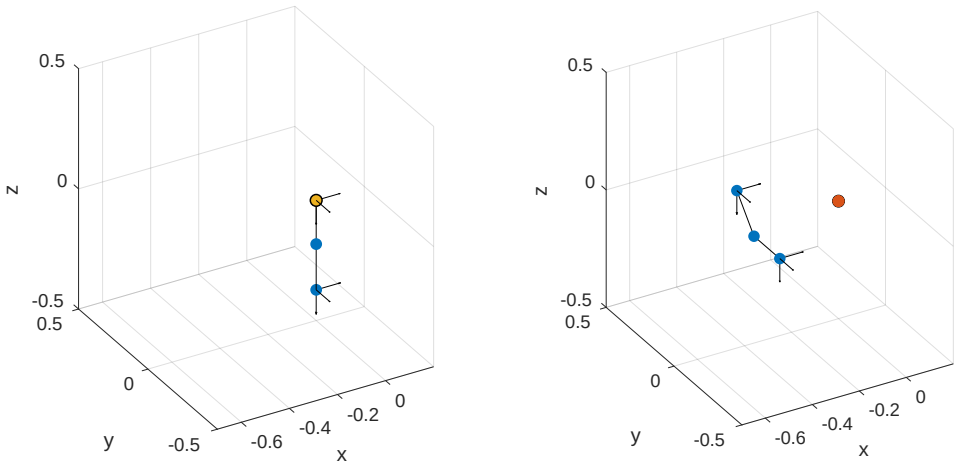
Figure 3.3: MPC control test for ground end-effector

does not improve the performances. This is reasonable since all the model assumptions are met in such simulation. However, to keep consistency to the gazebo and experiments, we choose a quite slow control with $T_s = 0.1\,\mathrm{s}$ and a prediction horizon $N = 10$ samples, which means a time window $T_p = 1\,\mathrm{s}$.

The results are reported in Fig. 3.3. The MPC is able to stabilize the error dynamics on the end effector, both in position and orientation. From the graphs we note that the slowest component is on the z axis. This is due to the fact that, while linear and rotational movements are directly achievable with some inputs (either the base or the first joint velocities), the movement along z requires the joints to properly perform coordinated movements in order to avoid increasing the error on rotation. As a result, such kind of movements have an higher cost in the optimal problem and are then slower.

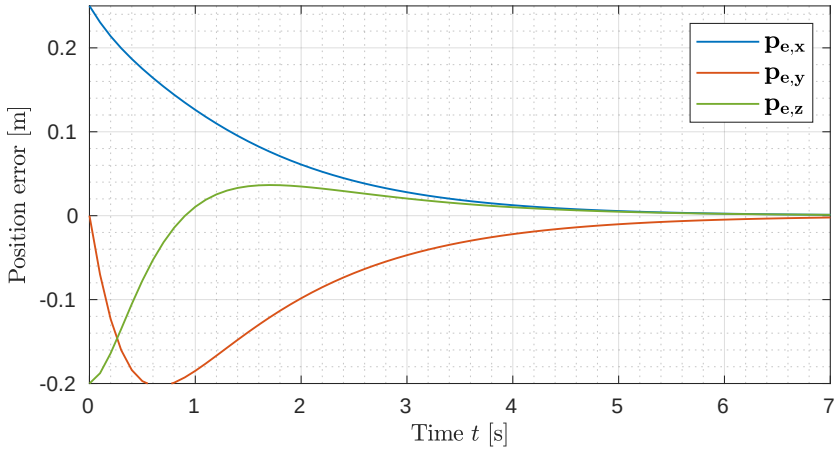
Aerial Robot Control

In the pure kinematic simulation, testing the aerial model is completely analogous, since no underactuated dynamics is taken into account. The main difference is that the manipulator has few degrees of freedom whereas the vehicle has a lot more mobility. This means that almost every configurations are achieved by moving the vehicle rather than the joints. Here we will test

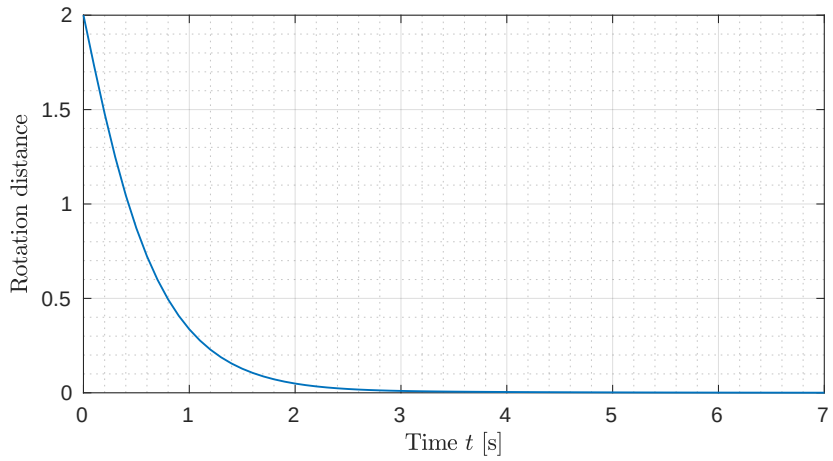


(a) Initial **(b)** Final

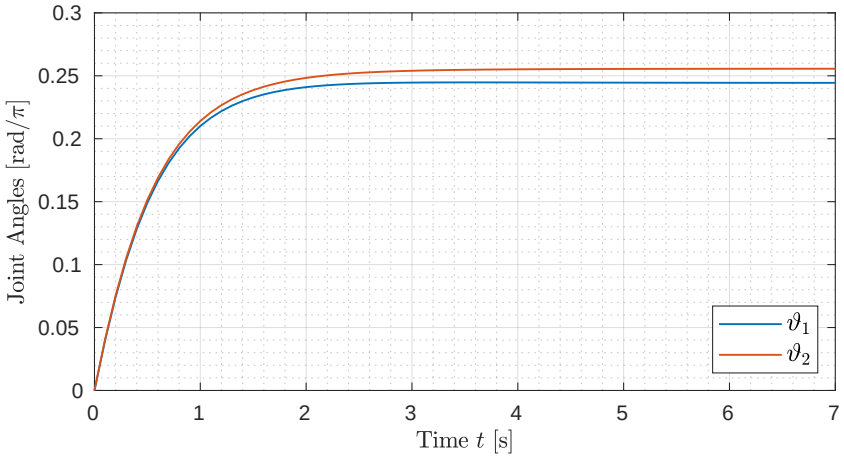
Figure 3.4: Aerial robot representation



(a) Position error



(b) Orientation error norm $\mathrm{d}_{\mathbb{SE}(3)}$



(c) Joints values

Figure 3.5: MPC control test for aerial end-effector

both the vehicle and the joints by considering an initial configuration defined by $\boldsymbol{p}_{v,a,0} = \mathbf{0}$, $\boldsymbol{R}_{v,a,0} = \boldsymbol{I}_3$ and $\boldsymbol{q}_{a,0} = \mathbf{0}_2$, with setpoint:

$$\begin{aligned} \boldsymbol{p}_{e,a,des} &= \boldsymbol{p}_{e,a,0} + [-0.25, 0, -0.2] \\ \boldsymbol{R}_{e,a,des} &= \boldsymbol{R}_{e,a,0}\boldsymbol{R}_z(\tfrac{\pi}{2}) \end{aligned} \tag{3.27}$$

where $(\boldsymbol{p}_{e,a,0}, \boldsymbol{R}_{e,a,0}) = \boldsymbol{T}^w_{e,a}(\boldsymbol{q}_{a,0})$ in the same way as the ground controller. Note that in the end effector the z axis is the rotation axis of the joints. The initial and final configurations are depicted in figure 3.4a and 3.4b.

The simulation results are reported in 3.5. The same time window and sample time as the ground case have been used and we note that it is able to stabilize the error dynamics with comparable performances as as the ground MPC. Note that by setting the same weight to the two joints control input the rotation of $\frac{\pi}{2}$ is obtained with a displacement of around $\frac{\pi}{4}$ for each joint so the control effort is equally shared among the two joints.

3.4 Cooperative Algorithm Simulation

We can directly employ the two MPCs to implement the multiagent algorithm in Simulink. The overall scheme is reported in Fig. 3.6. Obstacle avoidance will also be tested afterwards. We assume the object is positioned as $\boldsymbol{T}_o = ([0, 0, 0.25], \boldsymbol{I}_3)$. Preliminarly, we exploit the individual controllers to find a configuration such that the end effectors are aligned along the y axis:

$$\begin{aligned} \boldsymbol{T}_{e,g} &= ([0, -0.1, 0.25], \boldsymbol{R}_{g,y}) \\ \boldsymbol{T}_{e,a} &= ([0, \quad 0.1, 0.25], \boldsymbol{R}_{a,-y}) \end{aligned} \tag{3.28}$$

where $\boldsymbol{R}_{g,y}$ and $\boldsymbol{R}_{a,y}$ are rotations in each end-effector frame that align the frame with y with positive and negative direction respectively. The initial configuration is depicted in Fig. 3.7a. Then, we simulate phases 1 (*Handshake*) and 2 (*Transform estimation*) by computing the relative transform from ground to leader $\boldsymbol{T}_{e,a}^{e,g}$, which we will use as constant estimate throughout the prediction, i.e.

$$\hat{\boldsymbol{T}}_{e,a}^{e,g}(t) = \hat{\boldsymbol{T}}_{o}^{e,g}(t) \left(\hat{\boldsymbol{T}}_{o}^{e,a}(t)\right)^{-1} \equiv \boldsymbol{T}_{o}^{e,g}(0) \left(\boldsymbol{T}_{o}^{e,a}\right)^{-1}(0) \tag{3.29}$$

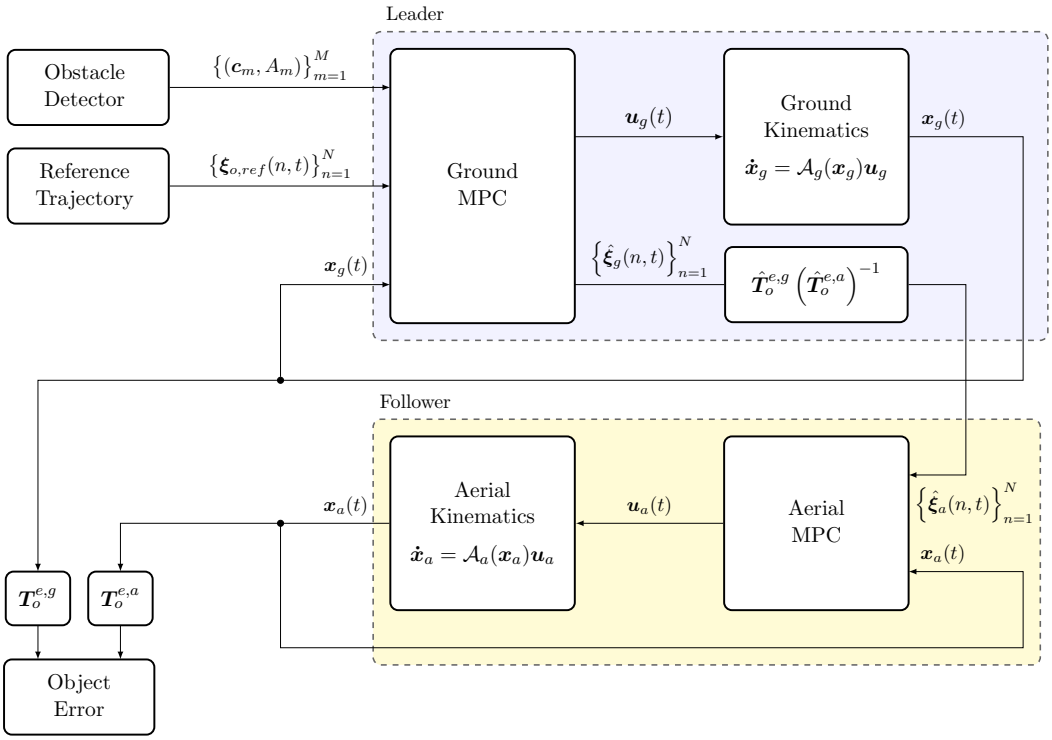
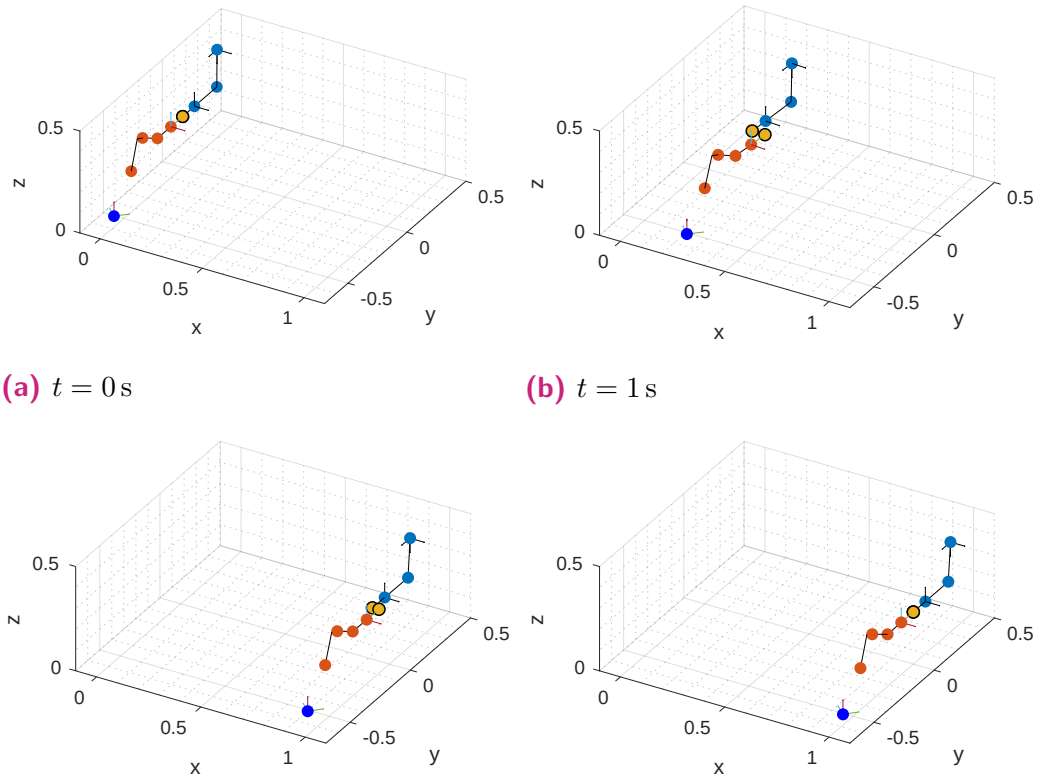


Figure 3.6: Block diagram of the overall system.



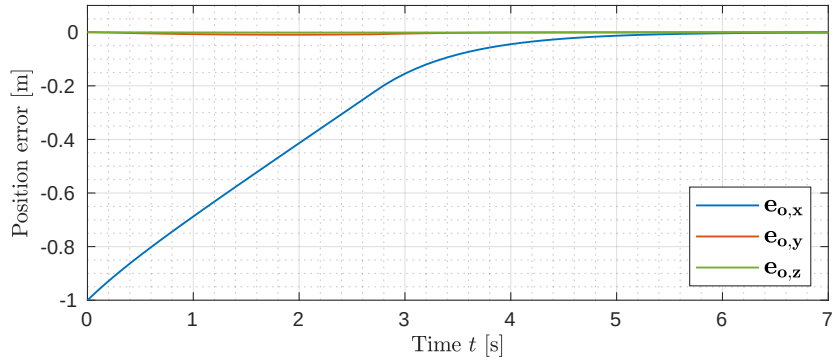
(a) $t = 0\,\mathrm{s}$ **(b)** $t = 1\,\mathrm{s}$

(c) $t = 4\,\mathrm{s}$ **(d)** $t = 7\,\mathrm{s}$

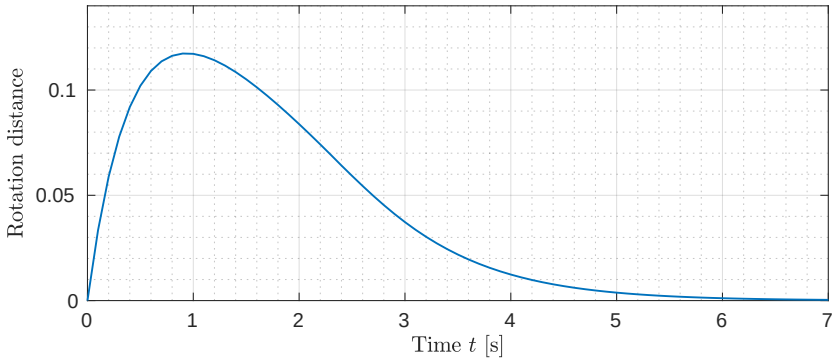
Figure 3.7: Coopertive transportation snapshots showing different tracking errors. Yellow dots are the two object estimates from the agents.

According to the algorithm (phase 4, *Leader control and prediction*), given as object trajectory a constant set-point, the ground robot, which is designated to be the leader agent, convert it to the corresponding pose of its end effector, defining its own desired trajectory, which will track employing its MPCs. As said, the MPC other than providing the control input, it yields the predicted trajectory for its end effector, which will be supplied to the aerial robot (the follower), transformed by $\hat{\boldsymbol{T}}_{e,a}^{e,g}$ (phase 5, *Follower Tracking*). The aerial robot will then compute a prediction that minimize the tracking error with its MPC.

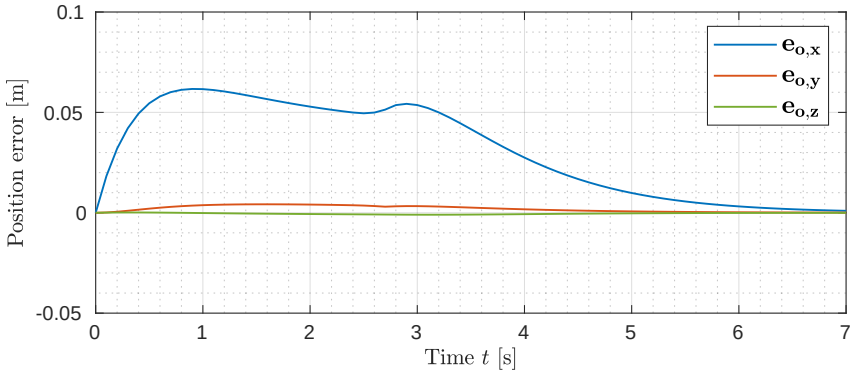
To test the algorithm we set an object reference with the same orientation and centered in $\boldsymbol{p}_{o,des} = [1, 0, 0.25]$. Figure 3.7 shows the robot configuration in four time instants and the results graph are reported in Fig. 3.8. From both figures we can notice that the follower is able to track the leader with an error. To improve the tracking accuracy, in this experiment the ground maximum velocity is limited by a constraint in the MPC to $0.3\,\mathrm{m/s}$, as can be noticed in Fig. 3.8a in the slope of the x error component.



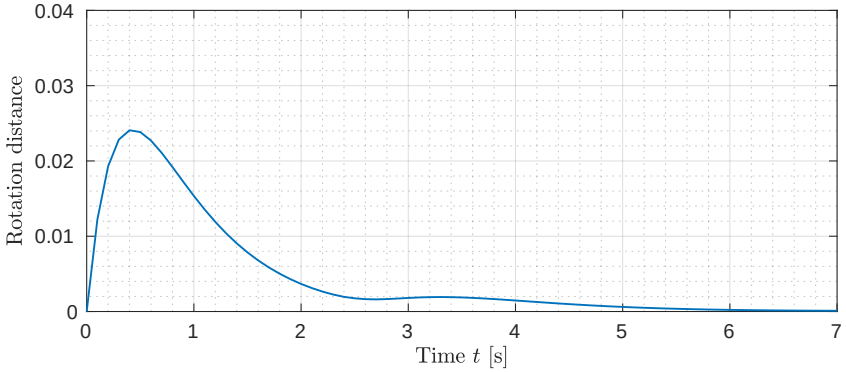
(a) Leader position error



(b) Leader orientation error



(c) Follower tracking error



(d) Follower orientation error

Figure 3.8: Cooperative manipulation error results

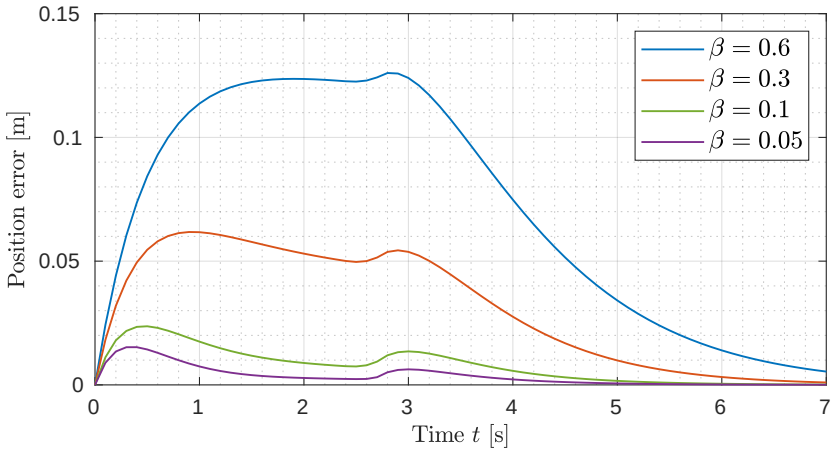
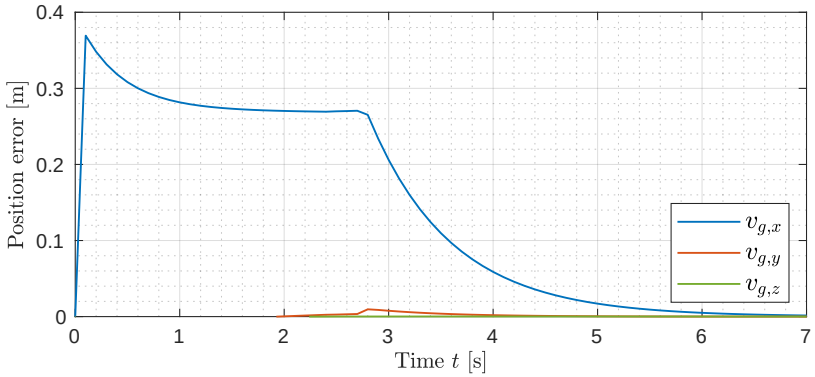


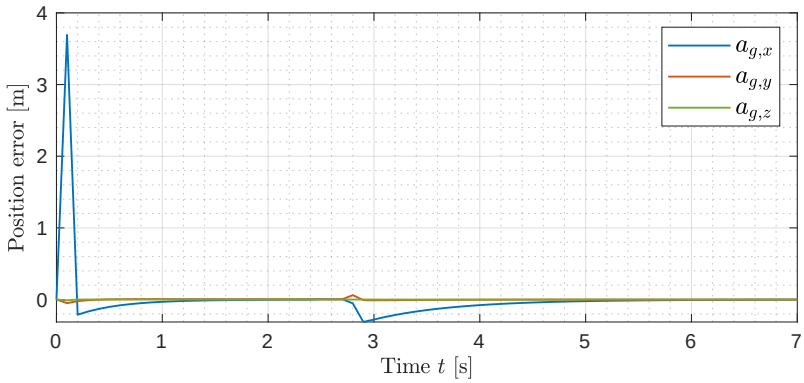
Figure 3.9: Follower tracking error comparison with different weights for control input, fixed the error weight as unitary.

In this pure kinematic simulation the tracking error can be made almost arbitarily small (with limitations given by sample time) by tweaking the control input cost weight, say β. In Fig. 3.9 are reported the tracking errors with different values for β while mantaining unitary the weight on the error cost. Note that by halving from 0.6 to 0.3 the difference is much higher than from 0.1 to 0.05 and this suggests the presence of a limit in the minimum achievable error, for which however is not really interesting to investigate since the theoretical minimum is surely way less than what is practicable in a real environment.

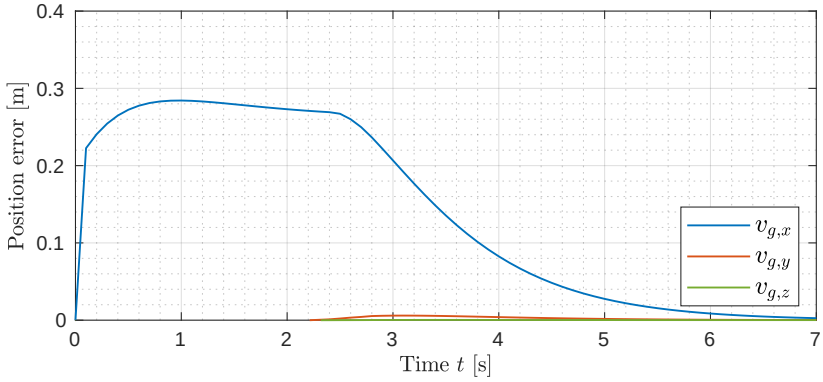
By comparing Fig. 3.9 with Fig. 3.10 we notice that the error is correlated with both velocity and acceleration. In the first instants, the ground MPC controller supplies to the system a non-zero velocity which is istantaneously tracked by the system, producing an impulse in the acceleration. The follower's MPC, to satisfy the optimal control problem, will apply a velocity which is similiar to that of the ground robot according to the weights on the cost function. However, the assumptions on the control input in the real robot are not met and a too high control input may cause instabilities and undesired oscillations, so the weights cannot be too small. Also, the initial impulsive acceleration has no effects in this simulation since second order dynamics is not considered. However, it will have a critical impact in the experiments, as described in the next sections.



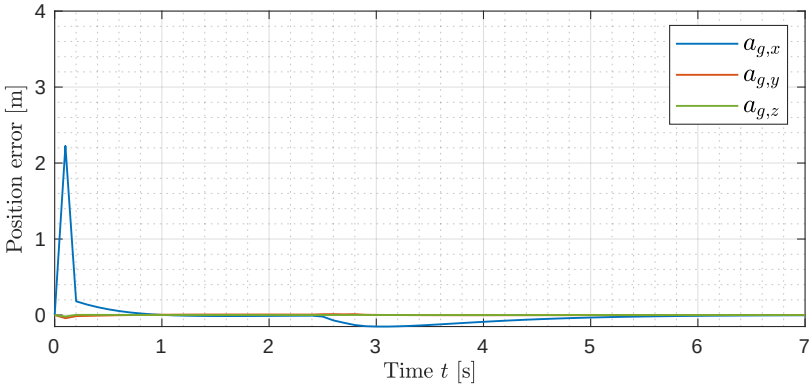
(a) Ground end effector velocity



(b) Ground end effector acceleration

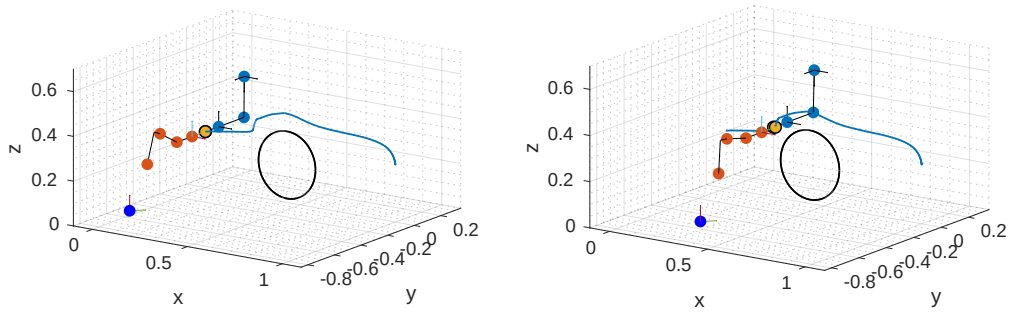


(c) Aerial end effector velocity

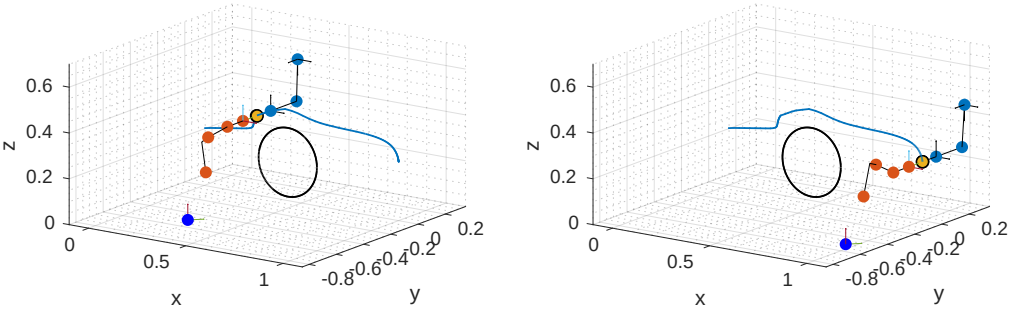


(d) Aerial end effector acceleration

Figure 3.10: Velocity and acceleration comparison for ground and aerial robot



(a) $t = 0\,\mathrm{s}$ **(b)** $t = 1.5\,\mathrm{s}$



(c) $t = 3\,\mathrm{s}$ **(d)** $t = 10\,\mathrm{s}$

Figure 3.12: Coopertive transportation snapshots showing different tracking errors. Yellow dots are the two object estimates from the agents. The black circle represents a view of the obstacle

3.4.1 Obstacle Avoidance Test

Here we will present the results of the collision avoidance strategy presented above. We consider a spherical obstacle centered in $\boldsymbol{c} = [0.5, -0.1, 0.2]$ with radius $r = 0.15$ and we run the cooperative transportation algorithm with high cost for tracking error and with the same setpoint as before $\boldsymbol{p}_{o,des} = [1, 0, 0.25]$, as depicted in Fig. 3.12.

From that figure, along with 3.11, we notice that the algorithm is able to overcome the obstacle by passing over it. Finally, figure 3.13 shows that the algorithm is slow, due to the vertical movement which is expensive and also the controller is able to keep the tracking error bounded but not small during the overtake.

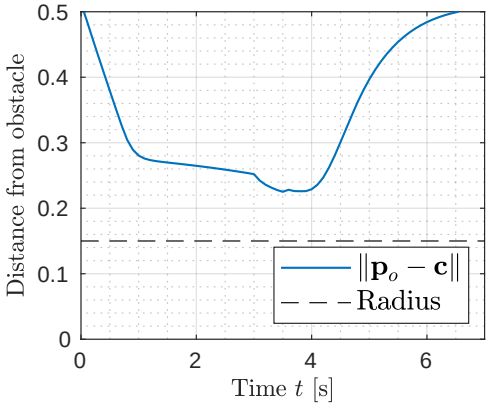
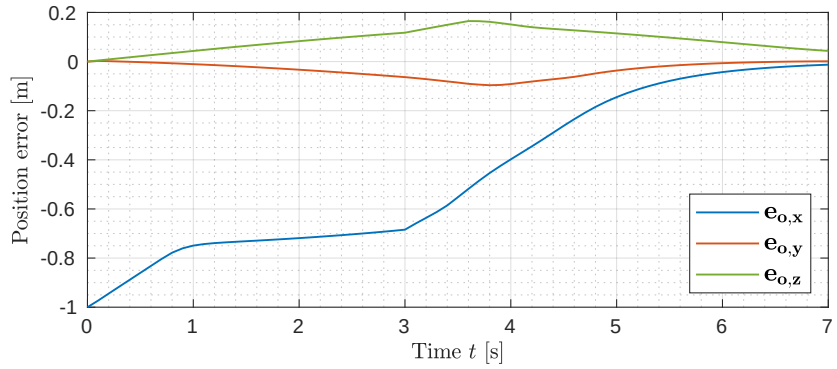
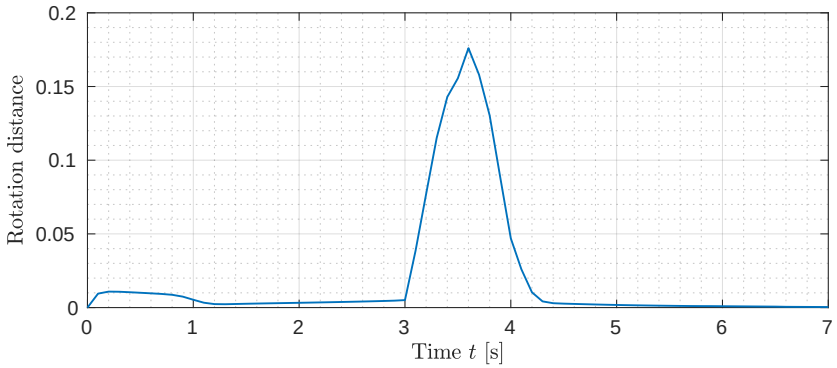


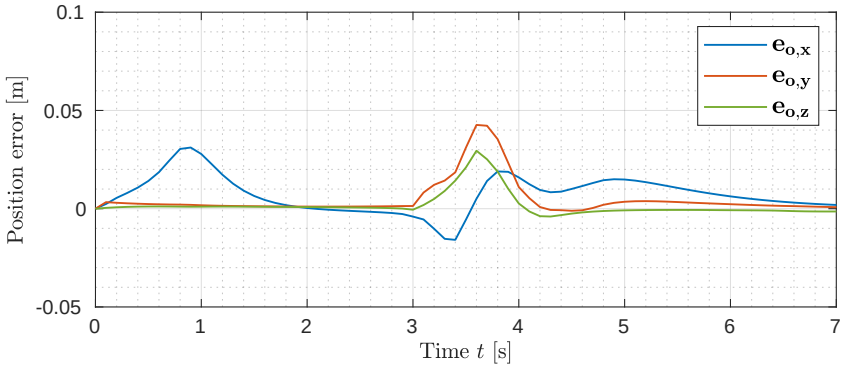
Figure 3.11: Distance between object and obstacle center.



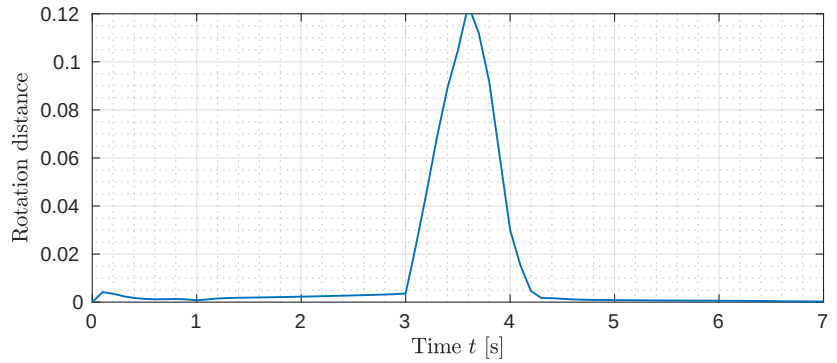
(a) Leader position error



(b) Leader orientation error



(c) Follower tracking error



(d) Follower orientation error

Figure 3.13: Cooperative manipulation error results with an obstacle

4 Simulation Environment Setup

The Simulink tests are an useful tool at the design stage that allows to validate the algorithm without accounting for side effects and thus to discern between theoretical and practical problems. However, the assumption to have perfect actuators that tracks the velocity input with zero error is rather harsh and the models employed in the MPCs need to be tested in a more realistic simulation to actually validate the algorithm. This is crucial for the aerial robot in particular, in which we neglected the underactuated dynamics. To this aim, the algorithm is implemented in C++ with a ROS interface and tested in Gazebo, a robot dynamics simulator. The flexibility in the implementation provided by ROS allows to directly perform the experiment in a flying arena with physical robots by only changing the input-output connections.

4.1 Introduction to ROS

The *Robotic Operating System* is a framework for robot software development. Besides a set of libraries, it provides services for communication between processes which is transparent with respect to the medium so that they can interact regardless being in the same physical device or across the network. In particular, this allows for hardware abstraction by providing a standardized communication format with wich processes communicate with devices.

The framework is characterized by a graph architecture, where processes are represented by the *nodes*, which can send and receive messages to and from other processes or devices through abstract communication channels referred to as *topics*. A topic is a stream of messages with a defined type that carry data of any kind, such as sensor data and actuation input values as well as internal messages between processes.

Topics implement a *publish/subscribe* communication mechanism in which nodes, before the transmission, open a topic by declaring its name and message type. Then they can start to send (*publish*) data on the topic. Nodes

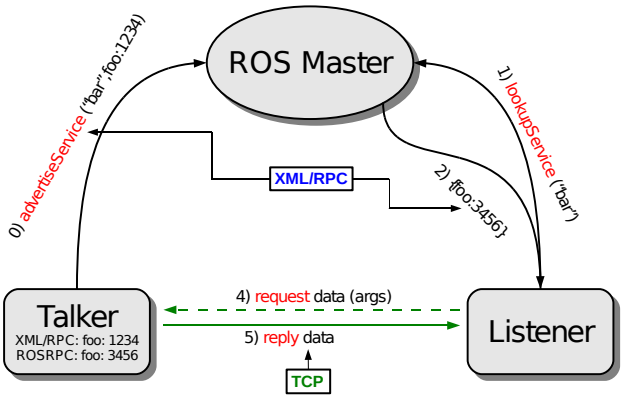


Figure 4.1: ROS mechanics scheme

that want to receive those messages can *subscribe* to that topic and then the will receive all the messages sent to that topic. The network is governed by a *master* node, the `roscore`, that manages the connections. Topics are anonymous, in the sense that the sender does not need to know which nodes are going to receive its mesages and receivers do not know the source of messages.

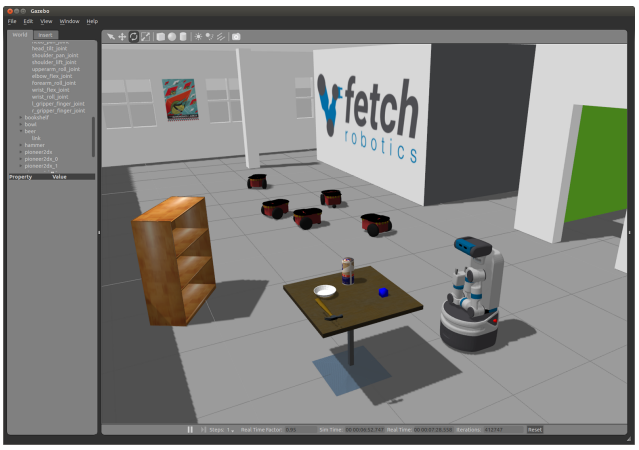


Figure 4.2: Typical screenshot of the Gazebo simulator [22]

4.1.1 Gazebo

Gazebo is a 3D dynamics simulator that allows for robot simulation. It uses the Open Dynamics Engin for rigid-body physics, in which object are assumed to be incompressible. While this assumption drastically improves the computational performance, it requires a set of tricks to solve inconsistencies that may be caused by approximations. In most of the cases this will result in a physically plausible behavior and quite accurate results. The cooperative manipulation task presented in this thesis, however, would require the

simulation of the effects of the internal forces on the object caused by each manipulator. However, the latter can only be simulated with a rigid grasp, so we will validate the algorithm by looking at the displacement error on the follower with the object detached from the robot.

4.2 Building the Gazebo models

The model for Gazebo are defined with *Unified Robot Descripion Format* (URDF), an XML format that allow to respresent the kinematic and dynamic description of the robot, along with its visual and collision representation.

4.2.1 URDF format

An URDF file consists in a root node that contains all the robot definitions:

```
<robot name="[model_name]">
    ...
</robot>
```

The robot is described via the tags `<link>` and `<joint>`

Link

The `<link>` tag defines a single link of a robot. It can describe its visual, collision and dynamics properties and is structured as follows:

```
<link name="[link_name]">
    <visual> ... </visual>
    <collision> ... </collision>
    <inertial> ... </inertial>
</link>
```

Specifically:

- `<visual>` defines how the link will be visualized in Gazebo. It could be a geometric shape defined directly with a XML tag or a mesh. It is not accounted in the physics computations
- `<collision>` defines the collision shape. Although in general it should be equal to the visual, to simplify the calculations it is usually a simpler shape, such as a bounding box or ellipsoid, that approximate the visual.

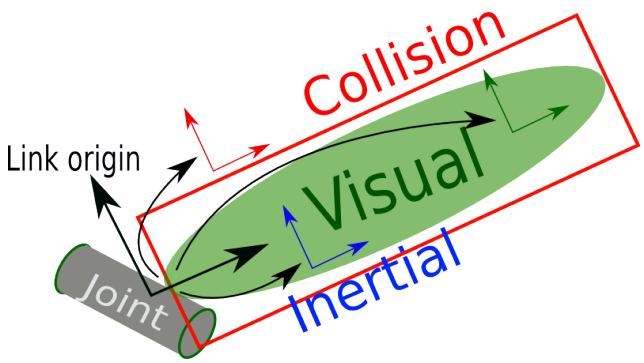


Figure 4.3: Link properties

- `<inertial>` defines the dynamic properties: the mass, inertia matrix and center of mass.

Joint

The `<joint>` tag represents a robot joint between two links. It describes the kinematics and dynamics and also specifies safety limits. The tag must be defined as follows:

```
<joint name="[joint_name]"
        type="[joint type]">
    <parent link="[parent_link]" />
    <child link="[child_link]" />

    <axis xyz="[joint axis]" />
    <origin xyz="..." rpy="..." />
    <dynamics ... />
    <limit ... />
</joint>
```

Specifically:

- `type` specifies the joint type, which can be *revolute*, *prismatic*, *fixed*, etc.
- `<parent>` and `<child>` specify the two links that the joint connects.
- `<axis>` specifies the joint axis in the joint frame.
- `<dynamics>` defines physical properties of *damping* and *friction*.
- `<limit>` defines the joint limit values in position, effort and velocity.

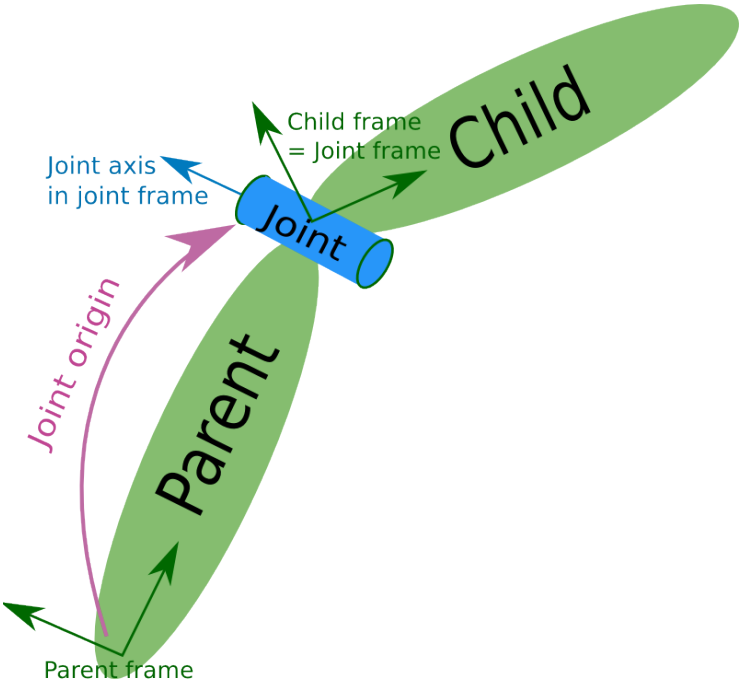


Figure 4.4: Joint properties

4.2.2 Case-study models

The file format described above can be employed to define the robot models to be simulated in Gazebo.

Ground robot model

The ground robot is composed of a prism-shaped base with four omnidirectional wheels (RB-Nex-03 from *Nexus Robot*). To define the model, we first consider a link that defines the base with the same size as the real one, `base_link`. Then, the omnidirectional wheels are simulated with two overlaid (continuous) revolute joints, each accounting for either a movement on x or y axis.

As regards the arm, the WidowX arm from Trossen, the joints are simply defined providing the geometric characteristics and by using a mesh obtained from the manufactorers. The arm has an additional wrist joint which is not considered in this work and is then substituted with a fixed joint. The resulting URDF tree is schematized in figure 4.5. Figure 4.6 shows the Gazebo rendering of the model here defined.

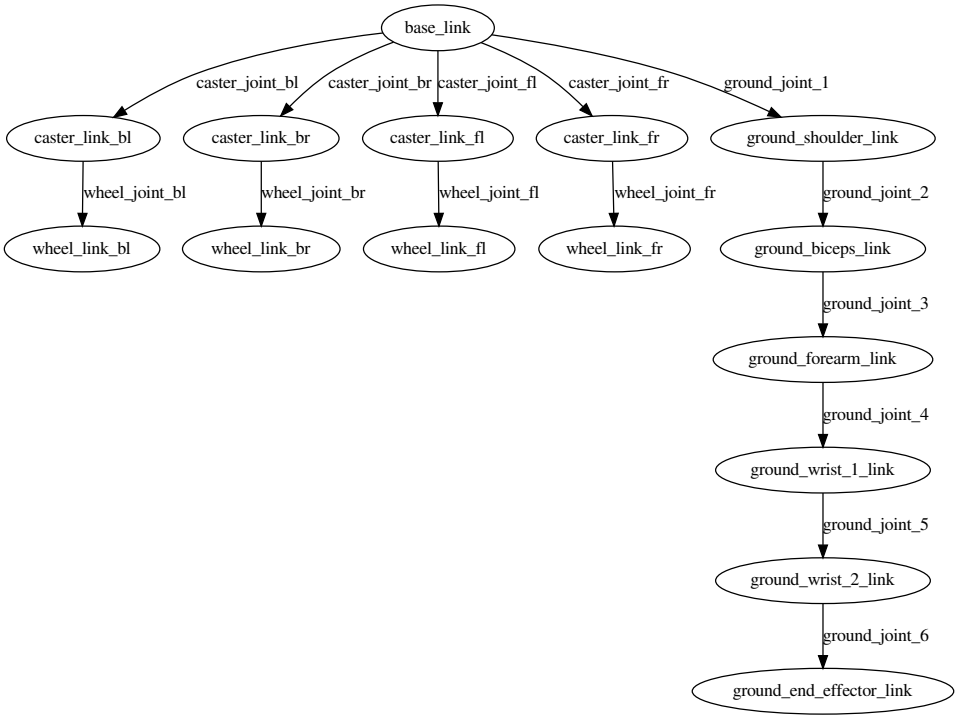


Figure 4.5: Ground robot URDF hierarchy tree

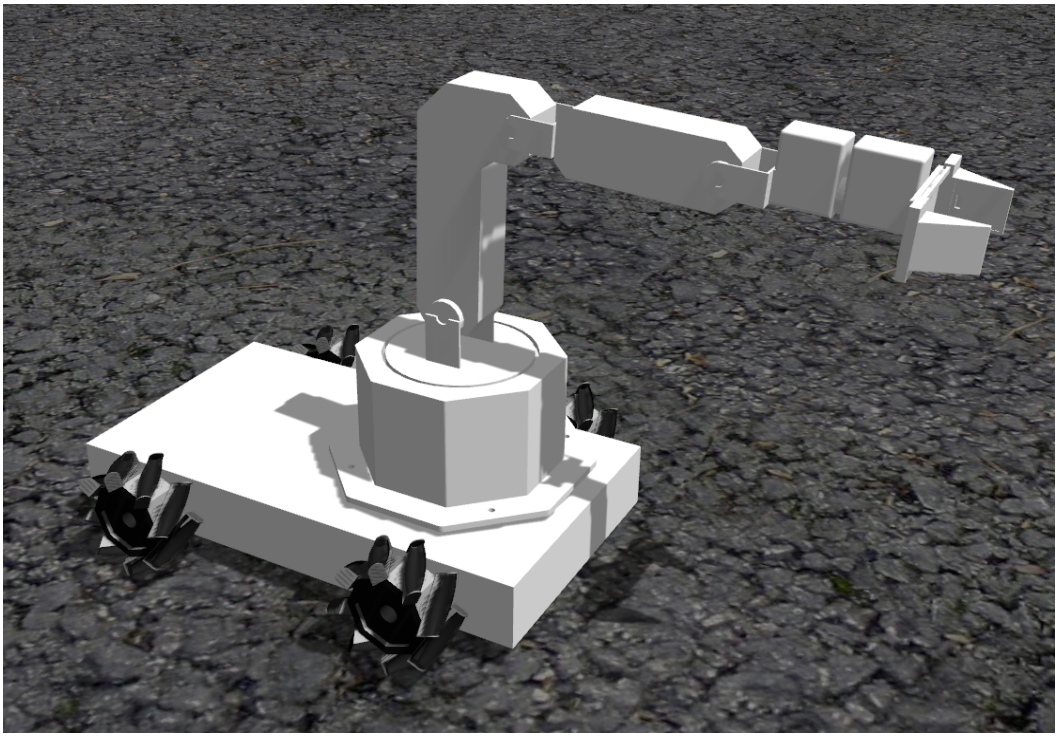


Figure 4.6: Ground robot Gazebo Rendering

Aerial robot model

The aerial robot used in the experiments is a costum assembled drone available in the Smart Mobility Lab in KTH, with six propellers. Since the control design is at high level and does not account for particular dynamics properties, for the simulations an existing URDF description of a similiar hexacopter is used. The used model is from `rotorS`, a gazebo simulator package, which will be employed for the flying dynamics simulation.

Similiarly to that of the ground robot, the URDF model file defines a `base_link` that accounts for all the aerial body structure, to which the six rotors are attached via continuous revolute joints. To the base frame also the two-joints arm is attached with a fixed joint. All the meshes for the vehicles are taken from the `rotorS` package, while the arm is the same used for the ground.

In figure 4.7 is schematized the tree strucuture of the URDF file, while the Gazebo rendering is shown in 4.8.

Low-level control and sensors

After having defined the robots description, it is necessary to set up the control and sensor interface with Gazebo that shall accept inputs and provide feedback via ROS topic messages, which is made possible by several plugins:

- The ground mobile base is controlled with the Gazebo Planar Move plugin, which accept a `geometry_msgs/Twist` message type. The plugin also provides the state information on pose and velocity with message `nav_msgs/Odometry`.
- The arms, both for ground and aerial robot, are controlled in velocity thanks to the Gazebo ROS Control plugin, which accept each joint velocity in a different topic. The same plugin provides the joint state information via other topics. The real robot interface reads all joints inputs and provides state outputs both in topic of type `std_msgs/Float32MultiArray` for all joints, so a simple ROS node to interface between the two messages type has to be written in order to mantain compatibility.
- The package `rotorS` contains a Gazebo controller plugin to simulate and control the flying vehicle. Experimental tests showed that con-

trolling the latter in velocity results in poor performances, so we will control the roll-pitch-yaw-thrust of the vehicle with message of type `mav_msgs/RollPitchYawrateThrust`, as will be explained in details later.

- Feedback data about the aerial vehicle pose and velocity is provided by `rotorS` Gazebo Odometry plugin with a `nav_msgs/Odometry` message type. The same plugin is employed to obtain the pose of the ground robot end effector. The one of the aerial robot cannot be acquired in a compatible way in the real environment, as explained later, so the end effector pose is derived by direct kinematics in a dedicated ROS node that has been purposely written.

This is the input-output message interface which is chosen to be the same in the simulation and with the real robots, so that switching from one to the other does not require changing the ROS node but only the configuration files.

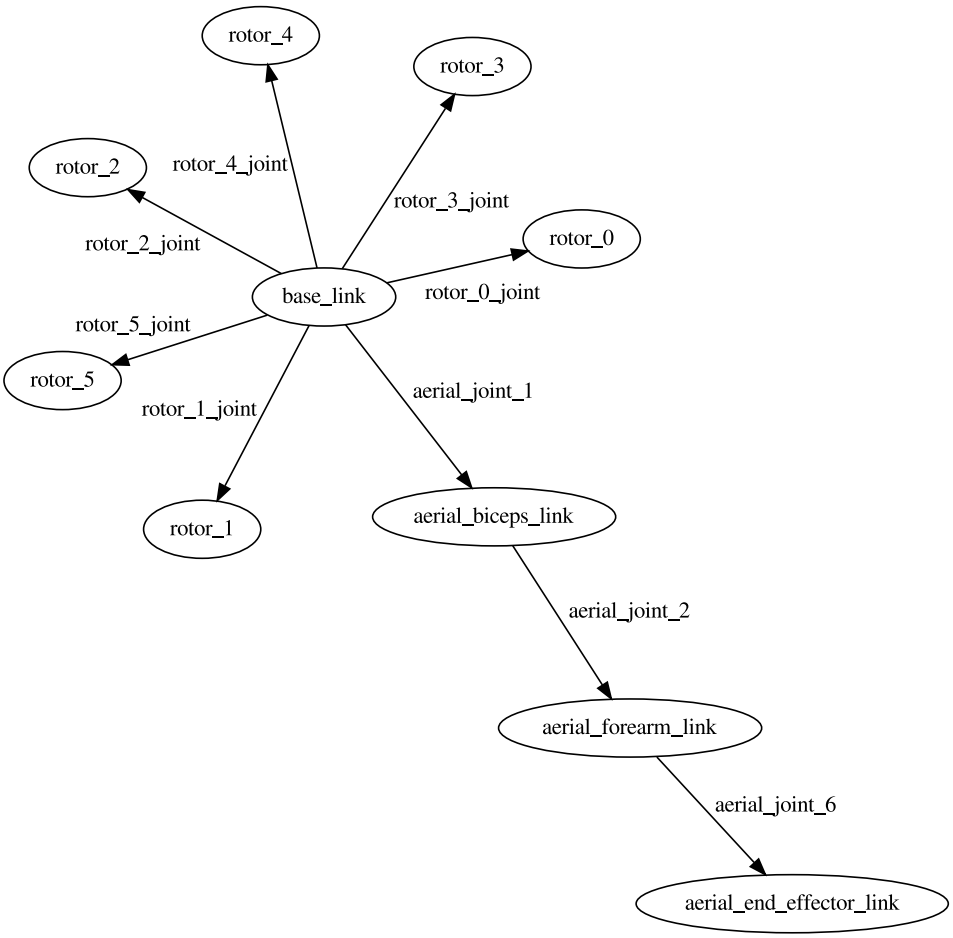


Figure 4.7: Aerial robot URDF hierarchy tree

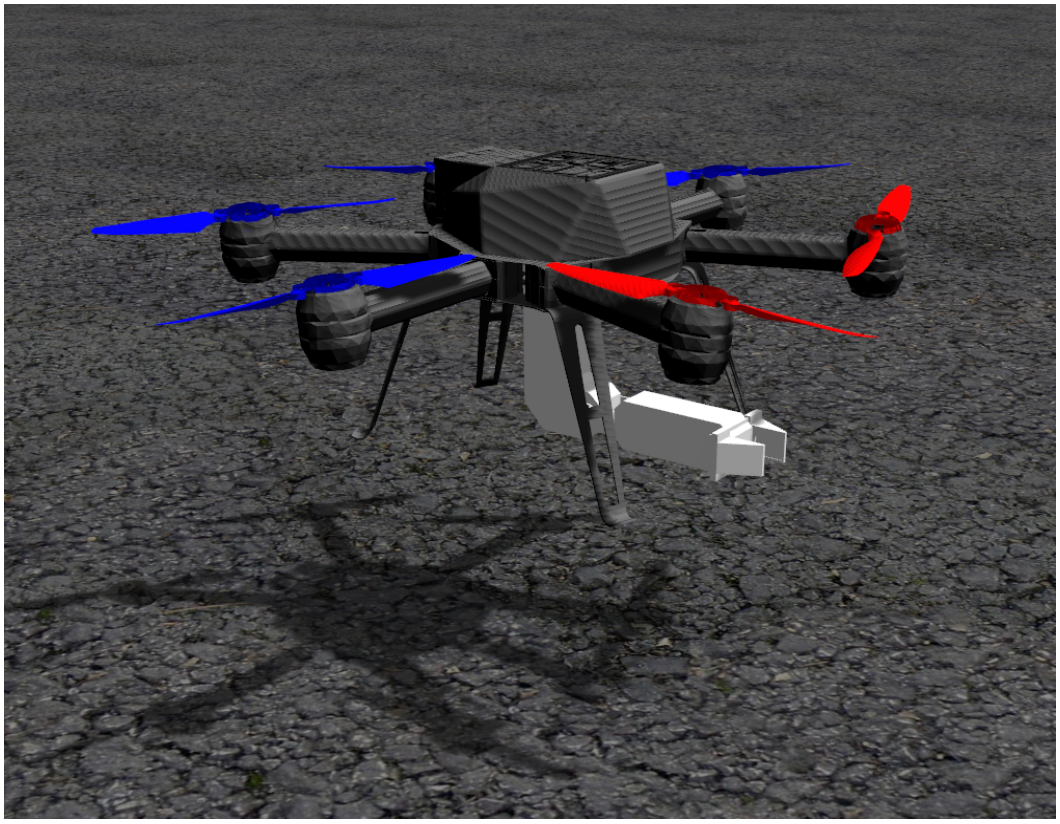


Figure 4.8: Aerial robot Gazebo Rendering

4.2.3 Gripper simulation with *EasyGripper*

It is well known in the online community that Gazebo is not able to properly simulate the physics of grasps. Also, defining or adapting an algorithm to perform grasps with the case-study grippers is a work that is out of the scope of this project since the initial assumption is that the agents are already grasping the object. However, is convenient to be able to programmatically attach and detach the object to and from the end effectors. To this aim *EasyGripper* has been written, a Gazebo plugin that is able introduce or remove a fixed joint between two links upon request.

The request can be sent on a configurable topic (`/easy_gripper/command` by default) according to the following costum message:

```
easy_gripper/GripCommand:
    string gripper_link
    string object_link
    int8 command
```

where

- `gripper_link` is the string identifying the end-effector link, as defined in the URDF file.
- `object_link` is the string identifying the object link, as defined in the URDF file.
- `command` is an integer specifying the command, which can be:
  - `0`, `command_attach`: to introduce a fixed joint between `gripper_link` and `object_link`. *EasyGripper* will keep track of such joint by storing a pair defined by the two strings. Note that this pair must be unique.
  - `1`, `command_detach`: to remove the joint between `gripper_link` and `object_link`. The pair must be previously stored with a previous `command_attach` or an error will occur.
  - `2`, `command_print`: this will print out the list of active grasps on the terminal the plugin is originally run. This is meant for debug only.

The plugin will compute the relative pose of the two links at the moment of request, which will be then onward kept constant. The simulated joint is perfectly rigid: this may cause inconsistencies when two rigid bodies are attached to the same object. This is caused both by an intrinsic problem of coherence between two bodies exerting a force on a common one and by the inaccuracies in the simulation that Gazebo introduces to keep the simulation simple.

The specific case of implementation is particularly critical since the aerial robot will not be able to roll and pitch. For this reason, the grasp on the ground robot side only will be simulated to assess the performances. However, once verified the performances of the algorithm by considering the position error, both grasps can be simulated. This will produce internal forces on the object, which the simulator is not able to properly report, but it is possible to check visually if there are inconsistencies in the simulation. However, some results may be altered by simulation inaccuracies introduced by Gazebo so the behavior must be carefully analyzed each time.

5 The Universal MPC Wrapper for ROS

Nowadays, most of the MPC implementations have an application-specific design. This means that almost each time the model is modified the code has to be updated. Here we will present a flexible MPC implementation in ROS that is suitable to be used with every possible model defined with ACADO, a dynamic optimization library whose generation tool the software relies on. The versatility lies in the standardized I/O ROS messages that is applicable to any kind of input and outputs

The working principle is the following:

- First, the problem is defined in MATLAB employing the library provided by ACADO.
- Then, the tool will produce C code that solves the problem.
- The C++ code for the ROS node is designed to work with every possible code generated by ACADO.
- Finally, the ROS code is compiled together with the ACADO generated code that is placed in a predefined directory.

This procedure will produce a ROS node that accepts in an input topic all the data needed for each computation step and provides in output all the computed data.

5.1 Matlab problem definition with ACADO

`ACADO Toolkit` is a software environment and algorithm collection written in C++ for au- tomatic control and dynamic optimization. It provides a general framework for using a great variety of algorithms for direct optimal control, including model predictive control as well as state and parameter estimation.

The ACADO Code Generation tool allows to export C code to solve nonlinear model predictive control of the form:

$$\min_{\substack{\boldsymbol{x}_0,\ldots,\boldsymbol{x}_N \\ \boldsymbol{u}_0,\ldots,\boldsymbol{u}_{N-1}}} \quad \sum_{k=0}^{N-1} \left\| \boldsymbol{h}(\boldsymbol{x}_k,\boldsymbol{u}_k,\boldsymbol{d}_k) - \boldsymbol{y}_{ref,k} \right\|^2_{\boldsymbol{W}_k} + \left\| \boldsymbol{h}_N(\boldsymbol{x}_N) - \boldsymbol{y}_{ref,N} \right\|^2_{\boldsymbol{W}_N}$$

$$\begin{aligned}
\text{subject to:} \quad & \boldsymbol{x}_0 = \hat{\boldsymbol{x}}_0 \\
& \boldsymbol{x}_{k+1} = \boldsymbol{F}(\boldsymbol{x}_k,\boldsymbol{u}_k,\boldsymbol{z}_k), \quad k = 0,\ldots,N-1 \\
& \boldsymbol{x}_{k,min} \leq \boldsymbol{x}_k \leq \boldsymbol{x}_{k,max}, \quad k = 0,\ldots,N \\
& \boldsymbol{u}_{k,min} \leq \boldsymbol{u}_k \leq \boldsymbol{u}_{k,max}, \quad k = 0,\ldots,N \\
& \boldsymbol{r}_{k,min} \leq \boldsymbol{r}_k(\boldsymbol{x}_k,\boldsymbol{u}_k) \leq \boldsymbol{r}_{k,max}. \quad k = 0,\ldots,N-1 \\
& \boldsymbol{r}_{N,min} \leq \boldsymbol{r}_k(\boldsymbol{x}_N) \leq \boldsymbol{r}_{N,max}
\end{aligned} \tag{5.1}$$

where:

- $\boldsymbol{x}_k \in \mathbb{R}^{N_x}$ is the differential state
- $\boldsymbol{u}_k \in \mathbb{R}^{N_u}$ is the control input
- $\boldsymbol{z}_k \in \mathbb{R}^{N_z}$ is a vector of algebraic variables
- $\hat{\boldsymbol{x}}_0 \in \mathbb{R}^{N_x}$ is the current state measurement

Reference functions $\boldsymbol{h}(\cdot,\cdot,\cdot) \in \mathbb{R}^{N_y}$ and $\boldsymbol{h}(\cdot,\cdot) \in \mathbb{R}^{N_{y,N}}$ are the output reference functions. Running and terminal weighting matrices are denoted with $\boldsymbol{W}_k \in \mathbb{R}^{N_y \times N_y}$ and $\boldsymbol{W}_N \in \mathbb{R}^{N_{y,N} \times N_{y,N}}$. Variables $\tilde{\boldsymbol{y}}_k \in \mathbb{R}^{N_y}$ and $\tilde{\boldsymbol{y}}_N \in \mathbb{R}^{N_{y,N}}$ denote time varying references. $[\boldsymbol{x}_{k,min}, \boldsymbol{x}_{k,max}]$ and $[\boldsymbol{u}_{k,min}, \boldsymbol{u}_{k,max}]$ are the state and control inputs ranges, that might change along the horizon. The last two terms define path and point constraint, respectively, with constraint functions $\boldsymbol{r}_k \in \mathbb{R}^{N_{r,k}}$ and $\boldsymbol{r}_N \in \mathbb{R}^{N_{r,N}}$. Function $\boldsymbol{F}(\boldsymbol{x},\boldsymbol{u},\boldsymbol{z})$ defines a discretized ODE, automatically generated by ACADO given the continuous time system.

ACADO Implemented Algorithms

The exported C code solves nonlinear MPC problems by means of the real-time iteration scheme with Gauss-Newton Hessian approximation. Discretization of the continuous time ODE is done via shooting techniques [11]. The resulting quadratic problem is solved with qpOASES [4, 5].

5.1.1 ACADO OCP definitions

The problem definition is coded in a MATLAB script. As a general example that explains the ACADO syntax, in the following we will present the description of the ground robot, following the definition provided in sec. 3.1. The aerial one is completely analogous.

First, differential states and controls are to be declared.

```
DifferentialState   p_e(3) R_e_vec(9);
DifferentialState   th1 th2 th3 th4 x_v y_v psi_v;
Control             uth1 uth2 uth3 uth4 ux uy upsi;
```

where `p_e` is a 3D vector representing the end effector position and `R_e_vec` is a 9-components vector containing the components of its rotation matrix. In the second line we define the joints variables and the pose of the vehicle, with clear reference to what defined in sec. 3.1. Finally, the third line defines the control input.

ACADO allows to define *online data*, which is a vector as long as the prediction window and is supplied at each computation step. We will use this to input the obstacle data and optionally the keypoints as well as the reference end-effector orientation, since computing the error requires a computation that cannot be resorted to a difference as in (5.1).

```
OnlineData R_e_des_vec(9);
OnlineData v_o_1(3) r_1;
OnlineData v_o_2(3) r_2;
OnlineData keypts(3*K);
```

where `R_e_des_vec` is the reference rotation matrix of the end-effector frame, `v_o_1`, `v_o_2` and `r_1`, `r_2` defines the center and the radius of two obstacles. `keypts` is the array of `K` keypoints to be considered, with `K` hardcoded.

By exploiting the MATLAB *Simbolic Toolbox*, it is possible to derive the parametric functions needed to define the differential equation describing the kinematics of the end effector pose:

- `[J_P, J_O] = getJacobian (q, groundParams)` returns the position and orientation Jacobians of the ground model, as defined in sec. 3.1, given `q`, containing the current joint state and vehicle pose, and the set of link lenghts and geometric displacement in the ground robot in `groundParams`.

- `B = getInputMatrix (q)` represents the input matrix $\boldsymbol{A}_{p,g}$ defining the input constraints as in (3.5).

When dealing with matrix it is necessary to account for the conversion from a 3×3 matrix to its 9-D vector representation. In the following we assume this conversion implicit by considering a variable with `_vec` appended, and viceversa. The rotational dynamics can then defined through the auxiliary variables:

```
omega_e = J_O * u;
R_e_dot = skew (omega_e) * R_e;
```

where `omega_e` is the instantaneous angular acceleration given the input `u`, which contains all the control variables, and `skew` which yield the skew symmetric matrix.

The overall agent dynamics is defined in the ACADO library syntax by means of the function `dot`, which informs the processor that the expression in the argument shoud be treated as a derivative. The ODE definition is then intuitive:

```
f = [dot (p_e) == J_P * u; ...
     dot (R_e_vec) == R_e_dot_vec; ...
     dot (q) == B * u];
```

The output function $\boldsymbol{h}(\cdot,\cdot,\cdot)$ is defined as in section 2.2.

The controlled quantity is the end effector position `p_e`, which can be compared directly to the reference in (5.1). The orientation error, instead, needs a preliminar computation:

```
Err_R_e = norm (R_e' * R_e_des - eye(3)),'fro');
```

which is the Frobenius distance between rotation difference and the identity.

To implement obstacle avoidance, we insert in the output function the cost defined in (2.45), where $\chi_i(\cdot)$ is implemented in a function `costCollide (p, obst, r, margin)`, where,in the specific case, `obst` is `v_o_1` and `v_o_2`, `margin` is choosen to be $0.1\,\mathrm{m}$ and `p` is considered as:

- `p_v = [x_v; y_v; z_v]`, the base position vector.
- `p_e`, the end effector position.
- `p_2`, `p_3`, the second and third joint position, obtained with the direct kinematics via functions `getP2`, `getP3 (q, groundParams)`.

- `w_fr`, `w_fl`, `w_rr`, `w_rl`, keypoints centered in the four wheels, based on the specific robot geometry, hardcoded.
- `keypts`, optional keypoint accepted as online data (intended to be the object pose, when available, should be infinite otherwise)

The final output vector is then `hN = [p_e; total_cost; Err_R_e]` for the terminal term, and `h = [hN; controls]` for the running cost, where `total_cost` is the sum of the result of `costCollide` for the two obstacle and each keypoint defined above. `controls` is an ACADO shortcut to refer to all the control inputs.

To export the code we need first to setup ACADO (first line below), define an OCP variable and the size of the cost matrices `W` and `WN`:

```
acadoSet ('problemname','mpc_export');
ocp = acado.OCP (0.0, N*Ts, N);

W = acado.BMatrix (eye (length(h)));
WN = acado.BMatrix (eye (length(hN)));
```

given `N` to be the prediction count and `Ts` the sample time. We then setup the MPC cost function and model with:

```
ocp.minimizeLSQ (W, h);
ocp.minimizeLSQEndTerm (WN, hN);
ocp.setModel (f)
```

with clear meaning of the functions. The maximum control and joint values are set with `ocp.subjectTo (... <= var <= ...)`.

Finally, to export the code:

```
mpc = acado.OCPexport (ocp);
mpc.set ( ... );
...
mpc.set ( ... )

mpc.exportCode ( <path> )
```

where `mpc.set` will set the problem options define in table 5.1, which are rather standard. For integration with the ROS wrapper, if the script is in the `matlab` directory of the package, the target directory should be `../src/export_MPC`.

Option	Value
HESSIAN_APPROXIMATION	GAUSS_NEWTON
DISCRETIZATION_TYPE	MULTIPLE_SHOOTING
SPARSE_QP_SOLUTION	FULL_CONDENSING_N2
INTEGRATOR_TYPE	INT_IRK_GL2
NUM_INTEGRATOR_STEPS	2 * N
QP_SOLVER	QP_QPOASES3
LEVENBERG_MARQUARDT	1e-4

Table 5.1: ACADO MPC problem options

5.2 MPC Wrapper implementation

In this section we will present the implementation of the ROS MPC Wrapper node. First, we will briefly outline the ACADO generated code interface. The MPC Wrapper will employ the latter to implement the MPC in two layers: the outer layer is the ROS interface, which is responsible of communication with external ROS node by reading inputs and publishing the outputs. The inner layer is responsible of converting ROS messages to ACADO structure and to run the actual MPC solution, with the additional possibility of handling input packet loss without interrupting the control.

5.2.1 ACADO generated code interface

The ACADO generated code consists of several source files, the main ones are `acado_solver.c`, `acado_integrator.c`, which contain the C translation of the expressions defined in MATLAB and the interface for problem solution.

The main definitions, which will be used in the wrapper, are located in `acado_common.h`, that are:

- Constants indicting the exact sizes of the problem: `ACADO_N`, number of control intervals, `ACADO_NOD`, number of online data values, `ACADO_NU`, number of control variables, `ACADO_NX`, number of differential states, `ACADO_NY`, number of references per sample on the first N samples, `ACADO_NYN` number of references on the last $(N+1)$-st sample.
- `ACADOvariables`, structure containing all the variables of the problem: states, controls, online data, running and terminal reference, cost matrices and the current state feedback vector $\hat{\boldsymbol{x}}_0$.
- `acado_initializeSolver`, to initialize the solver.
- `acado_preparationStep`, to setup variables before each computation step.
- `acado_shiftStates`, and `acado_shiftControls` to shift the stored values of the states and controls. See below for details.
- `acado_feedbackStep`, main function in which the computation actually occur.

After initializing the solver, at each computation step, ACADO requires to initialize the values of state, online data and reference as well as the controls for the whole prediction horizon and this will be used as initial condition in each solution of the optimization problem. Before the first calculation, of course, no data is available for the controls, so the vector will be filled with zeros. After the first calculation, at each step a set of predicted control values is available from the previous one: the first one has already been applied, after the feedback the previous open loop control will not be optimal and the state prediction will be different according to the measured state. However, those represent a good initialization for the next optimization step, so all the control prediction and states should be shifted forward at each computation step.

5.2.2 MPC Wrapper inner layer

The inner layer, implemented as a C++ class, is responsible for handling the main MPC computation, assuming to have all the input information decoded. It is based on a finite state machine whose transition are governed by the outer layer counterpart. It is designed to work in two modes: *synchronous* and *asynchronous*.

- In *synchronous* mode the computation must be triggered (by the outer layer) at a specific rate, regardless if new measurement are available or not. If no new data is provided, the next step will output the last computed prediction, with the next open-loop input as current control, as described below.

- In *asynchronous* mode, the computation is executed at the same time new data is available, with no timing constraints. This preclude the possibility of handling data losses.

Note that timing and data updating is responsibility of the outer layer: depending on its implementation, one of the two behavior will arise.

Despite the advantages of the former mode, besides requiring a quite precise time base synchronization, it should be noticed that in case of data loss the provided prediction window is progressively shorter.

In this project, this entail different window length between leader and follower, and this may arise additional problems. For this reason, for simplicity, the asynchronous mode is chose for the specific application.

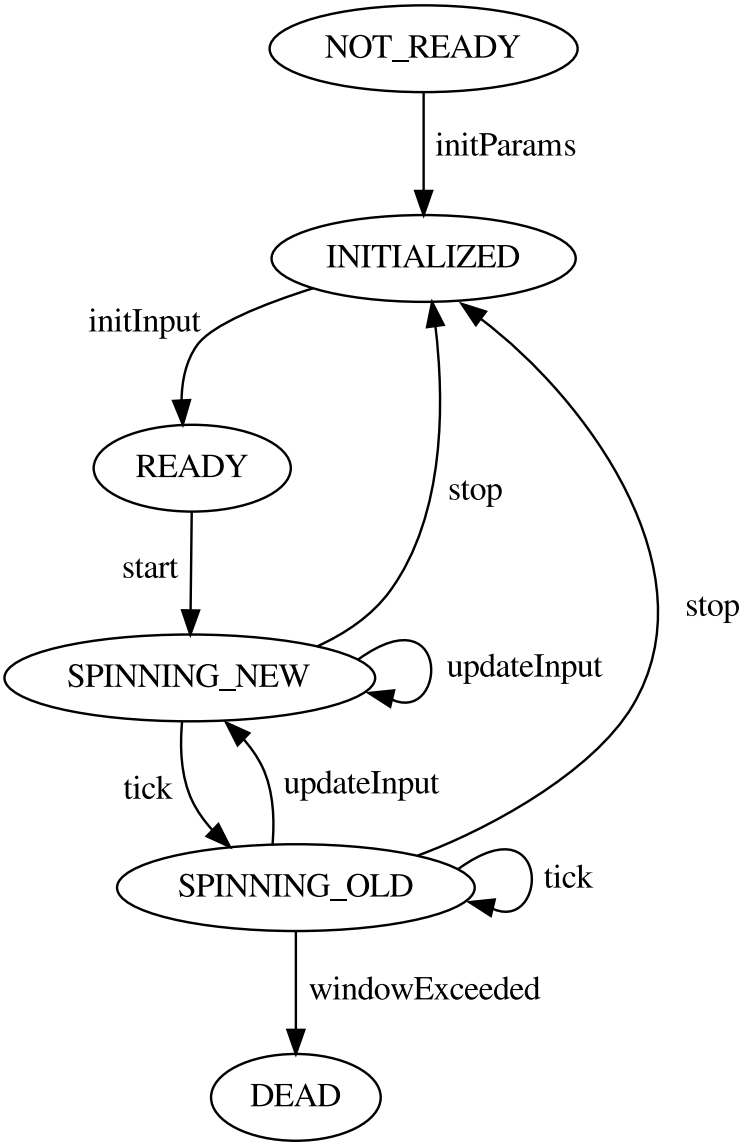


Figure 5.1: MPC Wrapper Finite State Machine diagram

The implementation of the inner layer is based on a finite state machine. The interface from and to the outer layer consists of a set of input actions (`initParams`, `initInput`, `start`, `updateInput` and `tick` and `stop`) and two data structures, `InputData` and `OutputData`, defined later.

The states and transitions are defined as follows:

1. `NOT_READY`, is the initial state when the machine is loaded. It waits until the `initParams` action is called to specify the MPC initialization:
   - Initial state $\boldsymbol{x}_0$.
   - Initial algebraic state $\boldsymbol{z}_0$, if any.

- Initial control input $\boldsymbol{u}_0$. Without any information, the default choice is a vector of zeros.
- Weight matrices for runnning and terminal cost, $\boldsymbol{W}$ and $\boldsymbol{W}_N$ respectively.
- Maximum number of iterations N_{iter}.

This will then cause the state to change to `INITIALIZED`

2. `INITIALIZED`. In this state the MPC parameters are set and the machine is waiting for the first input data of type `InputData` to be supplied through the action `initInput`.The state will subsequently change to `READY`.

3. `READY`. The MPC is ready and waits for the `start` action to be called for synchronization purpose, which will initialize the solver and trigger the computation of the first solution. The state will also change to `SPINNING_NEW`

4. `SPINNING_NEW`. New data is available and three actions can be performed:
   - `updateInput`, which will reupdate the input data, discarding the current. This is inteded to occur in case of packet loss after a small time, so that the new system state is close to the old one. Otherwise the `stop` action should be taken so to reinitialize the MPC.
   - `tick`, which will start the solution according to the current input data. After the computation, new output of type `OutputData` is available to be fetched by the outer layer. Consequently, the state will change to `SPINNING_OLD`.
   - `stop` will change the state to `INITIALIZED` to disable output and allow a reinitialization of the MPC.

5. `SPINNING_OLD`. In this state the solution for the current input data has already been computed. The same three actions can be called as in the previous state, with different results:
   - `updateInput` will return to state `SPINNING_NEW` where new data has no solution yet.

- `tick`. This can only occur in *synchronous* mode when the next control has to be computed but the new data is not available. This will keep the last solution and set the `openLoopIndex` field of the `OutputData` structure, which is initially 0, to 1, which means that the control sample to be applied is the second in the prediction array. If `tick` is called again from this state, the `openLoopIndex` will be increased, until either a new `updateInput` is triggered, which will reset the counter, or the prediction window is exceeded, which will bring the machine to the fault state `DEAD`.
- `stop` will change the state to `INITIALIZED` like before.

6. `DEAD`. The MPC prediction is over and no new data is supplied. This faulty condition cannot be handled by the MPC: a constant prediction position and zero velocity input is yield as output.

The data structures used to exchange input and output informations are the following

```
struct InputData:
    Vector state
    Matrix refWindow
    Matrix refTerminal
    Matrix onlineData
```

```
struct OutputData:
    Matrix stateTrajectory
    Matrix controlTrajectory
    int openLoopIndex
    int solverStatus
    int iterNumber
    double kktTol
    double cpuTime
    double objectiveValue
```

where:

- The fields of `InputData` contain information about current state, window and terminal reference and online data, where the rows of the matrices are relative to the different instants of the prediction window.
- The fields of `OutputData` contain the solution to the optimal control problem in `controlTrajectory` and the state prediction, both throughout the prediction window. The `openLoopIndex`, as mentioned, refers to the control sample that has to be applied according to the algorithm and is always zero in *asynchronous* mode. `solverStatus`, is the error code occurred in the solution:

  0: No error.

1: QP could not be solved within the given number of iterations.

-1: QP could not be solved due to an internal error.

-2: QP is infeasible and thus could not be solved.

-3: QP is unbounded and thus could not be solved.

-30: QP causes nan KKT.

The resulting tolerance value from the solution is stored in `kktTol`, the actual number of iterations needed for the solution is reported in `iterNumber`, the time taken for the solution in `cpuTime` and the current objective function value in `objectiveValue`.

The sizes of the vectors and matrices are defined by ACADO in the generated code: this allows to the node to be adapted to any problem size without manual modifications to the code.

5.2.3 MPC Wrapper Outer Layer

The outer layer is an actual ROS node which is responsible for both receiving and publishing data from and to the other nodes and to trigger the computation with the right timing. Currently, an hybrid synchronous and asynchronous mode has been realized, leaving more advanced implementations as future works.

Inputs and outputs

The first requirement for the ROS node to be reusable is to have a standardized input-output interface, which shall be realized through a ROS message. To this aim it is important that the structure of the message type is sufficiently flexible. The standard message `std_msgs/Float64MultiArray` perfectly suits the needs. By definition it is composed of a layout specification part and a one-dimensional array that contains all the data:

```
std_msgs/Float64MultiArray:
    MultiArrayLayout  layout    # specification of data layout
    float64[]         data      # array of data
```

A further look into the definition of `MultiArrayLayout` shows how the data into array should be fitted:

```
std_msgs/MultiArrayLayout:
    MultiArrayDimension[] dim   # Array of dimension properties
    uint32 data_offset          # padding elements at front of data

std_msgs/MultiArrayDimension:
    string label                # label of given dimension
    uint32 size                 # size of given dimension
    uint32 stride               # stride of given dimension
```

According to the documentation, to define a matrix the layout field is used as follows:

```
    matrix[i, j]  = data[data_offset + dim[1].stride * i + k]
```

The dimensions of the data must be known a priori both from the wrapper and from the receiver/sender (depending on the direction of the communication) but this is the case in any application: the wrapper is aware of the dimensions from the ACADO generated code while the external nodes are application-dependant and they are then able to provide all the necessary information.

In this light, we can define the costum messages type according which the nodes will exchange information:

```
mpc_wrapper/InputData:
    std_msgs/Float64MultiArray state
    std_msgs/Float64MultiArray refWindow
    std_msgs/Float64MultiArray refTerminal
    std_msgs/Float64MultiArray state

mpc_wrapper/OutputData:
    std_msgs/Float64MultiArray controlTrajectory
    std_msgs/Float64MultiArray stateTrajectory
    std_msgs/Float64MultiArray currentControl
    int64   openLoopIndex
    float64 kktTol
    float64 solverStatus
    float64 cpuTime
    int64   nIter
    float64 objVal
```

This to the definition is completely analogous to the data structure of the inner layer: it is then easy to convert it according to and from the given layout and supply to and receive from the inner layer.

Main loop

As mentioned, the current implementation is an hybrid of the two modes which constantly provides data to the inner layer and the procedure is schematized as follows:

1. The ROS node loads the MPC initialization parameters from the configuration file (default `config/mpc_wrapper.yaml`) which defines the parameters described above as well as the loop rate.

2. When the first input data arrives, it sets the first input and automatically starts the MPC, bringing its state to `SPINNING_NEW`.

3. Then, the main loop starts to run at constant rate while the input data is updated in parallel. The node will always update the input and trigger a `tick`, regardless that new data is actually available: if not, the old data is kept as constant.

4. Right after the solution is triggered, the output data generated by the inner layer is fetched, converted to the output ROS message and published to the external nodes.

The overall schemes for the tree implementations are schematized in figures 5.2, 5.3 and 5.4. In the scheme, the shapes have the following meaning:

- Rectangles are data storage: they latch data until is refreshed by a new input.
- Parallelograms represents actions, that are triggered by incoming arrows or may trigger other actions pointed by outcoming arrows.
- Ellipses are external and autonomous source of signals, that might emit data (e.g. *ROS IN*) or logical signals (e.g. *CLOCK*).
- Triangles are "tri-state buffers": when the lateral signal is on, it means that a data transfer from the input to the output is performed.
- Arrows coming to and from data blocks contain data signals, while those coming to and from action blocks represent logical signals.
- Diamond shapes have the common meaning of conditial action which emits logical signals if the stated condition is true.

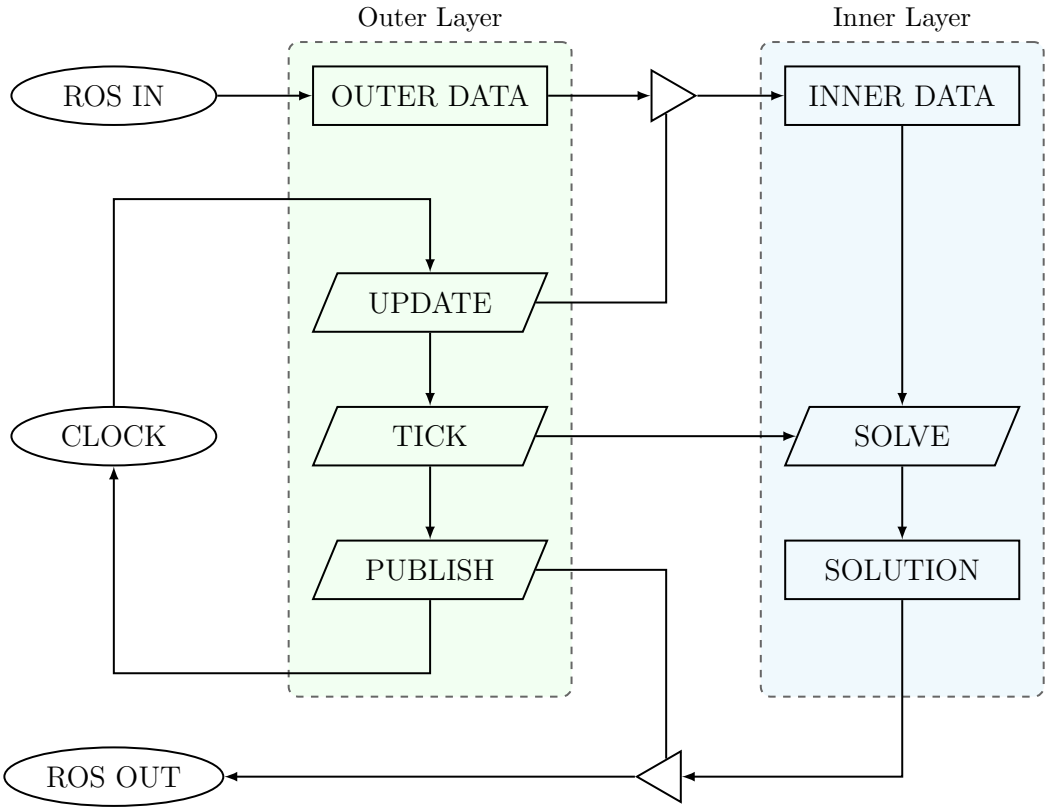


Figure 5.2: MPC Wrapper outer layer "hybrid" implementation scheme

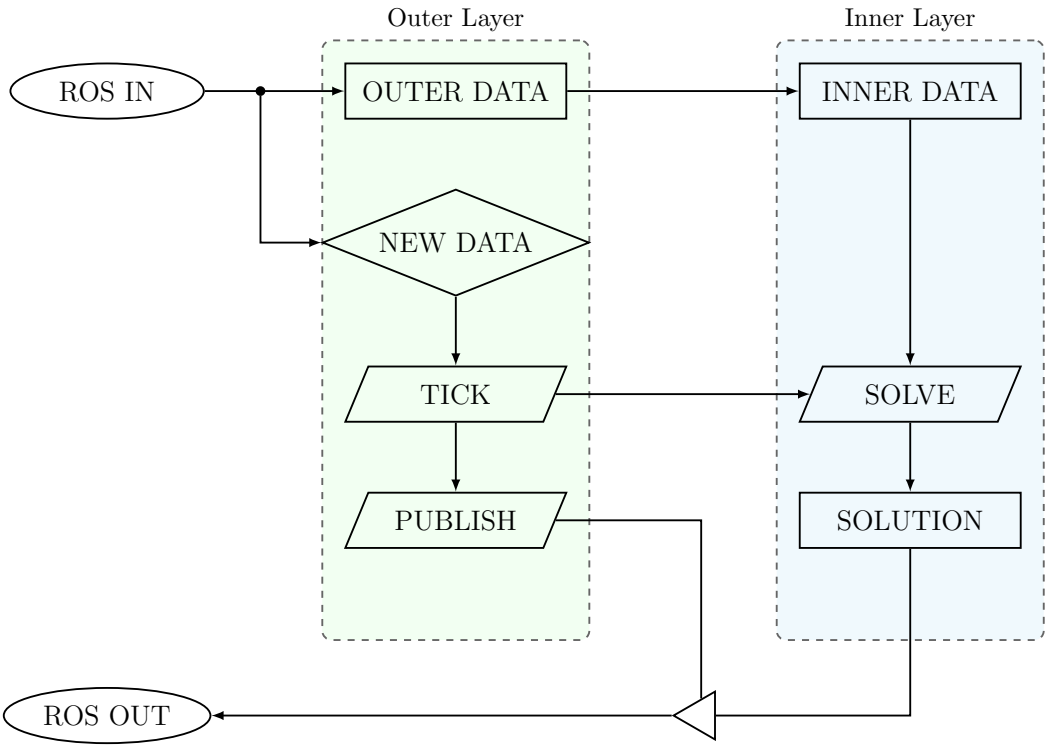


Figure 5.3: MPC Wrapper outer layer *asynchronous* implementation scheme

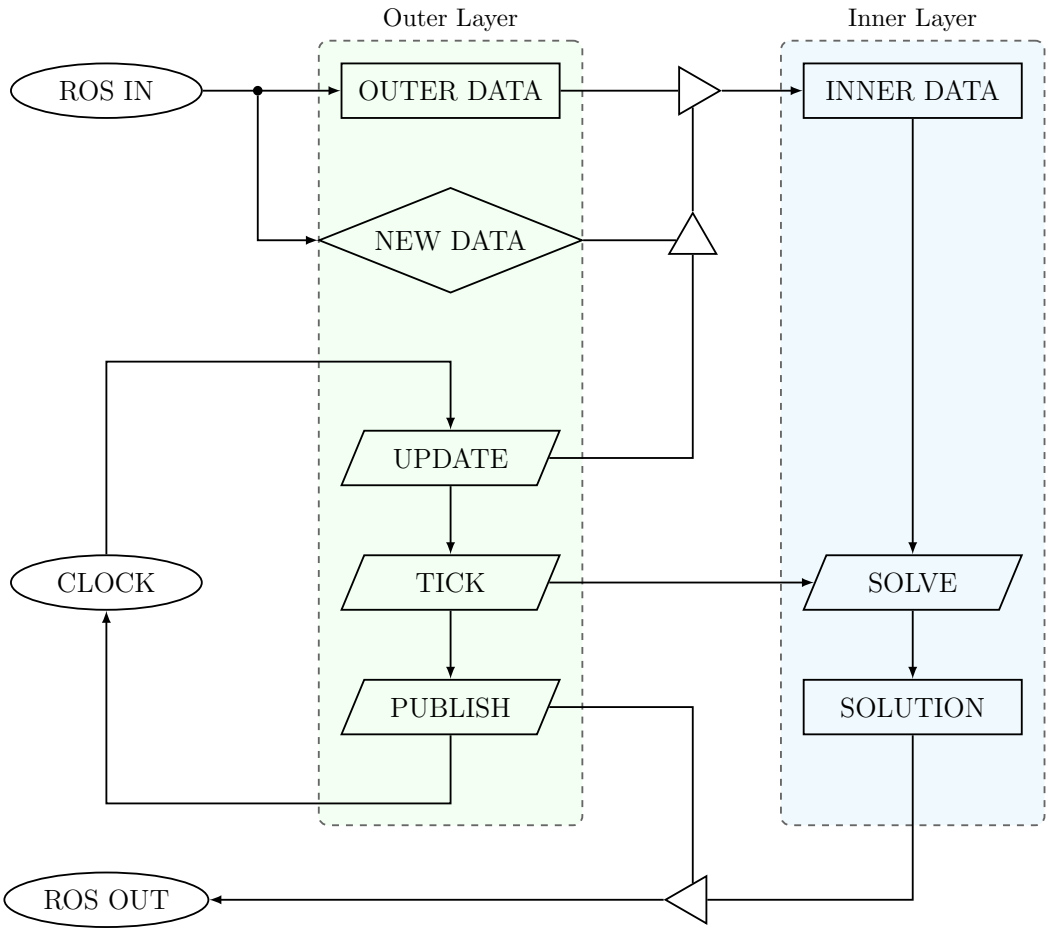


Figure 5.4: MPC Wrapper outer layer *synchronous* implementation scheme

Summary

The *Universal MPC wrapper* is a flexible tool that allows an easy implementation of a generic optimal control problem. First, the problem is defined with ACADO in a matlab script that generates the relative C code.

The MPC Wrapper is responsible of receiving data through a ROS message of the type `mpc_wrapper/InputData` described above, solving such problem according to the interface provided by the generated code, and will output the results in a ROS message of the type `mpc_wrapper/OutputData`, that contains all the resulting data.

Depending on the implementation, the MPC wrapper is able to handle input data losses by exploiting the open-loop prediction of the control input. Currently, only the first "hybrid" scheme is realized, in the future versions the user will be able to choose which scheme to employ.

6 Experiment Implementation and Results

In the previous section we presented the general implementation of the MPC wrapper as tool to solve generic optimal control problem, with a specific example from the case-study. Here we will briefly present how this is integrated in the ROS system that manages the informations routing from sensors and to actuators as well as the implementation of the actual multi-agent algorithm. Afterwards, we will present the simulation and experimental results.

6.1 Ground and Aerial MPC interfaces

The MPC wrapper communicates through a costum ROS message type: the data, however, comes from different sensors and the control incorporates values targeted at different actuators. To this reason, two ROS nodes have to be implemented as interfaces with the wrapper, both for the ground and for the aerial controllers.

Both interfaces have been implemented according to a similiar scheme as the "hybrid" procedure described in the previous chapter:

1. Incoming and outcoming data are latched in member variables of a class whenever a new message is received or a new solution is read.

2. The main task is to process the data to either gather the useful sensors variables and building `mpc_wrapper/InputData` structure or to extract the computed control input from `mpc_wrapper/OutputData`.l

3. A main loop spins in parallel with a constant rate and publishes at each iteration the processed input data structure. In general sensors are not synchronized and when the received data is sampled they can refer in general to slightly different time instants.

The conversions are schematized in fig. 6.1 and 6.2

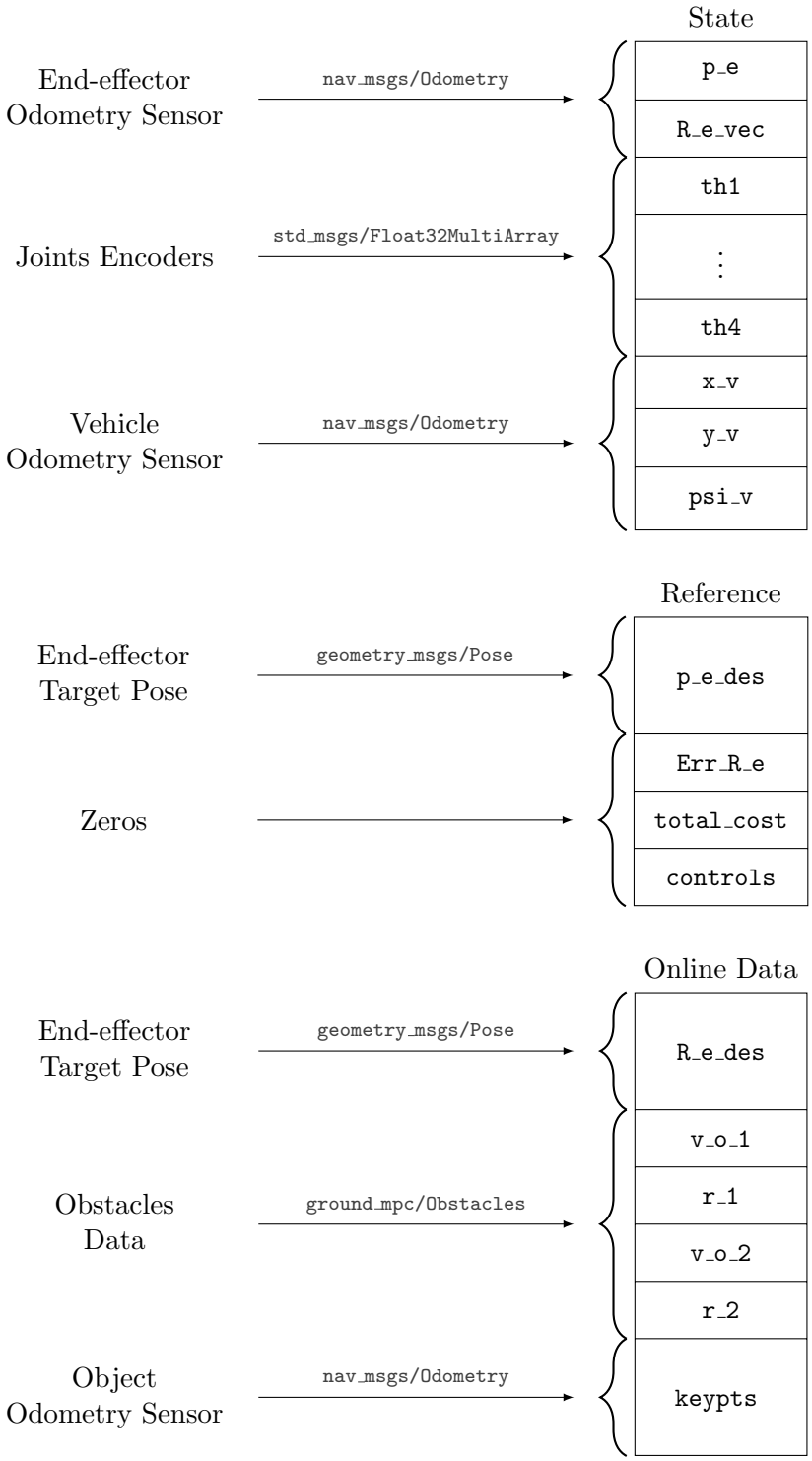


Figure 6.1: Data conversion table for the ground MPC wrapper input interface

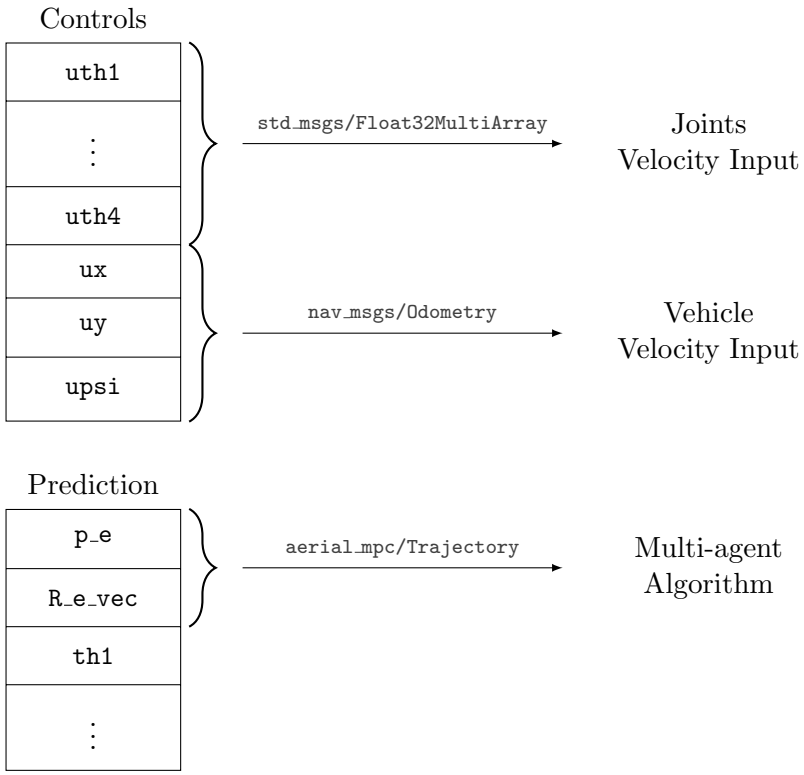


Figure 6.2: Data conversion table for the ground MPC wrapper output interface

The MPC may run at very slow rate and if data is gathered at the same rate this error could be relevant. For this reason, the interface loop frequency shoud be higher and comparable to the sensor data rate. As mentioned, the MPC wrapper allows for overwriting input data before the computation starts, and with a small time period if the packets refer to different time instants, such error is neglectable.

Note that the output message contains the state prediction for the ground (the leader) end effector that will be processed by the main algorithm and supplied to the follower, as will be explained in the next section.

6.1.1 Aerial Robust Control

While the input interface for the aerial controller is almost completely equivalent to the ground one, it has been noticed experimentally both in the gazebo and with real robots that supplying the input directly to a velocity controller for the aerial vehicle reults in really poor performances, not suitable for the cooperative task.

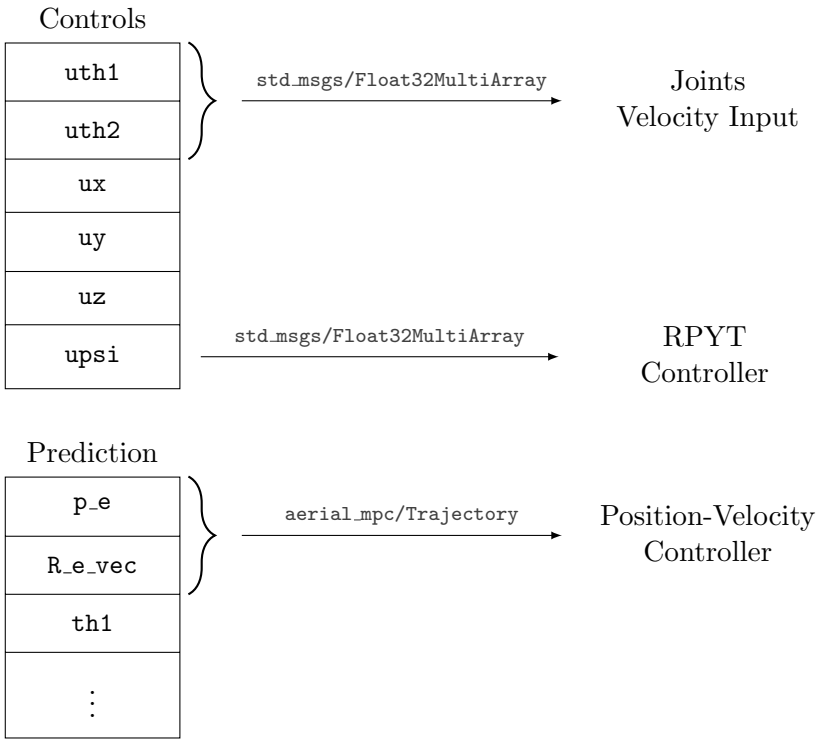


Figure 6.3: Data conversion table for the aerial MPC wrapper output interface

To this aim, a different control strategy has been considered. The idea is to exploit the position prediction to include both position and velocity information to a PID that outputs roll and pitch angles, which will then be supplied to a roll-pitch-yaw-rate-thrust controller. Specifically:

$$\begin{aligned} \boldsymbol{p}_{v,a,des}(t) &= \hat{\boldsymbol{p}}_{v,a}(t+1) \\ \boldsymbol{v}_{v,a,des}(t) &= \frac{\hat{\boldsymbol{p}}_{v,a}(t+2) - \hat{\boldsymbol{p}}_{v,a}(t)}{2T_s} \end{aligned} \tag{6.1}$$

where $\hat{\boldsymbol{p}}_{e,a}$ is the prediction generated by the solver and T_s is the MPC sample time. The control law considered is a modified version of [21], in which the state is stabilized in the point $(\boldsymbol{p}_{v,a,des}, \boldsymbol{v}_{v,a,des})$, instead of $(\boldsymbol{p}_{v,a,des}, 0)$.

The algorithm proposed first computes a desired force in the world frame like the system was fully actuated. Then, it convertes the force into desired roll and pitch angles and considers the body frame component. These are then supplied to an external controller, along with the yaw rate which is easy to handle due to the flying vehicle actuation.

6.2 The Task Commander

Last section described how the two controllers are integrated with the sensors and actuators. Here we will describe the ROS node that sets up the entire experiment, which provides for a first stage of rendez-vous in which the two robots approach and grasp the object and then implements the algorithm by computing the required quantities and handling the message routing from leader to folllower. In this way the algorithm is completely transparent to the agents controller, with the only exception that the aerial MPC must accept either a constant reference or an entire trajector. This is done by sending the reference command into two separate topics of message type:

- `geometry_msgs/Pose` will set all the reference samples in the prediction window $i = 1, \dots, N$ to the constant pose defined in the message. This is used in the first stage to command the robot into the grip position.
- `aerial_mpc/Trajectory`, defined as an array of `geometry_msgs/Pose`, specifies a different pose for each sample. This is required in the cooperative algorithm as a generic trajectory is the target of the optimization problem.

The task commander node is based on a hierarchy-based set of classes, whose UML schema is shown in figure 6.4. Each class inherits the properties and methods from the parent one according to the arrows in figures and can be described as follows:

- *ArenaEntity* is the base class and represents everything whose position and velocity can be measured (e.g. a mo-cap element in the real arena).
- *Agent* contains the common definitions for the agents, which includes constant position control and the desired initial position w.r.t. the object.
- *Object* is a passive element which is available to be grasped. Currently, this extension is only useful in simulation as grasping is not yet implemented with real robots.
- *Leader* and *Follower* extends the *Agent* class with case-specific commands and properties. Leader and follower must convert trajectories according to the algorithm: the leader needs to directly control the object position and it needs to know the relative transform, just like the follower needs one to convert the predicted trajectory.

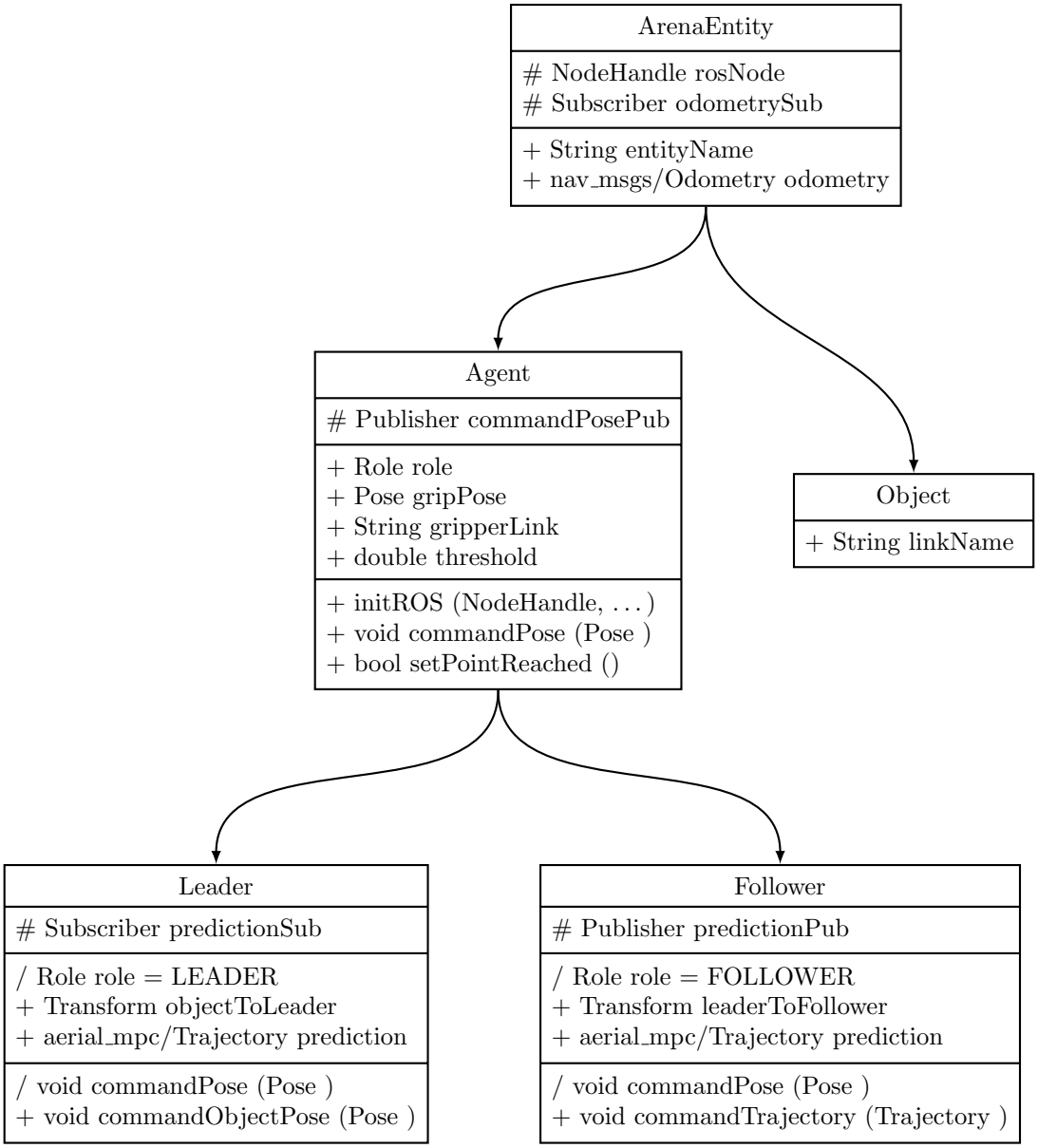


Figure 6.4: UML description of the main concepts in the task commander implentation

Note that those classes do not conceptually represent the agents as the control is not implemented at this point but they rather offer a convenient and unified way to process references to agents. This allows for a modular implementation of the multi-agent coordination and leaves open the possiblity to extend the number of involved agents as well as a change in the roles.

The experiment procedure

The procedure to execute the experiment is managed by a finite state machine that is composed of the following states:

1. `NOT_READY`. This is the initial condition: the robots are located in a generic position and the aerial robot is on ground. Both robot must be turned on, in particular the drone should have the motor started.

   A command received on topic `/world/commander/command` with content `data: 0` will start the aerial take off by supplying an hovering position as constant setpoint a predefined height above its current location, and advances the state.

2. `AERIAL_TAKEOFF`. The take off command has been sent and the machine is waiting for it to reach the setpoint under a specified threshold in position and velocity.

3. `IDLE`. The flying vehicle is hovering and robots are ready to start the rendez-vous stage. In this step the machine could wait for another command but currently it advances immediately to the rendez-vous state.

4. `START_RENDEZVOUS`. To both robots a setpoint nearby the object, bringing them to a relative pose with respect to the object frame, and the state is advanced.

5. `WAIT_RENDEZVOUS`. Wait until both setpoints have been reached according to the specified threshold. When this occurs, the state is advanced.

6. `WAIT_POSE`. The randez-vous stage has been accomplished and the machine waits for the object pose to be supplied through the topic `/world/commander/object_pose`. This will cause the state to advance.

7. `HANDSHAKE`. The first step once the object setpoint is given is to perform the handshake operation. To do that the node will sample the current agents pose and compute the relative transforms needed to convert the setpoints from object to leader and from leader to follower.

8. `COOPERATIVE_TASK_SPIN`. Right after the handshake, the machine will start the main loop, which will:

    a) Get the ground predicted trajectory.

    b) Convert it through the leader to follower estimated transform (see section 2.3 for concepts and remarks about estimation).

    c) Supply the processed trajectory to the aerial MPC through the message type `aerial_mpc/Trajectory`.

9. `STOP`. This state is reached whenever after the handshake the user send a `data: 1` on the topic `/world/commander/command`: this will cause the experiment to terminate and the aerial vehicle to land, by sending a constant setpoint with the current pose with the z component set to 0. The state will change back to `NOT_READY`.

Commands in the topics `/world/commander/*` are typically published manually via command line but they are suitable to be sent from another application as well.

Note that the object position is not strictly necessary for the experiment to run. It is needed for the rendez-vous to compute the absolute positions of the agents. Afterwards, it is only useful to account for collision avoidance. While in the simulation environment no problem can arise, in the real arena the object should have a position sensor, i.e. in the specific case with the motion-capture system, it should have the reflective markers or, otherwise, a separate simple estimation node should be introduced.

6.3 Gazebo simulation results

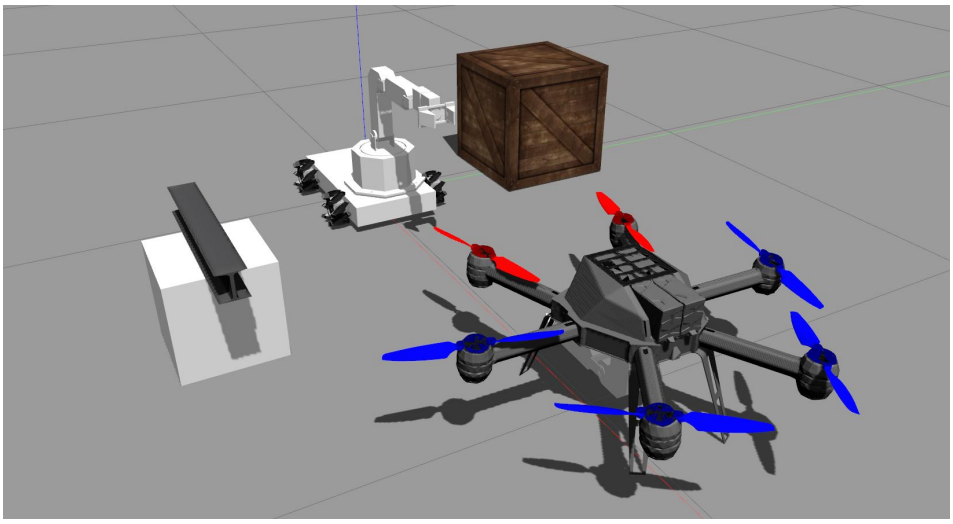
The ROS system nodes that implement the algorithm and the controllers can be interfaced with the Gazebo simulation environment defined in chapter 6. Here we will present some test of the cooperative algorithm.

6.3.1 Initialization and rendez-vous

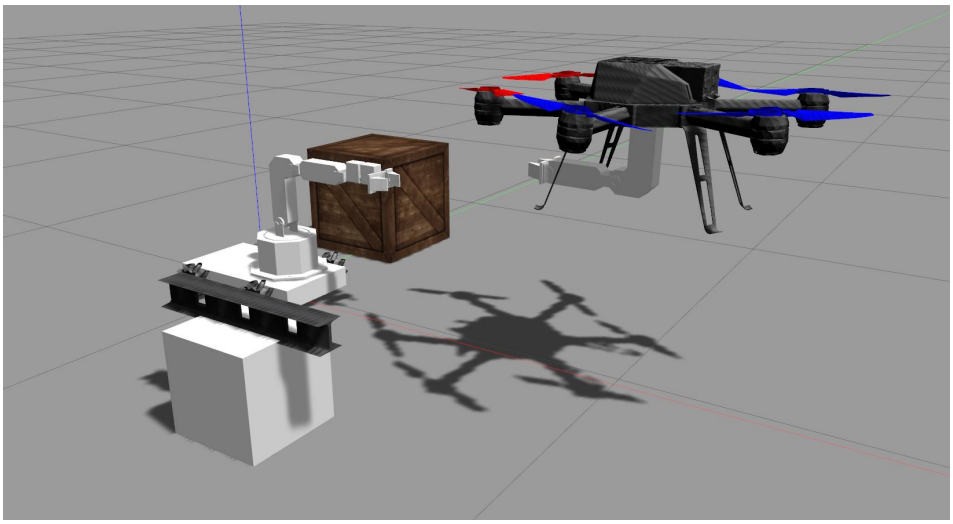
The initial condition for the Gazebo simulation is reported in figure 6.5a. According to the procedure defined in the previous section, the aerial robot will first take of (Fig. 6.5b) and then both robots will approach the object to the prescribed relative position, Fig. 6.5c.

The robots are controlled individually so we will take the opportunity to assess the single control performances. Figure 6.6 shows the position and orientation error for the ground robot from the starting point to the grasp configuration. From the graph is evident after about $6\,\mathrm{s}$ a disturbance that applies to both x and y component. From several simulations this appeared to be a disturbance introduced at the base by the physics engine of Gazebo at slow velocitiese, perhaps due to a bad simulation of static friction. However, after this small glitch the simulation proceeds smoothly and the controller is finally able to stabilize the position. We notice that, just like in the pure kinematic case, the vertical component is the slowest to converge. Also, by looking closely, it can be noticed a slightly wobbling trend, that matches the orientation error. This is an expected behavior since to keep the orientation error small a complex coordination of the joints is required and in a real environment this is inevitabily accomplished with some error, which anyways is sufficiently small.

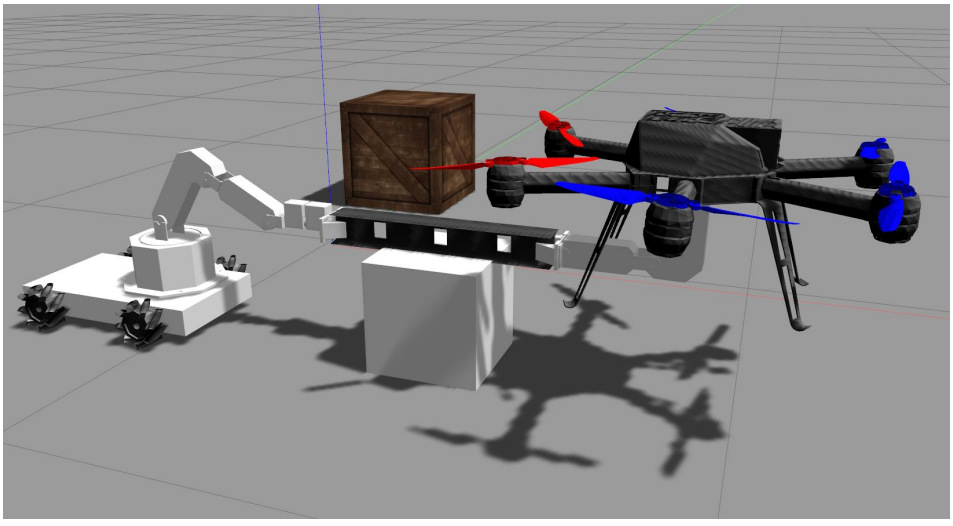
The aerial control (Fig. 6.7) is significantly faster and smoother: the low-speed unexpected disturbance is not present and the three components converge with about the same speed. This comes from the higher mobility of the aerial vehicle as most of the movements can be done by the vehicle rather than the joints. On the other hand, the orientation error has quite higher peaks, caused by the rolling/pitching movements that the underactuated vehicle must perform to achieve an acceleration. Such error could be critical in the cooperative task as it could produce too much internal torques. However, if the overall movement is kept slow, they will result in small and tolerable errors, as we will se in the next section.



(a) Initial condition

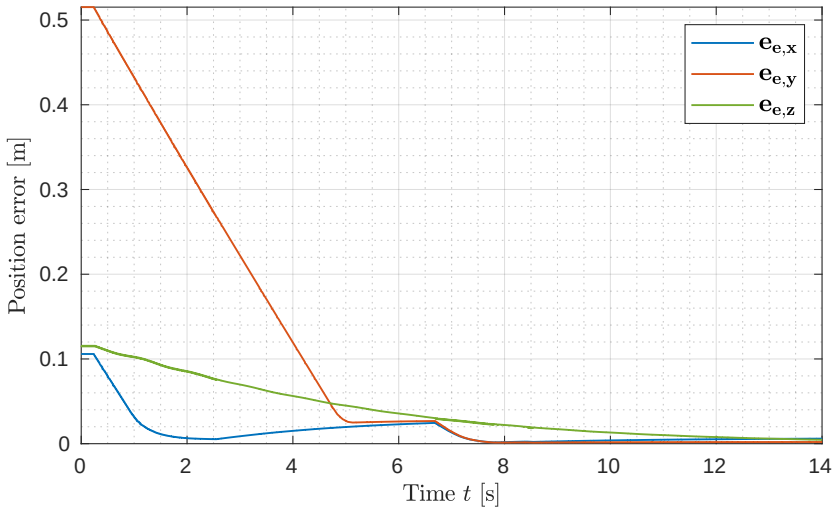


(b) Aerial take off stage

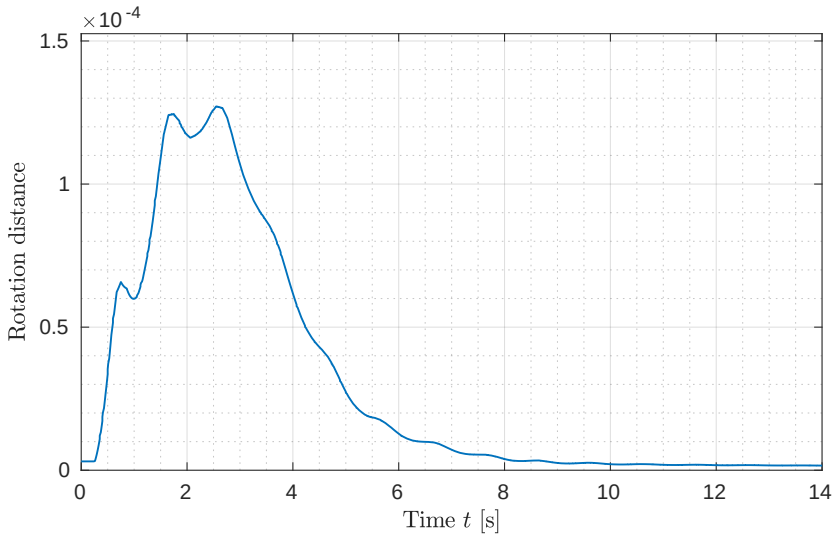


(c) Grasping position

Figure 6.5: Gazebo simulation of rendez-vous phase.

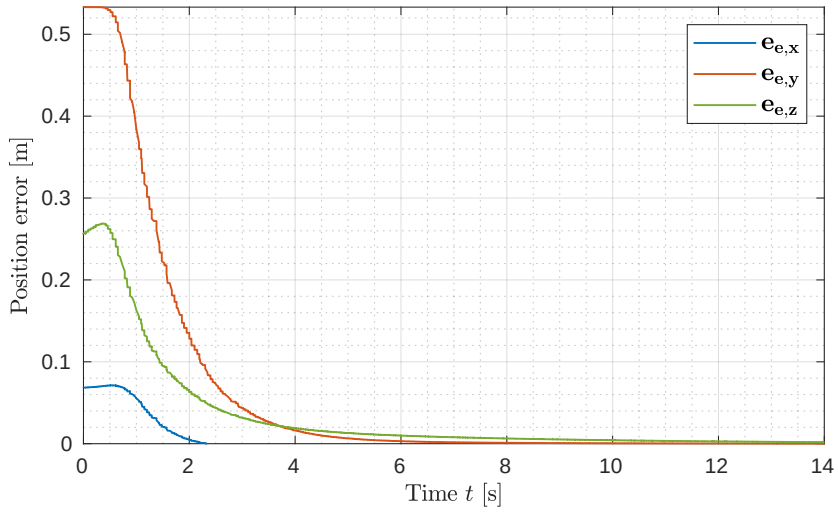


(a) Ground end effector position error

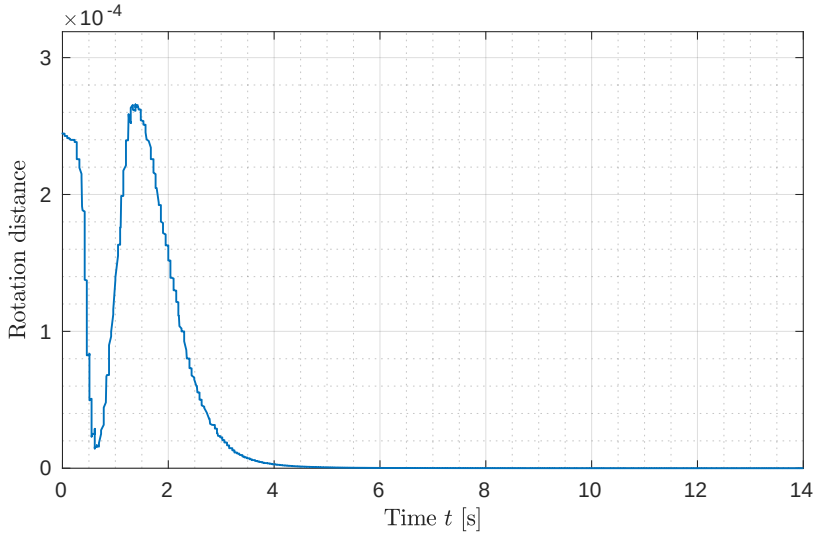


(b) Ground end effector orientation error

Figure 6.6: Individual control for ground end-effector during rendez-vous



(a) Aerial end effector position error



(b) Aerial end effector orientation error

Figure 6.7: Individual control for aerial end-effector during rendez-vous

6.3.2 Cooperative manipulation experiments

The cooperative algorithm starts from the gripping configuration (Fig. 6.5c) and, again according to the algorithm described in the previous section, executes the preliminary handshake stage. Then, the grasp is simulated with *EasyGripper* (sec. 4.2.3). To evaluate the performances, the object will be attached only to the ground robot, and check the follower tracking error as assessment. Here we present the results of experiment in several conditions.

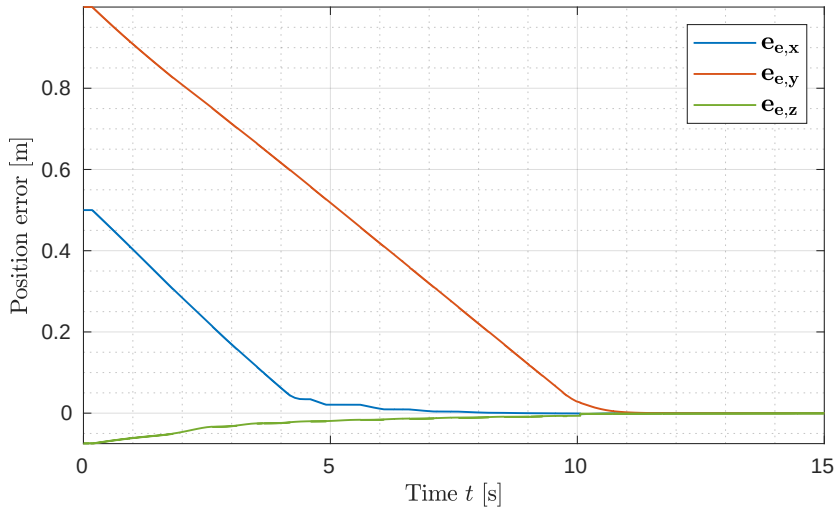
Transportation with no obstacle

First we will test the algorithm without obstacle. The object will be transported with the same orientation to a constant setpoint $\boldsymbol{p}_{o,des} = [0, -1.5, 0.3]$. Figure 6.10 shows three instants of the transportation task. The lines represent the last 10 predicted trajectory for the ground base (blue) and end effector (black), the corresponding desired trajectory for the aerial robot (purple) and the predicted trajectory from the aerial MPC (orange). The less these lines are scattered, the more accurate is the prediciton.

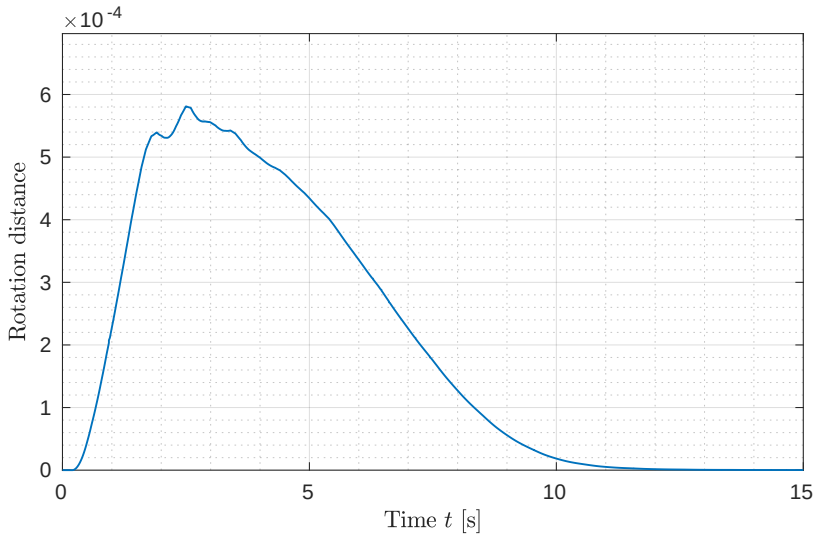
Figures 6.8 shows the resulting tracking error for the leader, which is just as expected. More interesting is the tracking error of the follower, in Fig. 6.9, that shows that the controller is able to track the reference trajectory with an error of the order of a few centimeters. It is notable that the tracking error is related to the same effect observed in the pure kinematic simulation: the initial acceleration causes a larger error in the first instants (fig 6.10b). However, the controller is able to overcome the error rather quickly.

Lift test

Right after the object was transported to the previous target, we test the ability of the algorithm to lift the object, which as we already discussed is a quite complex movement for the joint yet necessary to overcome ground obstacles. In figure 6.11 the initial and final configuration are reported. From figure 6.12 we notice that the wobbly trend is still present in the leader movements. This however has no side effects on the position stability and the slowness of the movements allows for a even smaller tracking error than horizontal transportation.

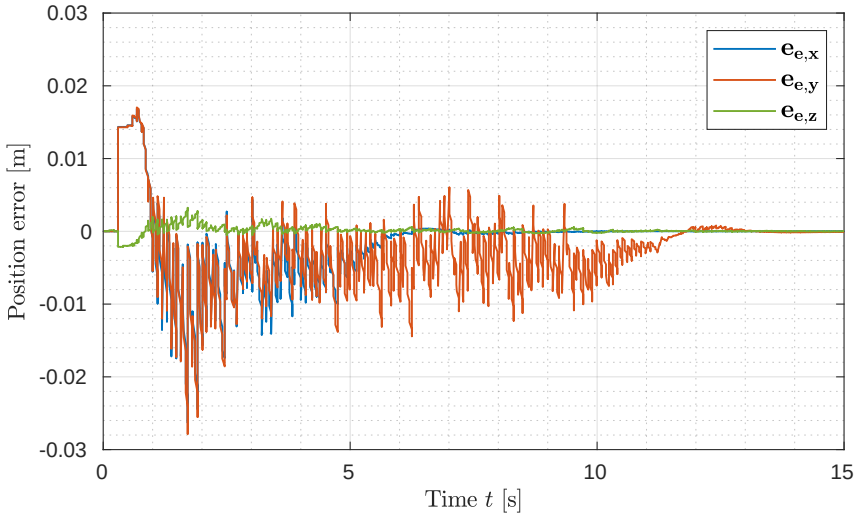


(a) Ground end effector position error

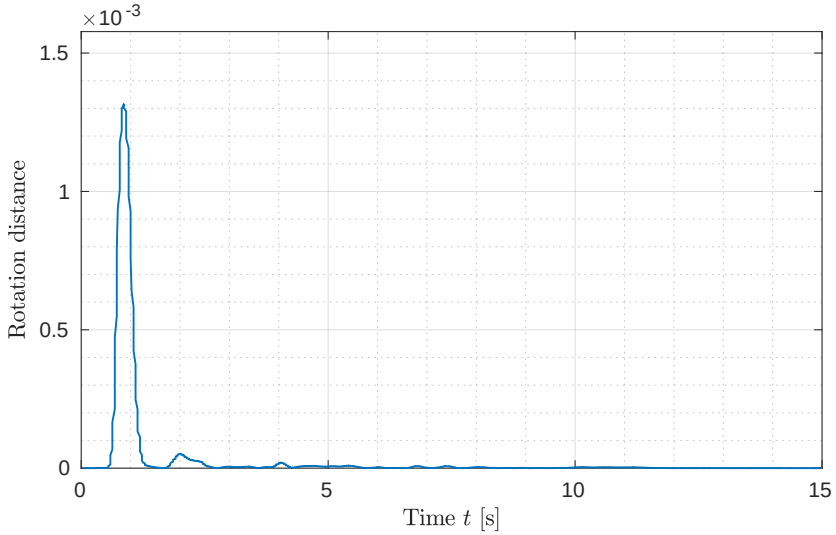


(b) Ground end effector orientation error

Figure 6.8: Individual control for ground end-effector in cooperative transportation without obstacles

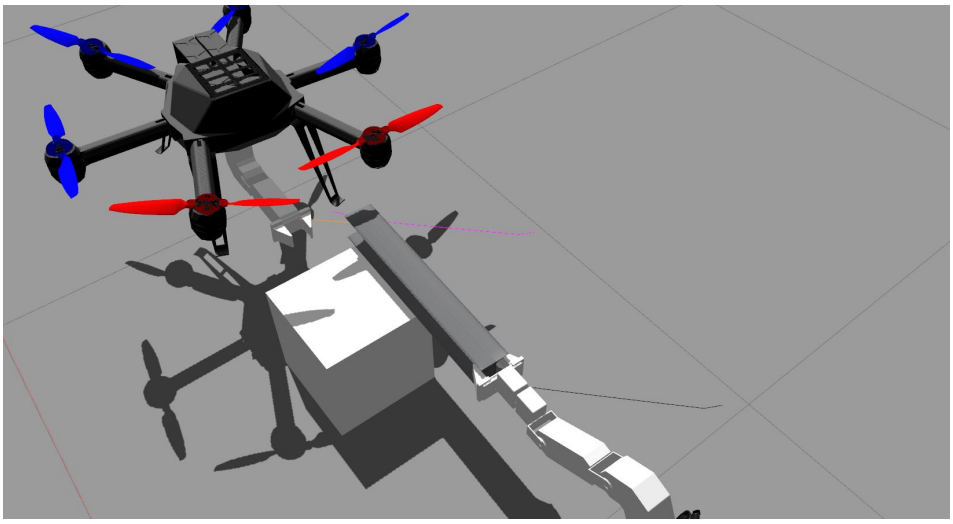


(a) Aerial end effector position error

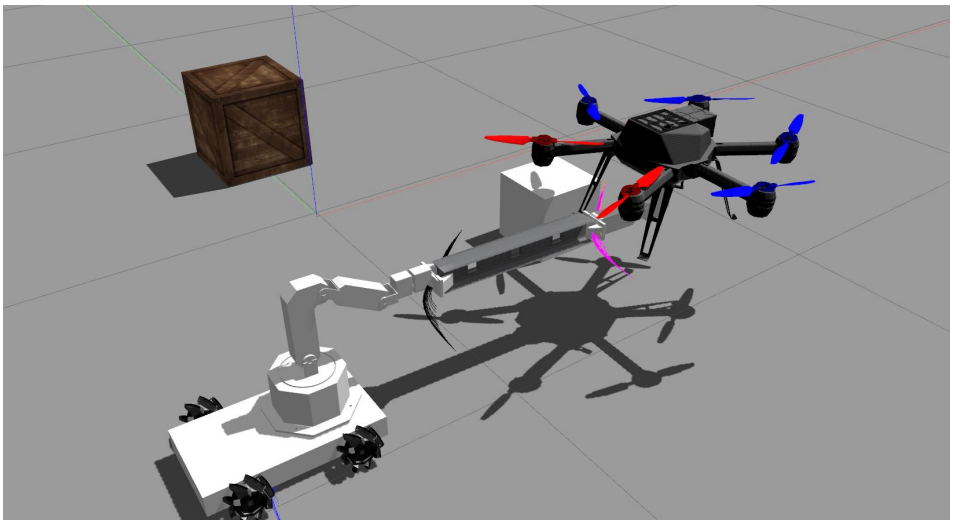


(b) Aerial end effector orientation error

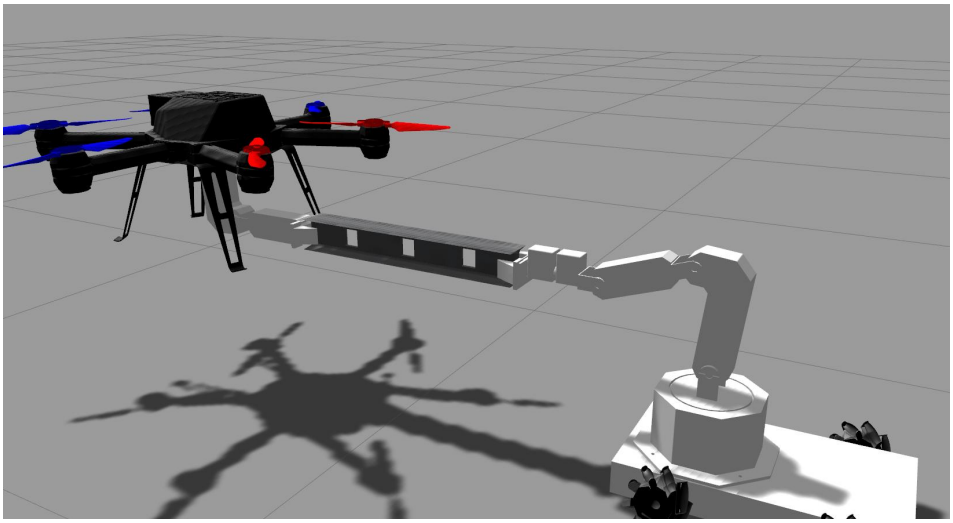
Figure 6.9: Individual control for aerial end-effector during in cooperative transportation without obstacles



(a) Initial error peak due to acceleration

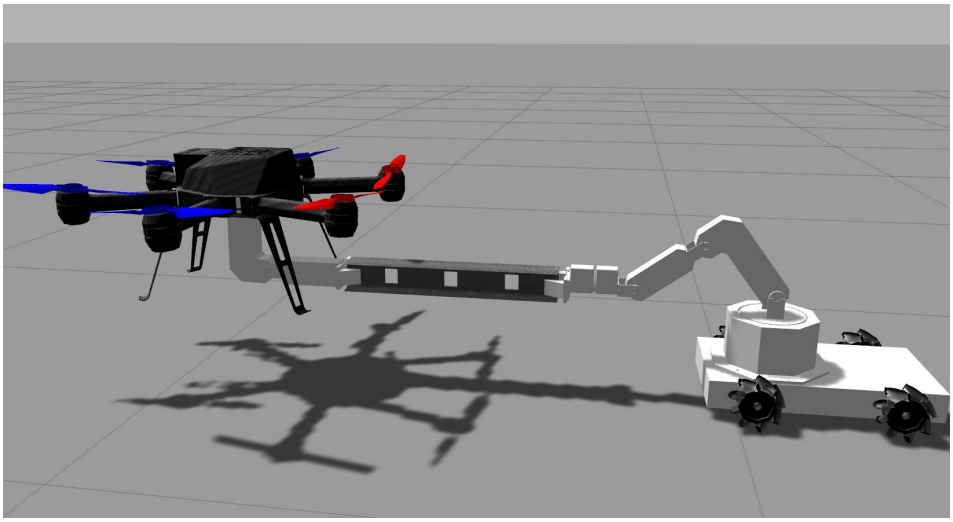


(b) Intermediate stage

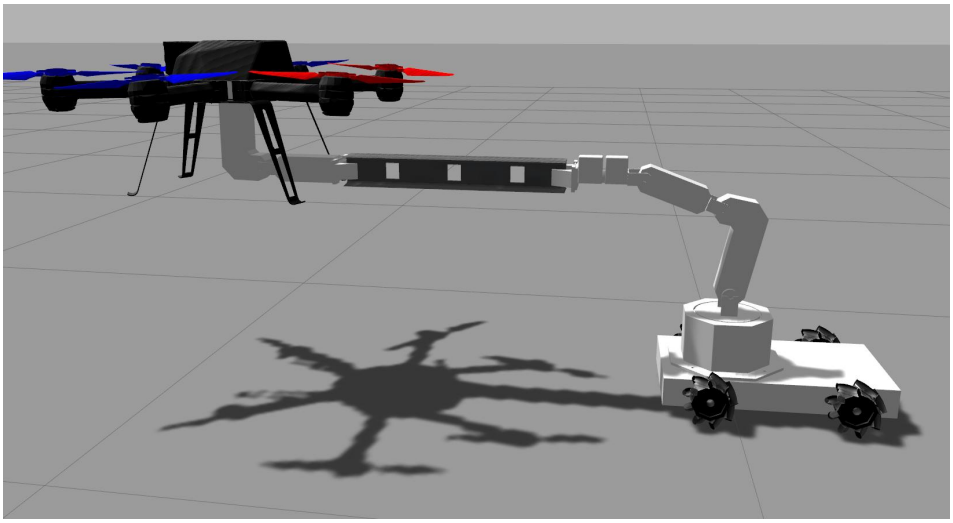


(c) Final condition

Figure 6.10: Gazebo simulation of a transportation without obstacles.

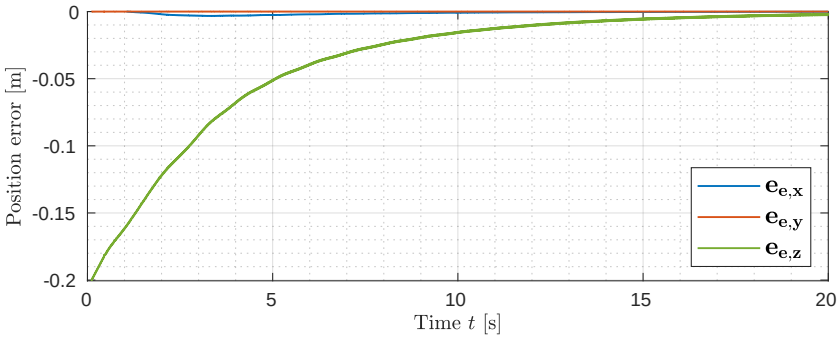


(a) Initial configuration

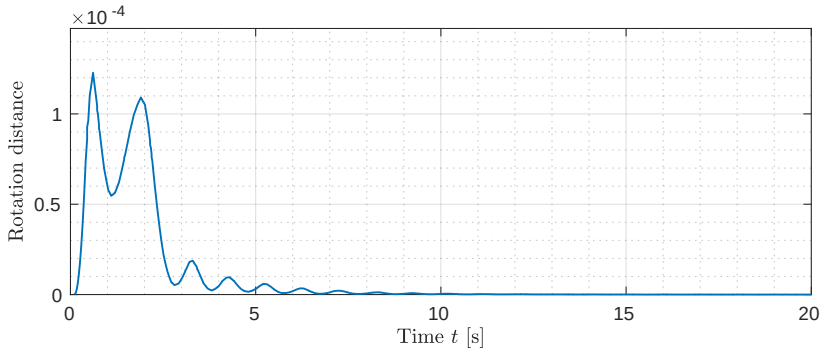


(b) Final configuration

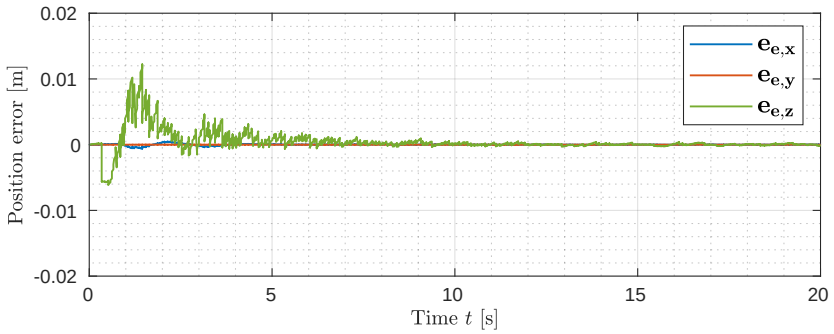
Figure 6.11: Gazebo simulation of object lift.



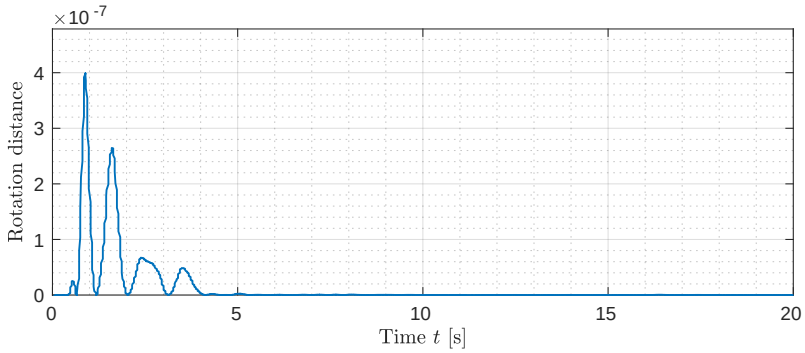
(a) Ground end effector position error



(b) Ground end effector orientation error



(c) Aerial tracking position error



(d) Aerial tracking orientation error

Figure 6.12: Errors for cooperative object lift

Transportation in presence of obstacles

In the previous paragraphs we showed the controller of horizontal transportation and vertical lifting separately. We now test the performances of the collision avoidance algorithm by considering setpoint with an obstacle in the middle of the path, in a similar way to what done in the pure kinematic simulation. To do this the algorithm shall find a path that requires both horizondal and vertical movements.

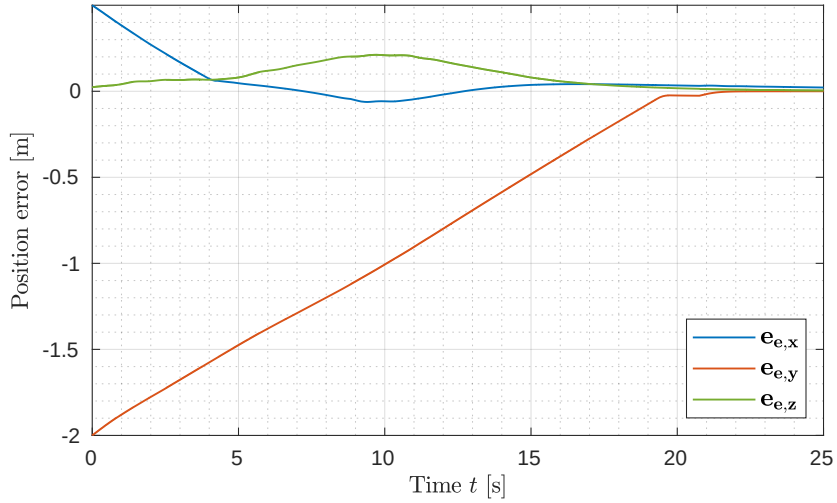
The obstacle is a box placed at $\boldsymbol{p}_{obst} = [0, 0.5, 0]$. The obstacle model that the algorithm uses is a sphere, which does not have to inscribe the box as it would most likely make the overcome impossible for the ground robot. On the contrary, it should approximate the contour with a certain margin: a safety margin is anyways considered from the algorithm (see sec. 2.4.2).

The supplied object setpoint is $\boldsymbol{p}_{o,des} = [0, 1.5, 0.2]$. The results are reported in figure 6.13 and 6.14 while figure 6.15 shows three instants of the gazebo simulation. From the first graph we notice the lift of about $0.2\,\mathrm{m}$ to avoid the obstacle, along with a small shift in the x axis as wel. That figure also shows that a considerable orientation error is produced. This could be avoided in theory by increasing the orientation error. However, due to imperfect actuation, this has been showed to produce an oscillatory behavior with the considered parameters.

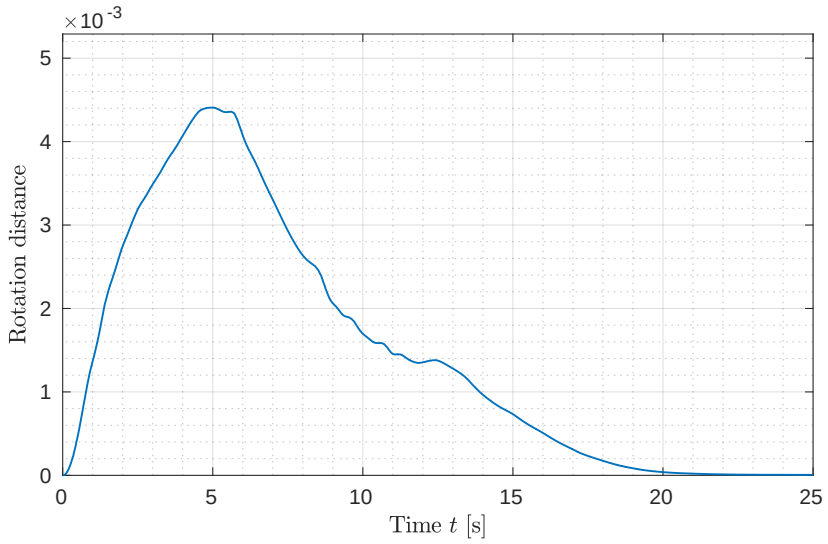
The prediction scattering that can be noticed in Fig. 6.15b reveal a higher uncertainty in the control in the presence of the obstacle. However, this is well handled by the robustness of the algorithm, that in any case is able to ensure an error which is comparable to that of a simple horizontal transportation. From 6.14 we notice that the initial acceleration and the horizontal movement, which is the fastest one, are responsible for the larger error component (x and y component).

Conclusions

The simulation experiments show that in such environment, which is quite realistic unlike the pure kinematic one, the algorithm is able to stabilize the system and provide small tracking error for the followers.

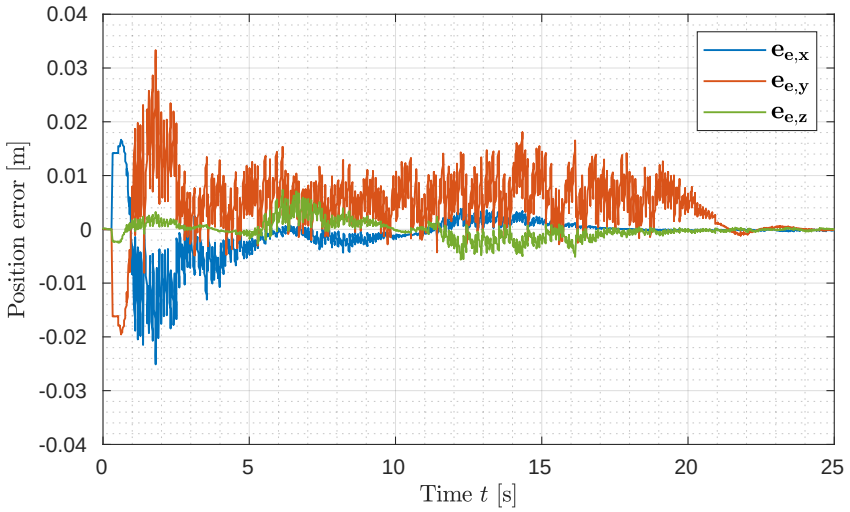


(a) Ground end effector position error

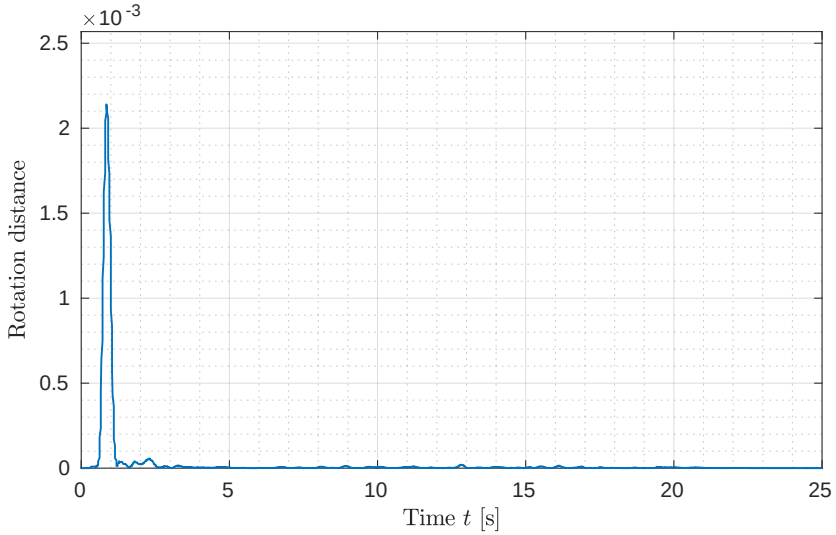


(b) Ground end effector orientation error

Figure 6.13: Individual control for ground end-effector in cooperative transportation with an obstacle

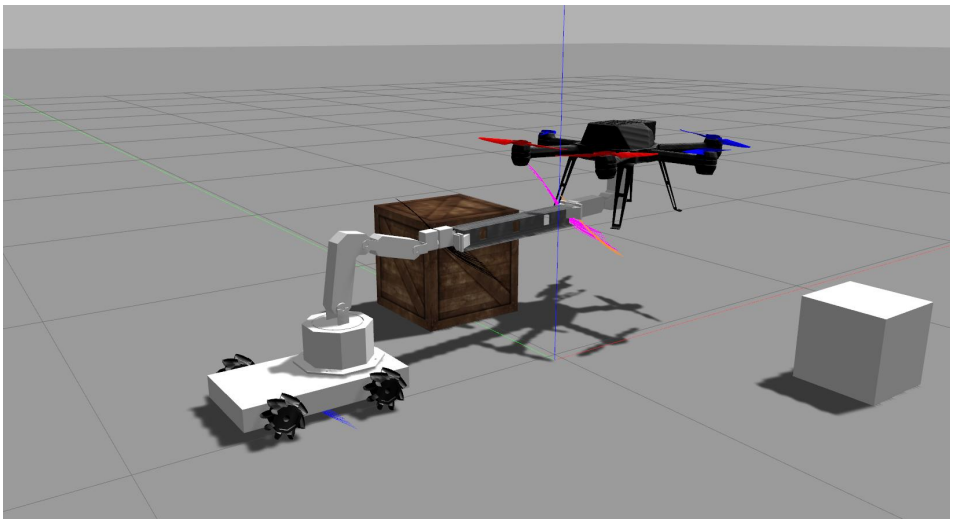


(a) Aerial end effector position error

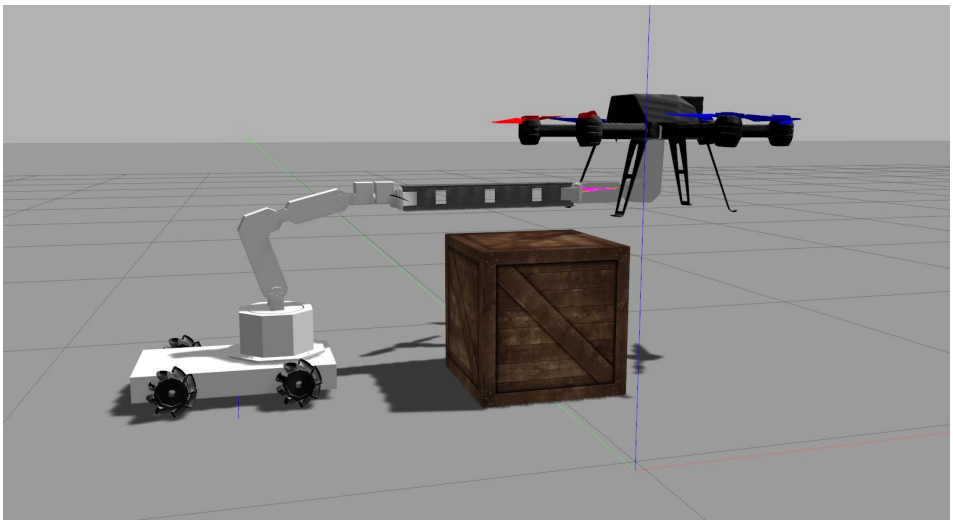


(b) Aerial end effector orientation error

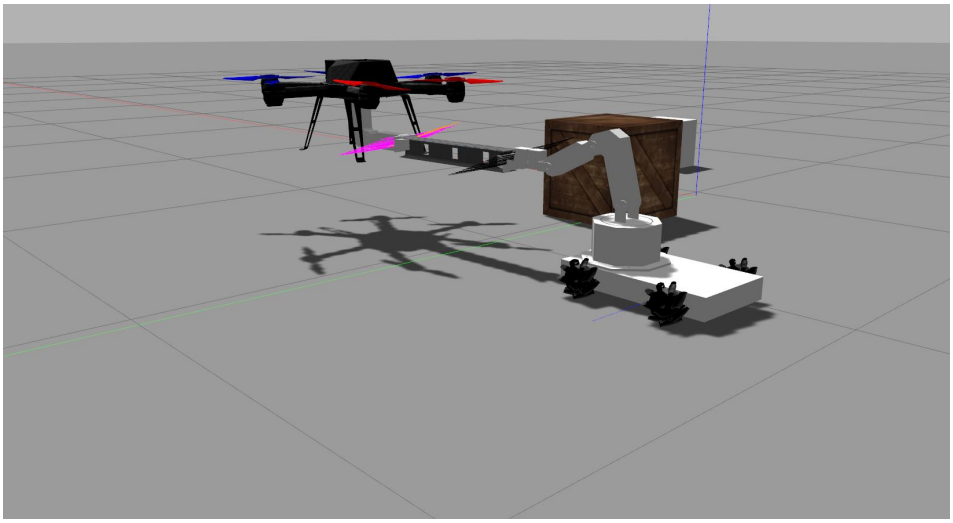
Figure 6.14: Tracking error for aerial end-effector during in cooperative transportation with an obstacle



(a) Initial curve to avoid obstacle.



(b) Object lifted above obstacle



(c) Obstacle took over.

Figure 6.15: Gazebo simulation of a transportation with an obstacle.

6.4 Experimental results

Although the gazebo simulation is more realistic and the results are more reliable than those obtained in simulink, the algorithm performances heavily depend on the phyisical and dynamical properties of the agents and objects so an experimental validation is required to actually assess the effectiveness of the algorithm.

6.4.1 Experimental setup

The experiments have been performed up to a preliminary stage in the *Smart Mobility Lab* (SML), part of the *Integrated Transport and Research Laboratory* (ITRL) of KTH, Royal Institute of Technology (Stockholm, Sweden).

The lab provides for a $6\,\mathrm{m} \times 6\,\mathrm{m} \times 3\,\mathrm{m}$ large flying arena (Fig. 6.16) as well as a motion capture system by Qualisys, capable of a precision of $6\,\mathrm{mm}$ and an output rate of $100\,\mathrm{Hz}$, that covers the whole volume. The arena is suitable also for testing ground mobile robots by removing the mattresses.

Figure 6.16: Flying arena in *Smart Mobility Lab* (KTH, Stockholm)

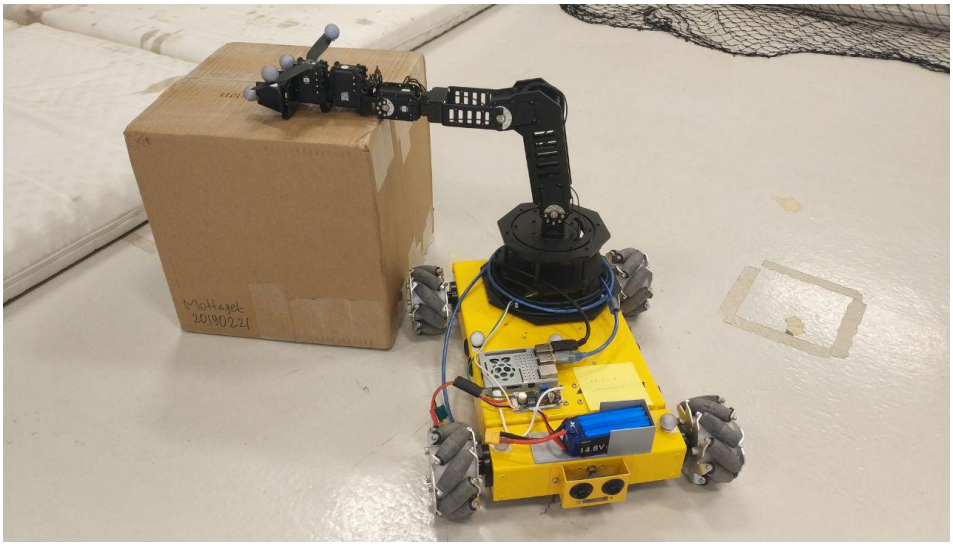


Figure 6.17: Ground robot composed of RB-Nex-03 omidirectional base and WidowX manipulator.

The ground robot (Fig. 6.17) is composed of an omnidirectional mobile base, RB-Nex-03 from *Nexus Robot*, equipped with a WidowX arm from Trossen, whereas the aerial robot is the costum built hexarotor Popeye (Fig. 6.18), endowed with a two joints arm extracted from the same manipulator.

However, the preliminary tests did not involve Popeye itself, but the Storm *srd370*, a smaller and lighter UAV, more safe to run initial experiments. Although it has no arm attached, in the tests a dummy arm was considered. From the Gazebo simulations it is noticeable that, since it has to keep a defined orientation, the aerial arm configuration is almost fixed: we consider

Figure 6.18: Popeye aerial robot

that one as fixed configuration of the virtual arm. Practically, zeros values for the joints are supplied to the MPC and the generated outputs for the joints velocities, which are expected to be close to zero in the particular experimental conditionss, are simply ignored.

The simulated position-velocity sensors in the Gazebo models are replaced by the measurement of the mo-cap, which can identify rigid bodies defined by at least 4 markers, that are placed on the base of the vehicles and on the ground end-effector. Since for the measurement to be reliable the markers should have a minimum distance, a broken propeller has been exploited to outdistance the markers, as can be noticed from figure 6.17.

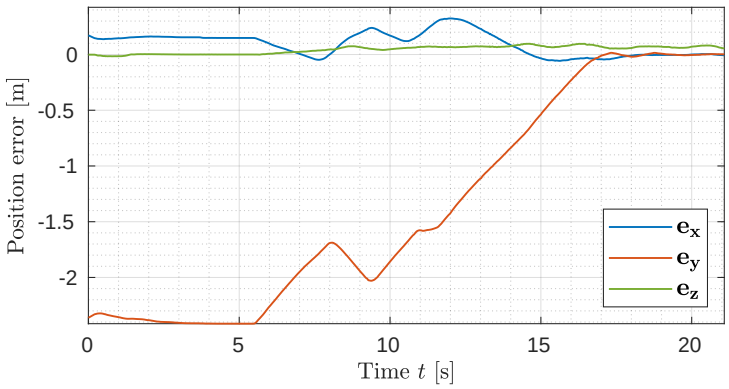
6.4.2 Individual control experiments

Ground robot with obstacle avoidance

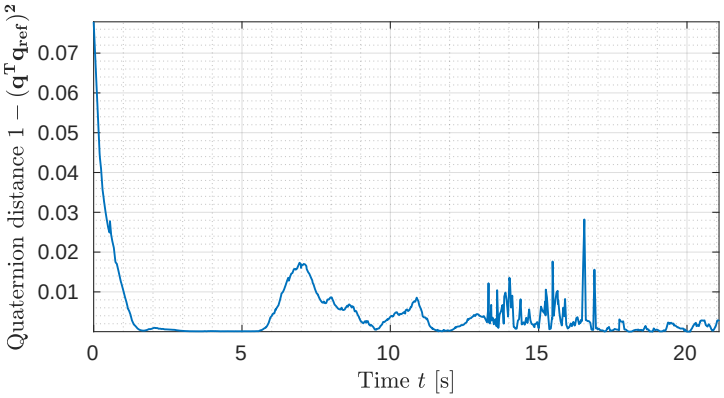
The ground robot has been tested for individual control which is the same kind of algorithm used for the cooperative task. An obstacle is placed at the center of the path, as represented in figure 6.20. The results are reported in 6.19 and shows that the controller is still able to accomplish both stability and obstacle avoidance. However, the resulting path is quite irregular, with sudden accelerations and turns. This is most cenrtainly due to a combination of the control actuation dynamics, not accounted in the MPC, and network delays.

Also, the base speed low-level control is rather inaccurate and suffers of a high non-linear distortion at low speeds. This forces to make the dynamics faster, to avoid running at those speeds, resulting in a less accurate control, being the sample time the same.

Another source of distortion might be markers occlusion from the mo-cap: given the configuration of the cameras, some spots of the arena are more critical to errors and this might lead the Qualisys marker shape recognition to fail. In such cases the state is not updated and, due to the current implementation scheme, the last available state is kept constant. When the markers are recognized again, the state updates abruptly, which may also justify part of the quick accelerations during the control.



(a) Ground end effector position error

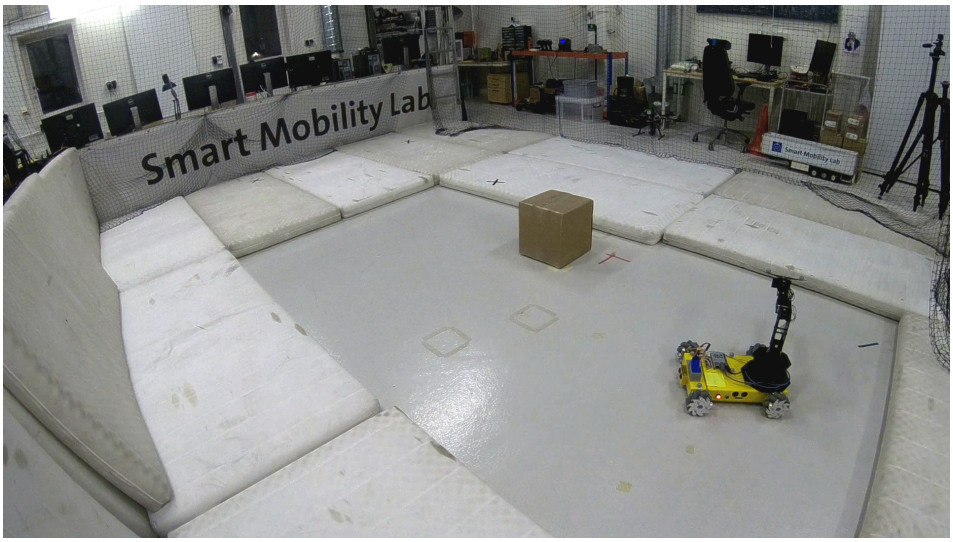


(b) Ground end effector orientation error

Figure 6.19: Individual control of real ground robot with obstacle avoidance

(a) Initial condition that emulates the grasping configuration.

(b) Obstacle avoidance accomplished by circumventing the box.



(c) Final desired condition.

Figure 6.20: Individual control on real robot in SML arena.

Aerial flight tests with srd370

As mentioned, preliminary flight tests have been done with the *srd370* robot, to make sure about the controller stability before employing the larger and more fragile hexarotor.

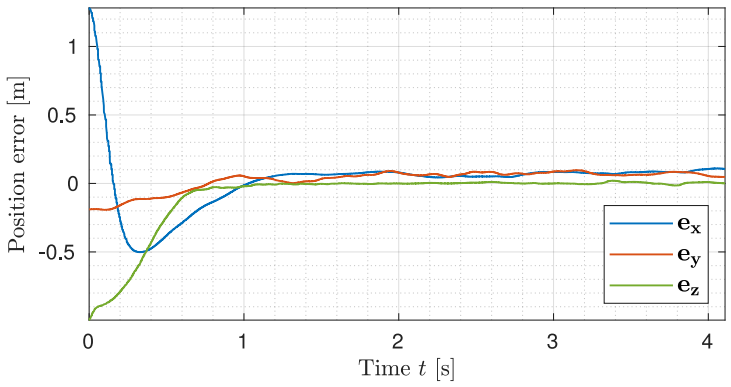
Graph in figure 6.21a shows the error convergence during take off. During the this first phase the low level controllers were not fine tuned yet and this motivates the large overshoot on the x component. Besides this, even with coarse low level tuning, the MPC is able to stabilize the error dynamics with rather high performances.

Figure 6.21b shows the results of an horizontal setpoint in the x direction. From the graph is evident that, even if the error initially converges fast, at steady state the aerial robot oscillates with a quite remarkable error. The srd370 is quite light and the popellers cannot ensure a perfect stability. Even though those errors could be reduced by tuning the low level controller, significantly higher stability and precision is expected from the exarotor which is the actual robot that can perform the cooperative manipulation, so fine tuning the test drone is not really useful.

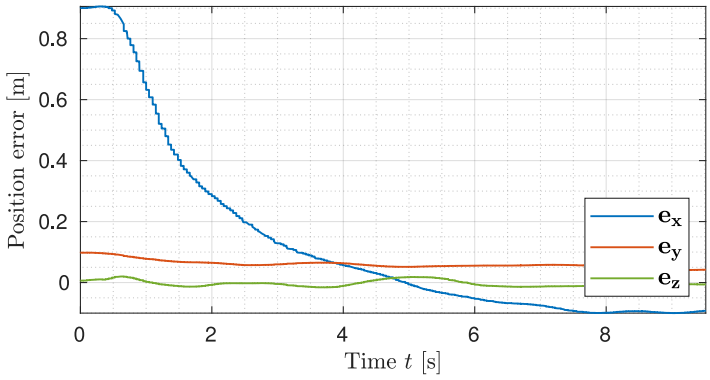
Finally, figure 6.21c shows the response to another setpoint and subsequently a robustness test to external disturbances, applied from outside the flying arena net for safety reasons (Fig. 6.22c). The controller is able to stabilized after the disturbance is applied, with the same steady-state error of the previous case.

6.5 Conclusions

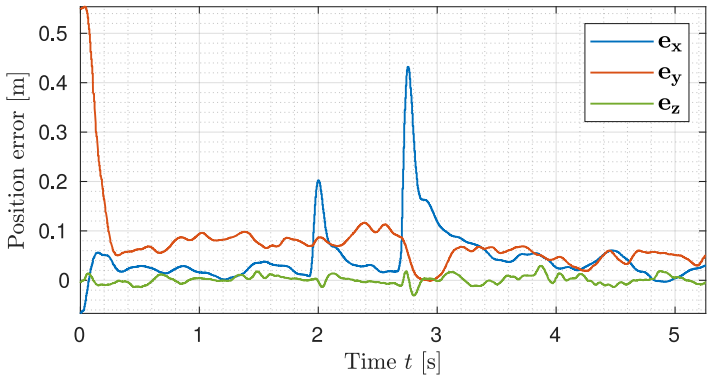
The proposed multi-agent technique has been revealed effective in the Gazebo simulations, whose results are comparable to those obtained in Simulink. From the preliminary tests with real robots, the MPC shows good converging performances as far as individual control is concerned. Since in simulation it has been noticed that the cooperative performances strongly depend on the performances of the single MPCs we can expect the cooperative manipulation with real robots to have comparable performances. However, due to time lack, this is left for future works.



(a) Take off



(b) Horizontal displacement set-point



(c) Robustness test to external disturbances test

Figure 6.21: Flight test with srd370: individual MPC control test

(a) Hovering position after take off

(b) First setpoint reached

(c) External disturbances

Figure 6.22: Individual control on real robot in SML arena

Bibliography

[1] D. Ariens, B. Houska, and H.J. Ferreau. *ACADO for Matlab User's Manual*. http://www.acadotoolkit.org. 2010–2011 (cit. on p. 3).

[3] M. Corah and N. Michael. „Active estimation of mass properties for safe cooperative lifting". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 4582–4587 (cit. on p. 2).

[4] H.J. Ferreau, H.G. Bock, and M. Diehl. „An online active set strategy to overcome the limitations of explicit MPC". In: *International Journal of Robust and Nonlinear Control* 18.8 (2008), pp. 816–830 (cit. on p. 54).

[5] H.J. Ferreau, C. Kirches, A. Potschka, H.G. Bock, and M. Diehl. „qpOASES: A parametric active-set algorithm for quadratic programming". In: *Mathematical Programming Computation* 6.4 (2014), pp. 327–363 (cit. on p. 54).

[6] Rolf Findeisen, Lars Imsland, Frank Allgower, and Bjarne A. Foss. „State and Output Feedback Nonlinear Model Predictive Control: An Overview". In: *European Journal of Control* 9.2 (2003), pp. 190 –206 (cit. on p. 5).

[7] A. Franchi, C. Secchi, M. Ryll, H. H. Bulthoff, and P. R. Giordano. „Shared Control : Balancing Autonomy and Human Assistance with a Group of Quadrotor UAVs". In: *IEEE Robotics Automation Magazine* 19.3 (2012), pp. 57–68 (cit. on p. 2).

[8] A. Gawel, M. Kamel, T. Novkovic, et al. „Aerial picking and delivery of magnetic objects with MAVs". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 5746–5752 (cit. on p. 2).

[9] B. Houska, H.J. Ferreau, and M. Diehl. „ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization". In: *Optimal Control Applications and Methods* 32.3 (2011), pp. 298–312 (cit. on p. 3).

[10] B. Houska, H.J. Ferreau, M. Vukov, and R. Quirynen. *ACADO Toolkit User's Manual*. http://www.acadotoolkit.org. 2009–2013 (cit. on p. 3).

[11] M. Kiehl. „Parallel multiple shooting for the solution of initial value problems". In: *Parallel Computing* 20.3 (1994), pp. 275 –295 (cit. on p. 54).

[12]K. Kondak, F. Huber, M. Schwarzbach, et al. „Aerial manipulation robot composed of an autonomous helicopter and a 7 degrees of freedom industrial manipulator“. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 2107–2112 (cit. on p. 2).

[13]H. Lee, H. Kim, and H. J. Kim. „Planning and Control for Collision-Free Cooperative Aerial Transportation“. In: *IEEE Transactions on Automation Science and Engineering* 15.1 (2018), pp. 189–201 (cit. on p. 2).

[14]T. Lee. „Collision avoidance for quadrotor UAVs transporting a payload via Voronoi tessellation“. In: *2015 American Control Conference (ACC)*. 2015, pp. 1842–1848 (cit. on p. 2).

[15]G. Loianno and V. Kumar. „Cooperative Transportation Using Small Quadrotors Using Monocular Vision and Inertial Sensing“. In: *IEEE Robotics and Automation Letters* 3.2 (2018), pp. 680–687 (cit. on p. 2).

[16]R. Mahony, V. Kumar, and P. Corke. „Multirotor Aerial Vehicles: Modeling, Estimation, and Control of Quadrotor“. In: *IEEE Robotics Automation Magazine* 19.3 (2012), pp. 20–32 (cit. on p. 2).

[17]D. I. Montufar, F. Muñoz, E. S. Espinoza, O. Garcia, and S. Salazar. „Multi-UAV testbed for aerial manipulation applications“. In: *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*. 2014, pp. 830–835 (cit. on p. 2).

[18]R. Naldi, A. Gasparri, and E. Garone. „Cooperative pose stabilization of an aerial vehicle through physical interaction with a team of ground robots“. In: *2012 IEEE International Conference on Control Applications*. 2012, pp. 415–420 (cit. on p. 2).

[19]T. Nguyen and E. Garone. „Control of a UAV and a UGV cooperating to manipulate an object“. In: *2016 American Control Conference (ACC)*. 2016, pp. 1347–1352 (cit. on p. 2).

[20]Alexandros Nikou, Christos K. Verginis, Shahab Heshmati-Alamdari, and Dimos V. Dimarogonas. „A Nonlinear Model Predictive Control Scheme for Cooperative Manipulation with Singularity and Collision Avoidance“. In: *CoRR* abs/1705.01426 (2017). arXiv: 1705.01426 (cit. on pp. 3, 12, 15).

[21]P. O. Pereira and D. V. Dimarogonas. „Lyapunov-based generic controller design for thrust-propelled underactuated systems“. In: *2016 European Control Conference (ECC)*. 2016, pp. 594–599 (cit. on p. 72).

[22]M. Quigley, B. Gerkey, and W.D. Smart. *Programming Robots with ROS*. O'Reilly Media, 2015 (cit. on p. 44).

[23]N. Staub, M. Mohammadi, D. Bicego, D. Prattichizzo, and A. Franchi. „Towards robotic MAGMaS: Multiple aerial-ground manipulator systems“. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 1307–1312 (cit. on p. 2).

[24]A. Suarez, G. Heredia, and A. Ollero. „Lightweight compliant arm with compliant finger for aerial manipulation and inspection“. In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 4449–4454 (cit. on p. 2).

Webpages

[2]Martin Behrendt. *MPC scheme basic*. 2009. URL: `https://commons.wikimedia.org/wiki/File:MPC_scheme_basic.svg` (cit. on p. 6).

List of Figures