POLITECNICO DI TORINO

Master's Degree in Mechatronics Engineering

Master's Degree Thesis

Model Predictive Control and Reinforcement Learning for Quadrotor Agile Flight Control

Supervisors

Prof. Luciano LAVAGNO

Postdoc Dirk REINHARDT

Prof. Sebastien GROS

Candidate

Giacomo DEMATTEIS

October 2022

Summary

The work presented in this master thesis project is related to the control aspects of fast and agile drone trajectory tracking. Aerodynamic forces make quadrotors trajectory tracking at high-speed extremely challenging. At high speeds these complex effects have a major impact in performance loss, measured in terms of large position tracking errors.

Model Predictive Control (MPC) together with Reinforcement Learning (RL) is used to tackle the problem. We propose to use RL to offline tune the MPC formulation using the data obtained from the system. MPC is an optimal control method with a well-established theory that exploits a dynamic model of the platform and provides constraint satisfaction. RL methods allow solving control problems with minimum prior knowledge about the task. RL automatically trains the decision-making process via trial and error and maximize the performance through a given reward function. In our approach, RL is used for adjusting the MPC parameters, through a Q-learning technique by exploiting MPC as a function approximator. Indeed, unlike Deep Neural Networks (DNN), MPC as a function approximator for RL, can explicitly achieve constraints satisfaction, stability, and safety. Therefore, the goal is to combine the advantages of both methods: the ability of MPC to safely control a physical robot through well-established knowledge and the power of RL to learn complex policies using experienced data. The resulting control framework can handle large-scale inputs, reduce human intervention in design and tuning, and eventually achieve optimal control performance.

The method is verified through precise and extensive simulation environment.

This work is the result of a seven months period at the Norwegian University of Science and Technology (NTNU), Trondheim, Norway under the group of Professor Sebastein Gros, supervised by Postdoctoral fellow Dirk Reinhardt.

Acknowledgements

ACKNOWLEDGMENTS

To Rebecca,
who makes it all much more beautiful.

Table of Contents

List of Figures

Acronyms

AI
Artificial Intelligence

MPC
Model Predictive Control

RL
Reinforcement Learning

MDP
Markov Decision Process

DP
Dynamic Programming

DNN
Deep Neural Networks

OCP
Optimal Control Problem

NTNU
Norwegian University of Science and Technology

DOF
Degree of freedom

PID
Proportional Itegrative Derivative

Chapter 1

Introduction

Quadrotors trajectory tracking at high speeds and high accelerations with good accuracy is still far from being a solved problem. Autonomous robots are gaining more and more popularity every year. The variety of industries touched by these robots ranges from transport to infrastructure, military, agriculture, security, entertainment and search and rescue. They provide a series of advantages, as they provide reliable, innovative, affordable and efficient aid. For instance, the risks associated with human intervention can be deeply lowered by the use of this kind of technology. However the general belief is that they still do not exploit their full capabilities of maneuver. A higher control precision would allow to prevent situations where even small deviations from reference have catastrophic consequences, besides having faster flights in known-free environments.

There is not much work on agile flight of quadrotors for speeds beyond 5m/s and accelerations above 2g. Most of the popular applications require not to go beyond this kind of velocities and accelerations. Identifying a dynamics model capable of thoroughly describing the aereodynamic effects while still being light enough for real time performance is the main challenge, especially at high speeds. [1]

The most common control implementation in the industry is the classical Proportional Integral Derivative (PID) control. A PID controller is a control loop mechanism employing feedback through the use of three different gains. The proportional gain is used to minimize the tracking error. It is responsible for a quick response and thus should be set as high as possible, but without introducing oscillations. If the P gain is too high we get high-frequency oscillations, if too low the vehicle will react slowly to input changes. The D (derivative) gain is used for rate damping. If the D gain is too high the motors become twitchy (and maybe hot), because the D term amplifies noise, if too low we see overshoots after a step-input. The I (integral) gain keeps a memory of the error. The I term increases when the desired rate is not reached over time. The PID control provides a reliable

and effective control strategy for a vast number of applications. It is also cheap and relatively simple to implement and mantain. So, for most common applications it is still the best option. But if we need to push the performance of our vehicle out of the ordinary, then PID is shown to have big limitations, and thus it does not allow to perform task that would require extreme accuracy and precision. Tasks that are, arguably, also the most interesting and with the most potential.

Model Predictive Control (MPC) has been shown to be a powerful model-based approach for solving complex quadrotor control problems. MPC is increasingly rising in popularity in many robotic domains, thanks to its capability of simultaneously dealing with complex nonlinear dynamic systems while satisfying different state and input constraints. However, many MPC applications still experience significant challenges, such as the need of an accurate mathematical model and the necessity of solving trajectory optimization problems online with the limited computational resources of embedded systems. This two requirements are opposite, since an accurate model often leads to computational overload.

On the other hand, Reinforcement Learning (RL) methods allow solving control problems with minimum prior knowledge about the task. The key idea of RL is to automatically train the policy via trial and error and maximize the task performance measured by the given reward function. While RL has achieved incredible results in a wide range of robot applications, the lack of interpretability of a controller trained using RL is of significant concern by the control community. Basically it lacks the intrinsic safety of MPC, crucial for most of critical applications.

Ideally, the control framework should be able to combine the advantages of both methods: the ability of model-based controllers, like MPC, to safely control a physical robot using the well-established knowledge in dynamic modeling and optimization and the power of RL to learn complex policies using experienced data automatically, in order to compensate the mismatch coming the dynamical model. [2]

The structure of this thesis is as follow: Chapter 2 contains all the fundamental theory knowledge, in preparation for the following parts of the work. Chapter 3 is the chapter that contains most of the work developed in the project. It encompasses the whole of Simulation structure, development and flow, together with a result preview. Chapter 4 shows a possible real implementation of the drone flight in a Motion Capture Laboratory. Initially, this part was intended to be a second main part of the project. Due to a major knee injury and relative surgery of the writer (making the in-person experiments a dream), the plan was shifted towards a simulations oriented project. So this part of the thesis remained to show the possibility for further investigations. Finally, the last two chapters are Results and Conclusion, where the fundamental outcomes are showed and explained.

Chapter 2

Theory

In this chapter, the theory foundations are presented. First we introduce the dynamics model of the quadrotor, with relative variables and state equations. After that, a brief discussion on the main characteristics of MPC is introduced. The relative mathematical representation is shown. Lastly, the RL theory is introduced. Starting from the very basics to end up with our innovative approach, this part represent the core of the theory chapter.

2.1 Dynamics Model

The aerodynamics of rotors was extensively studied during the mid 1900s with the development of manned helicopters, and detailed models of rotor aerodynamics are available in the literature. [3] [4]

The most popular multirotor aerial machine is the quadrotor vehicle. It has a very simple design. It consists of four individual rotors attached to a rigid cross body structure, as shown in 2.1. Control of a quadrotor is obtained by differential control of the thrust produced by each rotor. As all aerial vehicles, quadrotors have six Degree of freedom (DOF). Three corresponding to the movement along the three dimensions x-y-z, respectively Surge, Sway and Heave. The other three related with the rotation along the same axis, respectively Roll, Pitch and Yaw. In our quadrotor case, pitch, roll and heave controls are easily conceptualized. Further, as shown in 2.1, rotor i rotates anticlockwise (positive about the z axis) if i is even and clockwise if i is odd. Yaw control is obtained by adjusting the average speed of the clockwise and anticlockwise rotating rotors. Only these four DOFs are directly actuated and controlled. Thus, the system is underactuated and the remaining two DOFs corresponding to the translational velocity in the x-y plane must be controlled through the system dynamics. [4]

As for notation, we denote scalars in lowercase s, vectors in lowercase bold $\mathbf{v}$,

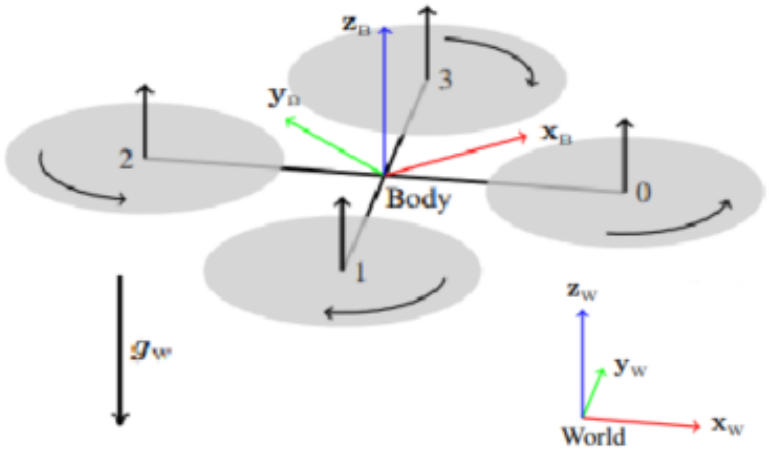

Figure 2.1: Diagram of the quadrotor model with the world and body frames and propeller numbering convention.[1]

and matrices in uppercase bold $\mathbf{M}$. We define the World W and Body B frames with orthonormal basis i.e. $\{\boldsymbol{x}_W, \boldsymbol{y}_W, \boldsymbol{z}_W\}$. The frame B is located at the center of mass of the quadrotor. Note that we assume all four rotors are situated in the xy-plane of frame B, as depicted in [1]. A vector from coordinate p_1 to p_2 expressed in the W frame is written as: ${}_W v_{12}$. If the vector's origin coincide with the frame it is described in, we drop the frame index, e.g. the quadrotor position is denoted as p_{WB}. Furthermore, we use unit quaternions $\boldsymbol{q} = (q_w, q_x, q_y, q_z)$ with $\|q\| = 1$ to represent orientations, such as the attitude state of the quadrotor body q_{WB}.

The 6-DOF nonlinear dynamics of the quadrotor can be then represented by a state vector containing position, attitude and linear/rotational velocities: $\boldsymbol{x} = [\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{v}, \boldsymbol{\omega}]^T$. The attitude representation is in quaternion form, thus leading for a total of 13 state variables. The control inputs $\boldsymbol{u}$ are the four quadrotor individual thrusts $T_i \; \forall i \in (0, 3)$. We assume that the quadrotor is a 6 degree-of-freedom rigid body of mass m and diagonal moment of inertia matrix $\boldsymbol{J} = \text{diag}(J_x, J_y, J_z)$. We write the nominal dynamics $\dot{\boldsymbol{x}}$ up to second order derivatives. The state space is thus 13-dimensional and its dynamics can be written as [1]:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{\boldsymbol{p}}_{WB} \\ \dot{\boldsymbol{q}}_{WB} \\ \dot{\boldsymbol{v}}_{WB} \\ \dot{\boldsymbol{\omega}}_B \end{bmatrix} = \boldsymbol{f}_{dyn}(\boldsymbol{x}, \boldsymbol{u}) = \begin{bmatrix} \boldsymbol{v}_W \\ \boldsymbol{q}_{WB} \cdot \begin{bmatrix} 0 \\ \boldsymbol{\omega_B}/2 \end{bmatrix} \\ \frac{1}{m} \boldsymbol{q}_{WB} \odot \boldsymbol{T}_B + \boldsymbol{g}_W \\ \boldsymbol{J}^{-1} \left(\boldsymbol{\tau}_B - \boldsymbol{\omega}_B \times \boldsymbol{J} \boldsymbol{\omega}_B \right) \end{bmatrix}, \tag{2.1}$$

where $\boldsymbol{g}_W = [0, 0, -9.81\,\text{m/s}^2]^\intercal$ denotes Earth's gravity, $\boldsymbol{T}_B$ is the collective thrust

and $\boldsymbol{\tau}_B$ is the body torque as in:

$$\boldsymbol{T}_B = \begin{bmatrix} 0 \\ 0 \\ \sum T_i \end{bmatrix} \qquad \text{and} \quad \boldsymbol{\tau}_B = \begin{bmatrix} d_y(-T_0 - T_1 + T_2 + T_3) \\ d_x(-T_0 + T_1 + T_2 - T_3) \\ c_\tau(-T_0 + T_1 - T_2 + T_3) \end{bmatrix} \tag{2.2}$$

where d_x, d_y are the rotor displacements and c_τ is the rotor drag torque constant. To incorporate these dynamics in discrete time algorithms, we use an explicit Runge-Kutta method of 4th order $\boldsymbol{f}_{RK4}(\boldsymbol{x}, \boldsymbol{u})$ to integrate $\dot{\boldsymbol{x}}$ given an initial state $\boldsymbol{x}_k$, input $\boldsymbol{u}_k$ and integration step δt by [5]:

$$\boldsymbol{x}_{k+1} = \boldsymbol{f}_{RK4}(\boldsymbol{x}_k, \boldsymbol{u}_k, \delta t). \tag{2.3}$$

2.2 Model Predictive Control

Model Predictive Control has its roots in optimal control. The basic concept of MPC is to use a dynamic model to forecast system behavior, and optimize the forecast to produce the best decision/the control move at the current time. Models are therefore central to every form of MPC.

MPC is a form of control in which the control action is obtained by solving online, at each sampling instant, a finite horizon optimal control problem in which the initial state is the current state of the plant. Optimization yields a finite control sequence, and the first control action in this sequence is applied to the plant. Open-loop optimal control problems often can be solved rapidly enough, using standard mathematical programming algorithms, to permit the use of MPC even though the system being controlled is nonlinear, and constraints on states and controls must be satisfied. [6]

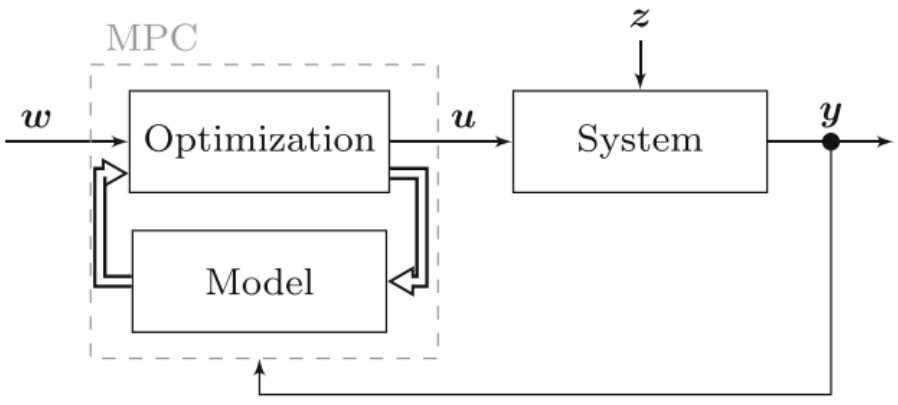

Figure 2.2: Simplified block diagram of a MPC-based control loop.[7]

Baseline Approach. As in 2.2, at each time step:

- A prediction over a given time horizon is performed, using a model of the plant
- The command input is chosen as the one yielding the "best" prediction (i.e., the prediction closest to the desired behavior) by means of some optimization algorithm. Typical algorithms are Non-Linear Program(NLP) solvers, a class of solvers that tackle a specific class of problems of the form similar to 2.4.

In its most general form, MPC stabilizes a system subject to its dynamics $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})$ along a reference $\boldsymbol{x}^r(t)$,$\boldsymbol{u}^r(t)$ by minimizing a cost $\ell(\boldsymbol{x}, \boldsymbol{u})$ as in:

$$\begin{aligned} &\min_{\boldsymbol{u}} \int \ell(\boldsymbol{x}, \boldsymbol{u}) \\ \text{subject to} \quad &\dot{\boldsymbol{x}} = \boldsymbol{f}_{dyn}(\boldsymbol{x}, \boldsymbol{u}) \qquad \boldsymbol{x}(t_0) = \boldsymbol{x}_{init} \\ &\boldsymbol{r}(\boldsymbol{x}, \boldsymbol{u}) = 0 \qquad \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{u}) \leq 0 \end{aligned} \tag{2.4}$$

where $\boldsymbol{x}_0$ denotes the initial condition and $\boldsymbol{h}$, $\boldsymbol{r}$ can incorporate (in-)equality constraints, such as input limitations.

For our application, and as most commonly done, we specify the cost to be of quadratic form $\ell(\boldsymbol{x}, \boldsymbol{u}) = \|\boldsymbol{x} - \boldsymbol{x}^r\|_Q^2 + \|\boldsymbol{u} - \boldsymbol{u}^r\|_R^2 = \|\boldsymbol{x}^d\|_Q^2 + \|\boldsymbol{u}^d\|_R^2$ and discretize the system into N steps over time horizon T of size $dt = T/N$. We account for input limitations by constraining $0 \leq \boldsymbol{u} \leq u_{max}$.

$$\begin{aligned} &\min_{u} \boldsymbol{x}_N^{d\intercal} Q \boldsymbol{x}_N^d + \sum_{k=0}^{N} \boldsymbol{x}_k^{d\intercal} Q \boldsymbol{x}_k^d + \boldsymbol{u}_k^{d\intercal} R \boldsymbol{u}_k^d \\ \text{subject to} \quad &\boldsymbol{x}_{k+1} = \boldsymbol{f}_{RK4}(\boldsymbol{x}_k, \boldsymbol{u}_k, \delta t) \\ &\boldsymbol{x}_0 = \boldsymbol{x}_{init} \\ &u_{min} \leq \boldsymbol{u}_k \leq u_{max} \end{aligned} \tag{2.5}$$

To solve this quadratic optimization problem we used the same previous work done by [1], where they constructed it using a multiple shooting scheme [8] and solve it through a sequential quadratic program (SQP) executed in a real-time iteration scheme (RTI) [8]. All implementations are done using ACADOS [9] and CasADi [10]. Later on for more information on these two software packages for optimization.

So, overall, Model Predictive Control is a general and flexible approach to nonlinear system control that allows us to deal with input/state/output constraints and to manage systematically the trade-off performance/command effort.

2.3 Reinforcement Learning

2.3.1 Basics

The idea that we learn by interacting with our environment is probably the first to occur when we think about the root of learning. Learning from interaction is a fundamental idea upon which nearly all theories of learning and intelligence are build. Reinforcement learning problems involve learning what to do (or more specifically: how to map situations to actions) so as to maximize a numerical reward signal.

A basic characteristics of reinforcement learning problems is simply to capture the most valuable aspects of the problem facing a learning agent interacting with an environment to achieve an objective.

Obviously, such an agent has to be able to sense the state of the environment to some extent and must be able to take actions that have an effect upon the state. The agent also must have a goal or goals in relation with the state of the environment. The formulation is intended to include just these three aspects (sensation, action and goal) in their simplest possible forms without trivializing any of them.

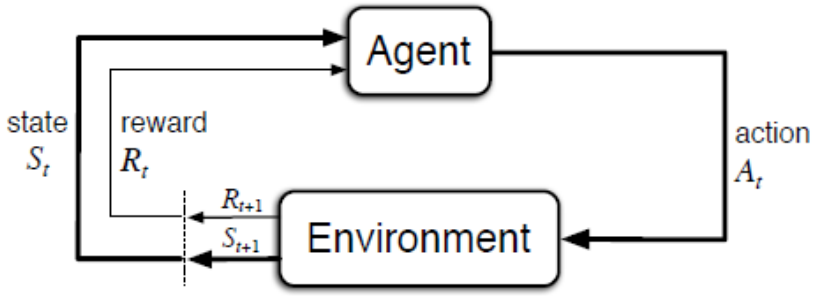

Figure 2.3: The agent-environment interaction in reinforcement learning.[11]

Beyond this elements, one can identify four other sub-elements in a reinforcement learning framework: a policy, a reward signal, a value function and, optionally, a model of the environment.

A policy specifies the learning agent's way of behaving or decision process at a certain time. Roughly speaking, a policy is a mapping from sensed states of the environment to actions to be taken while being in those states. The policy is the central part of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior.

A reward signal defines the objective in a reinforcement learning problem. Every

time step, the environment sends to the reinforcement learning agent a simple number, a reward. The agent's only goal is to maximize the total amount of reward it receives in the long run. The reward signal thus defines what are the good and bad events for the agent.

Whereas the reward signal specifies what is good in an immediate sense, a value function specifies what is good in the long run over a vast horizon. Roughly speaking, the value of a state is the total reward an agent is expected to accumulate in the future, starting from that specific state. Whereas rewards determine the immediate, intrinsic desirability of the states, values indicate the long-term desirability of states after taking into account the states that are likely to follow, and the rewards available in those states. It is values with which we are most concerned when making and evaluating decisions.

Action choices are made taken upon value judgments. We look for actions that head towards states of highest value, not highest reward, because these actions obtain the greatest amount of reward for us eventually. The derived quantity called value is the one we cared most about. Unfortunately, it is much more difficult to determine values than rewards. Rewards are basically given directly by the environment, but values must be estimated and re-estimated from the sequences of observation an agent makes over its entire lifetime. In fact, the most valuable component of almost all reinforcement learning algorithms is a method for accurately estimating values.

The fourth and final element of some reinforcement learning frameworks is a model of the environment. This is something that simulates the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. Models are used for planning, that is any way of deciding on a course of action by considering possible future situations before they are actually experienced. This is usually referred to as model-based Reinforcement Learning. [11]

So, the reinforcement learning problem is meant to be a direct framing for the problem of learning from interaction to achieve a goal. The learner and decision-maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. These interact continually, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent.

More specifically, in a discrete time framework, the agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$. At each time step t, the agent receives a characterization of the environment's *state*, $S_t \in \mathbb{S}$, where $\mathbb{S}$ is the set of possible states, and on that basis selects an *action*, $A_t \in \mathbb{A}(St)$, where $\mathbb{A}(St)$ is the set of actions available in state S_t. One time step further, in part as a consequence of its action, the agent receives a numerical *reward*, $R_{t+1} \in \mathbb{R}$,

and finds itself in a new state, S_{t+1}. Figure 3.2 diagrams the agent-environment interaction.

At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's policy and is denoted π_t, where $\pi_t(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Reinforcement learning methods defines how the agent varies its policy according to its experience.

The Markov Property and Markov Decision Process

In a reinforcement learning system, the agent makes its judgment as a function of the environment's state. In this section we discuss what are the requirements of the state signal, and what kind of information it should provide. In particular, we formally identify a property of environments and their state signals that is of specific interest, called the Markov property. Ideally, what we would like is a state signal that summarizes past sensations compactly, so that all relevant information is retained. This normally requires more than the immediate sensations, but never more than the complete history of all past sensations. A state signal that is able to retaining all relevant information is said to be Markov, or to have the Markov property. We formally define the Markov property for the reinforcement learning problem. To keep the mathematics simple, we assume here that there are a finite number of states and reward values. This enables us to work in terms of sums and probabilities rather than integrals and probability densities, but the approach is similar for continuous states and rewards. Let's consider how a general environment might respond at time $t + 1$ to the action taken at time t. In the most general, causal case this response could be dependent on all that has happened earlier. In this case the dynamics can be defied only by specifying the complete probability distribution:

$$Pr\{R_{t+1} = r, S_{t+1} = s'|S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \tag{2.6}$$

for all r, s', and all possible values of the past events: S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t. If the state signal has the Markov property, on the other hand, then the environment's response at $t + 1$ depends only on the state and action representations at t, in which case the environment's dynamics can be defined by specifying only

$$p(s', r|s, a) = Pr\{R_{t+1} = r, S_{t+1} = s'|S_t, A_t\} \tag{2.7}$$

for all r, s', S_t, and A_t. In other words, a state signal has the Markov property, and is a Markov state, if and only if 2.7 is equal to 2.6 for all s', r, and histories,

S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t. In this case, the environment and task as a whole are also said to have the Markov property.

If an environment has the Markov property, then its one-step dynamics 2.7 enable us to predict the next state and expected next reward given the current state and action. One can show that, by iterating this equation, one can predict all future states and expected rewards from knowledge only of the current state as well as would be possible given the complete history up to the current time. It also follows that Markov states provide the best possible basis for choosing actions. That is, the best policy for choosing actions as a function of a Markov state is just as good as the best policy for choosing actions as a function of complete histories.

Even when the state signal is non-Markov, it is still convenient to think of the state in reinforcement learning as an approximation to a Markov state. In particular, we always want the state to be a good basis for making predictions on future rewards and for selecting actions. In cases in which a model of the environment is learned, we also want the state to be a good basis for predicting subsequent states. Markov states provide an outstanding basis for doing all of these things. To the extent that the state approaches the ability of Markov states in these ways, one will obtain better performance from reinforcement learning systems. For all of these reasons, it is useful to think of the state at each time step as an approximation to a Markov state, although one should remember that it may not fully satisfy the Markov property.

A reinforcement learning task that satisfies the Markov property is called a Markov decision process, or MDP.

2.3.2 Our approach

The key idea for this part of the project is the use of Reinforcement Learning based on Model Predictive Control for the control of the drone. RL is basically used for adjusting the MPC parameters, using a Q-learning technique.

Reinforcement Learning is a powerful tool for tackling Markov Decision Process without prior knowledge of the process to be controlled. Indeed, RL attaches a reward function to each state-action pair and tries to find a policy to optimize the discounted infinite rewards labelled performance. Dynamic Programming (DP) methods can be used to solve MDP. However, DP requires a knowledge of the MDP dynamics, and its computational complexity is unrealistic in practice for systems having more than a few states and inputs [6]. Instead, most investigations in RL have focused on achieving approximate solutions, not requiring a model of the dynamics. Deep Neural Networks (DNN) are a common choice to approximate the optimal policy. However, analysing formally the closed-loop behavior of a learned policy based on a DNN, such as stability and constraints satisfaction is challenging.

[12]

Model Predictive Control is the well-known model-based control method described above, that employs a model of the system dynamics to build an input sequence over a given finite horizon such that the resulting predicted state trajectory minimizes a given cost function while respecting the constraints imposed on the system. The first input is applied to the real system, and the problem is solved at each time instant based on the latest state of the system. The advantage of MPC is its ability to explicitly support state and input constraints, while producing a nearly optimal policy. However, model uncertainties can severely impact the performance of the MPC policy.

In this work, we propose to use RL to offline tune the MPC formulation using the data obtained from the real system. Unlike DNN, MPC as a function approximator for RL, can explicitly handle constraints satisfaction, stability and safety.

Ideally, as already introduced, the control framework should able combine the advantages of both methods: the ability of MPC to safely control a physical robot using the established knowledge and the power of RL to learn complex policies using data automatically. Therefore, the resulting control framework can handle large-scale inputs, reduce human-in-the-loop design and tuning, and eventually result in optimal control performance. However, designing such a system remains a significant challenge. [12] [2]

2.3.3 Q-learning framework

Reinforcement Learning considers that the real system is described by a Markov Decision Process (MDP) with state transitions having the underlying conditional probability density $\mathbb{P}[\boldsymbol{s}_+|\boldsymbol{s},\boldsymbol{a}]$ where $\boldsymbol{s},\boldsymbol{a}$ is the current state-input pair and $\boldsymbol{s}_+$ is the subsequent state. The control literature typically uses the notation $\boldsymbol{s}_+ = f^{\text{real}}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{\zeta})$, where $\boldsymbol{\zeta}$ is a random disturbance and $\boldsymbol{f}^{\mathbf{real}}$ is the discretized real system dynamics (2.1) and $\boldsymbol{s} = [\boldsymbol{p}_{WB},\boldsymbol{q}_{WB},\boldsymbol{v}_{WB},\boldsymbol{\omega}_B]$.

We will label $L(\boldsymbol{s},\boldsymbol{a})$ as the baseline stage cost associated to the MDP at each transition. The optimum action-value function $Q_\star$, optimum value function $V_\star$ and optimum policy $\boldsymbol{\pi}_\star$ associated to the MDP are defined by the Bellman equations [12]:

$$V_\star(s) = \min_a Q_\star(s,a), \tag{2.8a}$$

$$Q_\star(s,a) = L(s,a) + \gamma\mathbb{E}[V_\star(s_+)|s,a], \tag{2.8b}$$

$$\pi_\star(s) = \arg\min_a Q_\star(s,a) \tag{2.8c}$$

where $\gamma \in (0,1]$ is the MDP discount factor.

Q-learning is a classical model-free RL algorithm that tries to capture the action value function $Q_\theta \approx Q_\star$ via tuning the parameters vector $\theta \in \mathbb{R}^n$. The

approximation of the value function V_θ and parametric optimal policy π_θ can then be extracted from the Bellman equations. Q-learning uses the following update rule for the parameters θ at state s_k:

$$\delta_k = L(s_k, a_k) + \gamma V_\theta(s_{k+1}) - Q_\theta(s_k, a_k) \tag{2.9a}$$

$$\theta \leftarrow \theta + \alpha \delta_k \nabla_\theta Q_\theta(s_k, a_k) \tag{2.9b}$$

where the scalar $\alpha > 0$ is the learning step-size, δ_k is labelled the Temporal-Difference (TD) error and the input a_k is selected according to the corresponding parametric policy $\pi_\theta(s_k)$ with possible addition of small random exploration.

Using MPC as a way of supporting the approximations V_θ and Q_θ has been proposed and justified in [13]. Hereafter, we detail how this can be done for the specific choice of MPC proposed here. [12]

2.3.4 MPC as a function approximator for RL

We propose to use the action-value function approximate $Q_\theta \approx Q_\star$ obtained from the following MPC scheme parameterized by θ [13]:

$$\begin{aligned} Q_\theta(s, a) = \min_{x,u} \; & \gamma^N V^f\left(x_N, \theta\right) + \\ & \sum_{i=0}^{N-1} \left(\gamma^i l\left(x_i, u_i, \theta\right)\right) && \text{(2.10a)} \\ \text{s.t. } & \forall i = 0, ..., N-1, \\ & x_{i+1} = f(x_i, u_i, \theta) && \text{(2.10b)} \\ & g(u_i) \leq 0 && \text{(2.10c)} \\ & x_0 = s && \text{(2.10d)} \\ & u_0 = a && \text{(2.10e)} \\ & && \text{(2.10f)} \end{aligned}$$

where $x = \{x_0, \ldots, x_N\}$ and $u = \{u_0, \ldots, u_{N-1}\}$ are the primal decision variables, N is the prediction horizon, f is the model dynamics, l and V^f are the stage and terminal costs, respectively.

Constraint (2.10c) represents the input inequality constraints.

In (2.10), θ is the parameters vector that can be modified by RL to shape the action-value function. Under some mild assumptions (see [13] for the technical details), if the parametrization is rich enough, the MPC scheme is able to capture the true optimal action-value function $Q_\star$, value function $V_\star$ and policy $\pi_\star$ jointly, even if the MPC model f does not capture the real system dynamics (2.1).

One can verify that the parameterized value function V_θ that satisfies the Bellman equations can be obtained by solving (2.10) without constraint (2.10e). Moreover, the parameterized deterministic policy π_θ reads as follows:

$$\pi_\theta(s) = u^\star_{k,0}(s, \theta) \tag{2.11}$$

where $u^\star_{k,0}(s, \theta)$ is the first element of $u^\star$, solution of the MPC scheme (2.10) when constraint (2.10e) is removed.

Therefore, the value function $V_\theta(s)$ can be acquired together with the policy $\pi_\theta(s)$ by solving a classic MPC scheme, while the action value function results from solving the same MPC scheme with its first input constrained to a specific value a.

The sensitivity $\nabla_\theta Q_\theta(s, a)$ required in (2.9b) is given by [13]:

$$\nabla_\theta Q_\theta(s, a) = \nabla_\theta \mathcal{L}_\theta(s, a, y^\star) \tag{2.12}$$

where $\mathcal{L}$ is the Lagrange function associated to the MPC (2.10), i.e.:

$$\mathcal{L}_\theta(s, a, y) = \Phi_\theta + \lambda^\top G_\theta + \mu^\top H_\theta \tag{2.13}$$

where Φ_θ is the cost (2.10a), G_θ gathers the equality constraints (2.10b), (2.10d), (2.10e), H_θ collects the inequalities (2.10c) and λ, μ are the associated dual variables. Argument y reads as $y = \{x, u, \sigma, \lambda, \mu\}$ and $y^\star$ is the solution to (2.10). [12] [5]

hyperref

Chapter 3

Simulation

In this section we go through the main work of this master thesis. We start by some introduction of the software packages used for optimization and for the MPC definition. CasADi and Acados have been used. Then all the aspects relative to the implementation, workflow and results of simulation are presented.

3.1 Optimization: CasADi and Acados

From a practical point of view, optimization is finding the best solution to a problem. Mathematically, it is basically finding the minimum (or maximum) of a function, in most cases subject to constraints. Optimization is an extremely important discipline in most fields (physics and engineering; economy, finance and management; ...)

MPC control design is an optimization problem: we have to design the controller in order to:

- minimize the tracking error;
- minimize the command effort;
- minimize the effects of disturbances.

Thus, optimization is fundamental in Model Predictive Control.

3.1.1 CasADi

CasADi is an open-source software framework for numerical optimization. It is a general-purpose tool that can be used to model and solve optimization problems with a large degree of flexibility. In particular, problems constrained by differential equations such as optimal control problems are of special interest. CasADi is

written in self-contained C++, but is most easily used through its interfaces to Python and MATLAB. In our case, the implementation is done in Python. Since its creation, it has been used successfully in applications from multiple fields, including process control, robotics and aerospace. [10]

CasADi started out as a tool for algorithmic differentiation (AD) using a syntax similar to a computer-algebra system (CAS), explaining its name. While state-of-the-art AD is still a key feature of CasADi, the focus has since shifted towards optimization. In its current form, CasADi provides a set of general-purpose building blocks that drastically decreases the effort for implementation of a large set of algorithms for numerical optimal control, without sacrificing efficiency. [10]

The symbolic framework

The core of CasADi is made of a symbolic framework that permits to build expressions and utilize these to specify automatically differentiable functions. These general-purpose expressions have no relation with optimization. First the expressions are created, then they can be utilized to efficiently obtain new expressions for derivatives using AD or be evaluated efficiently, either in CasADi's virtual machines or by using CasADi to generate self-contained C code. CasADi uses a MATLAB inspired "everything-is-a-matrix" type syntax, i.e., scalars are treated as 1-by-1 matrices and vectors as n-by-1 matrices. Besides, every matrix is sparse and stored in the compressed column format. For this kind of symbolic structure, having to do with a single sparse data type makes the tool simpler to learn and maintain.

The subsequent code shows a procedure to load CasADi into the workspace, create two symbolic primitives $x \in \mathbb{R}^2$ and $A \in \mathbb{R}^{2,2}$ and finally the creation of an expression for $e := Asin(x)$ [10]:

```
# Python
from casadi import *
x = SX.sym('x', 2)
A = SX.sym('A', 2, 2)
e = mtimes(A, sin(x))
print(e)
```

Output: $@1 = sin(x_0), @2 = sin(x_1), [((A_0 * @1) + (A_2 * @2)), ((A_1 * @1) + (A_3 * @2))]$

The output should be interpreted as the definition of two shared subexpressions, @1 := sin(x0) and @2 := sin(x1) followed by an expression for the resulting column.

In CasADi, the symbolic expressions can be used to define function objects, classes that behave like conventional functions but are instantiated at runtime. In addition to supporting numerical evaluation, CasADi function objects support symbolical evaluation, C code generation and also derivative calculations. They

can be created by giving a display name and a list of input and output expressions [10]:

```
# Python
F = Function('F',[x,A],[e])
```

This snippet defines a function object by the name "F" with two inputs (x and A) and one output (e), as defined in the previous code segments.

The creation of a function object in CasADi essentially sums up to topologically sorting the expression graph, turning the directed acyclic graph (DAG) into an algorithm that can be assessed. Differently from traditional tools for AD there is no relation between the order in which expressions were created and the order in which they are shown in the sorted algorithm. Instead, CasADi uses a depth-first search to topologically sort the nodes of the DAG. Provided the sorted sequence of operations, CasADi implements two register based virtual machines (VMs), one for each graph representation. The VMs in CasADi are designed in order to achieve high-speed and low overhead; for example, by avoiding memory allocation during numerical evaluation. In such a framework, that is often used for rapid prototyping with many design iterations, fast VMs are important not only for numerical evaluation, but also for symbolic processing, which can amount to a significant part of the total solution time.

C-code generation

However, as an alternative way to evaluate symbolic expressions in CasADi, user can generate C code for the function objects. When compiled with the correct compiler flags, the generated code can be significantly faster than the VMs. Given that the generated code is self-contained C and has no dynamic memory allocation, it is suited to be deployed on embedded systems.

CasADi thus merges together support for modeling with support for optimization. Two classes of optimization problems are supported: nonlinear programs (NLPs) and conic optimization problems. The latter class includes both linear programs (LPs) and quadratic programs (QPs). [10]

3.1.2 Acados

Acados software package is an ensemble of solvers for fast embedded optimization intended for fast embedded applications. Its interfaces to higher-level languages make it useful for quickly designing an optimization-based control algorithm. This is achieved by combining together different algorithmic components that can be readily connected and interchanged. Since the core of Acados is written on top

of a high-performance linear algebra library, it does not sacrifice computational performance. Thus, it is able to give both flexibility and performance through modularity, without the need to rely on automatic code generation, which facilitates maintainability and extensibility. The main features of acados are: efficient optimal control algorithms targeting embedded devices implemented in C, linear algebra based on the high-performance BLASFEO library, user-friendly interfaces to Matlab and Python, and compatibility with the modeling language of CasADi.

One challenge in developing software for embedded optimal control is found in the trade-off between flexibility, memory usage and speed. Many of the software packages are built on automatic code generation (such as Acado, the precursor of Acados). One reason for that is to have self-contained efficient linear algebra routines. Often however, the size of the problem and the choice of algorithms are then fixed for one specific optimal control instance. This leads to a loss of flexibility, other than more expenses and harder maintenance. The recently developed high-performance linear algebra package BLASFEO often outperforms code-generated routines. Given that the linear algebra operations usually amount for most of the computational complexity, Acados based on BLASFEO represents a better trade-off. [14]

Another relevant aspect of embedded optimal control software that affects flexibility, memory and run-time is the choice of the modeling language and corresponding automatic differentiation tool. Several modeling languages exist, such as Mathematica, sympy or the MATLAB Symbolic Toolbox. Commonly, these languages utilize expression trees to represent mathematical functions. This leads usually to a large code size, high memory usage and slow evaluation of higher-order derivatives. For simple models, they are usually good. But as soon as the complexity of the model gets larger, they start to face problems. On the opposite, CasADi modeling language is based on expression graphs. This in many cases leads to shorter instruction sequences and to smaller, usually faster code, which makes it more suitable for embedded applications. Also, it is free and open-source software. For all this motivations, CasADi is chosen for modeling nonlinear functions and differential-algebraic equations. Furthermore, Acados supports the use of hand-written or code-generated dynamic models as C source files.

Main Features

In summary, acados is a quite new software package for embedded optimal control that offers the following main features:

- efficient optimal control algorithms implemented in C.
- modular architecture enabling rapid prototyping of solution algorithms.

- interfaces to Python (our choice) and Matlab.
- high-performance linear algebra based on BLASFEO.
- compatible with CasADi expressions.
- deployable on a variety of embedded devices.
- publicly available as permissively licensed free and open-source software.

[9]

Python Interface

Acados is built in order to be user-friendly at a high level and efficient at a low level. To achieve a balance of these properties, it is structured on a base library written in C which provides functionality to the Python and Matlab interfaces. The core library of Acados contains for the most part an ensemble of modules. Every module has corresponding data types and variants of solvers, together with helper functions for memory management. To build applications by utilizing the core library directly can be cumbersome and error-prone, since a lot of details need to be considered. This is because it is designed in order to be efficient and flexible. To fully serve the specific needs of the end user, different interfaces are possible to the core of Acados. Our project is based on the Python interface.

In order to define the Optimal Control Problem (OCP) through the Acados modules (cost, constraints and dynamics), the main challenge is to pass the generally nonlinear functions and their derivatives to these modules. The Python and Matlab interfaces of Acados use CasADi as a modeling language. Basically, CasADi is used to define all the nonlinear portions of the OCP. The Acados high level interfaces can use CasADi's code generation and algorithmic differentiation to generate the C functions that are required for each Acados module. One of the most important advantages of using CasADi as a modeling language is that the solution behavior of Acados can be easily compared with the solutions coming from the numerous optimization tools interfaced with CasADi.

Workflow

Once the OCP to be solved is described through the domain-specific language implemented by the high-level interfaces, a human readable self-contained C project that makes use of templated code can be generated. The generated structure contains all the C code that is needed for function and derivative evaluations. These are generated through CasADi and the C code necessary to set up the NLP solver using the Acados C interface. Furthermore, a comprehensive system for its

compilation is generated. Be aware that this kind of code generation is inherently different from the one in ACADO, the precursor of Acados. This is because the templated code uses only the functions exposed by the C interface of Acados. In contrast to this, ACADO generated solvers are standalone C projects that are extremely problem specific and do not rely on a common library. So, basically, the advent of a new high-performance dense linear algebra package for embedded computations, BLASFEO, makes it possible to circumvent or even repell the need for the code generation. The highly efficient linear algebra routines, optimized for a range of computer architectures, often outperform code generated linear algebra. [14]

With the workflow described above, it is fairly possible to achieve a self-contained, high-performance solver that can be easily deployed on embedded hardware starting from a description of the OCP in a high-level language.[9]

So, to sum up, Acados is a fast and embedded solver for nonlinear optimal control. It is the successor of the ACADO software package developed at KU Leuven and University of Freiburg by the team of Prof. Moritz Diehl [9]. It provides an ensemble of computationally efficient building blocks suited for optimal control and estimation problems. Among others, it implements: modules for the integration of ordinary differential equations (ODE) and differential-algebraic equations (DAE), interfaces to state-of-the-art QP solvers like HPIPM, qpOASES, qpDUNES and OSQP, condensing routines and nonlinear programming solvers based on the real-time iteration framework. The back-end of Acados uses the high-performance linear algebra package BLASFEO, in order to boost computational efficiency for small to medium scale matrices typical of embedded optimization applications. MATLAB and Python interfaces can be used to conveniently describe optimal control problems and generate self-contained C code that can be readily deployed on embedded platforms.[14]

3.2 Code implementation and Simulation Environment

This is the part of the project where I have been spending most of my time and work. To make it quick: programming, and a lot of it. The first goal of this step was to have a simulation drone up and running with an MPC controller able to successfully target aggressive trajectories, so that then we could transfer everything into reality in the laboratory and have some real flight experimental data. The kind of trajectory considered for agile flight was the lemniscate shape (infinity shape), together with the circle trajectory. Both the trajectories are shown in figure 3.1. As already discussed previously, due to the writer knee injury and relative surgery

in the middle of the work, the real flight experiments became an unfeasible dream eventually. So in the end the whole project shifted towards extensive simulation validation.

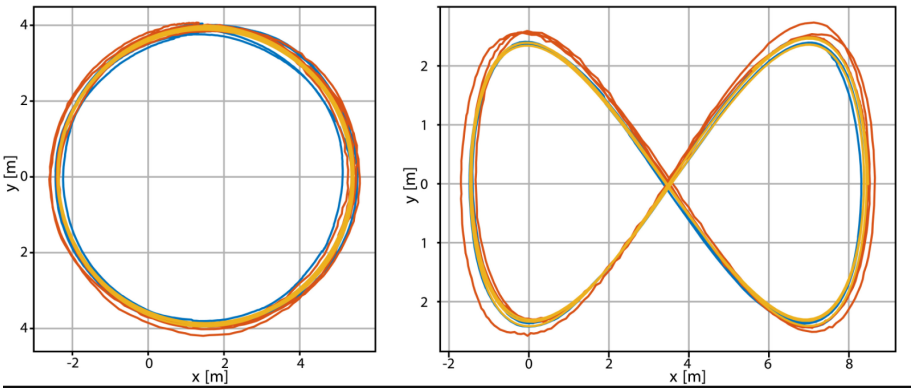

Figure 3.1: Examples of circle and lemniscate trajectories

The second goal was to then try to implement the Reinforcement Learning on top of the MPC, thus having some parameters of the MPC learned from experience.

All the code implementation can be found on my Github, at the following link: `https://github.com/DematteisGiacomo/mpc_rmf_ws`.

3.2.1 MPC implementation

Let's first focus on the first part, the implementation of the MPC. Basically, I had two code bases, both with some complementary relevant aspect to my project, that I needed to thoroughly understand, modify according to my needs and finally merge together for the final result. The first code base comes from [1]. It contained everything related to the implementation of the MPC and relative optimization, Acados, Casadi. The accompanying Github repo to the paper was quite constructive with guidelines on how to simulate their code, which was entirely implemented in Python. So what I first did was to try to get that up and running (with no modifications, their drone and model) in order to have a solid baseline with MPC following the types of trajectories we talked about above. At this stage, basically what I could do was make the drone follow circle/lemniscate trajectory, and showed tracking performance before and after. The challenge then was to adapt the case we were simulating to the platform that the NTNU drone group was providing: take the MPC for fast trajectory tracking developed by [1] and mount it on our drone setup, a drone made for navigation with a simple PID as controller and keep track of the results in terms of performance.

So here comes the second code base resulting from all the implementations and

relative infrastructure/hardware of the NTNU drone group. It contained all the relevant information of our drone, used for navigation, equipped with a PID as controller and with all the necessary links once it was needed to move into the laboratory. Roughly speaking, what I did was take the NTNU code and replace the PID with the MPC.

The final implementation makes use of both C++ and Python. It is built on ROS Noetic, Gazebo, Ubuntu 20.04, Python 3.8. The code is a combination of C++ and Python. The code regarding the MPC was implemented in Python, since it makes use of the related Acados interface.

Simulation Workflow

The MPC computes the series of reference state and reference input along the whole trajectory, to be accounted at runtime in the cost function. As a result, as we can see from the figure 3.2, we obtain the reference state (position, velocity, attitude, rate) in the three dimensions and the reference input (the four rotor thrusts) for the duration of the whole simulation. Additionally in the figure, it is showed a plot of the target lemniscate trajectory (loop trajectory can also be an option).

After this, the simulation is ready to run. The simulation environment used is Gazebo (figure 3.3). Gazebo is a 3D robot simulator. Its objective is to simulate a robot, giving you a close substitute to how your robot would behave in a real-world physical environment.

Another tool used for simulation is Rviz. Rviz (short for "ROS visualization") is a 3D visualization software tool for robots, sensors and algorithms. It enables you to see the robot's perception of its world (real or simulated). The purpose of Rviz is to enable you to visualize the state of a robot. It uses sensor data to try to create an accurate depiction of what is going on in the robot's environment. From figure 3.4, the left panel is the Displays panel. It has a list of plugins. These plugins enable you to view sensor data and robot state information.

A lot of times the two of Gazebo and Rviz get confused one with the other. After all, both programs enable you to view a simulated robot in 3D and both are two popular software tools that are used with ROS.

The difference between the two can be summed up in the following excerpt from Morgan Quigley (one of the original developers of ROS) in his book Programming Robots with ROS:

"Rviz shows you what the robot thinks is happening, while Gazebo shows you what is really happening."

At the end a whole trajectory simulation, results are given, providing the performance in terms of mismatch between reference and simulated values of state

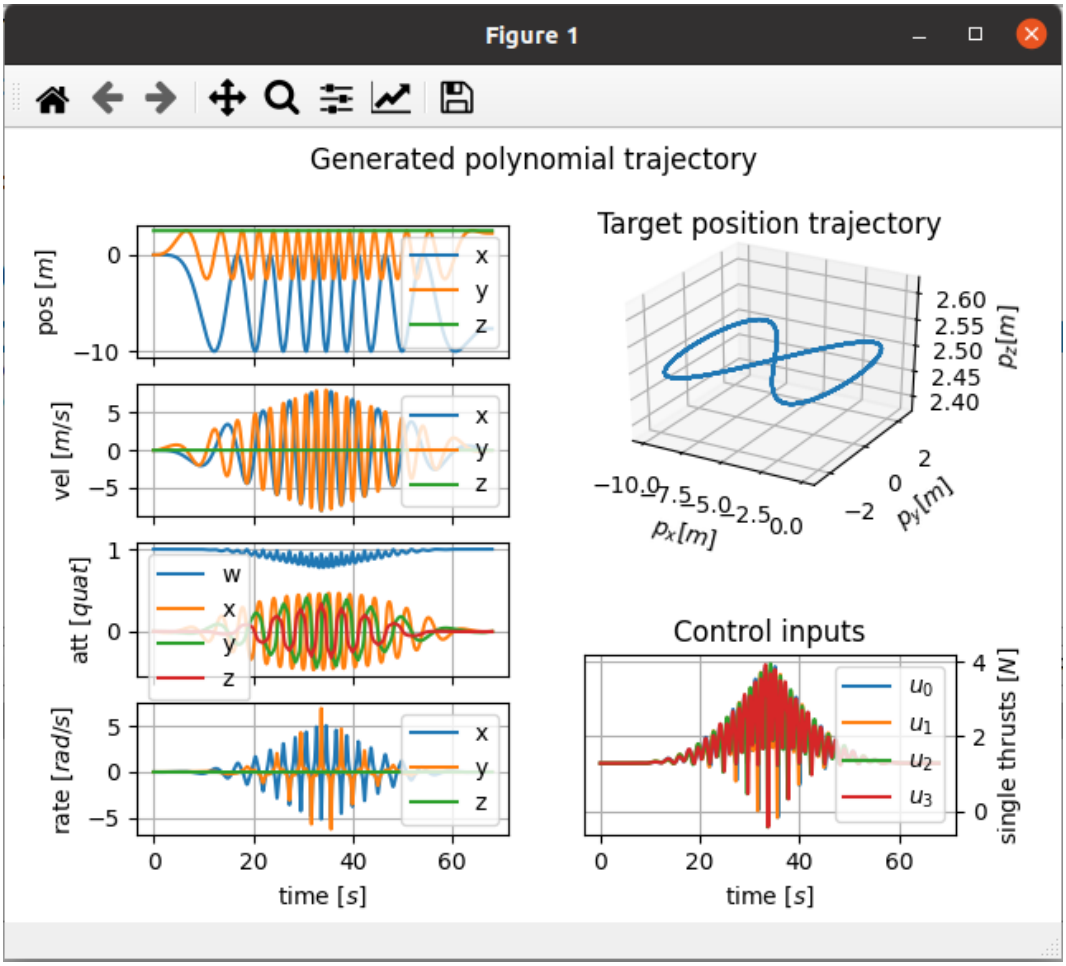

Figure 3.2: Plot of the reference state (position, velocity, attitude, rate) in the three dimensions and of the reference input (the four rotor thrusts), together with a plot of the desired trajectory (lemniscate in this case)

and input. Root Mean Square Error (RMSE) is considered as error metric. Plots can be given, that show the results of simulation comprehensive of the plot of the reference state versus the simulated, RMSE, and the input difference.

ROS

The Robot Operating System (ROS) is a set of software libraries and tools that help to build robot applications. The ROS runtime "graph" is a peer-to-peer network of processes (**nodes**), potentially distributed across machines, coupled using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous communication over services and

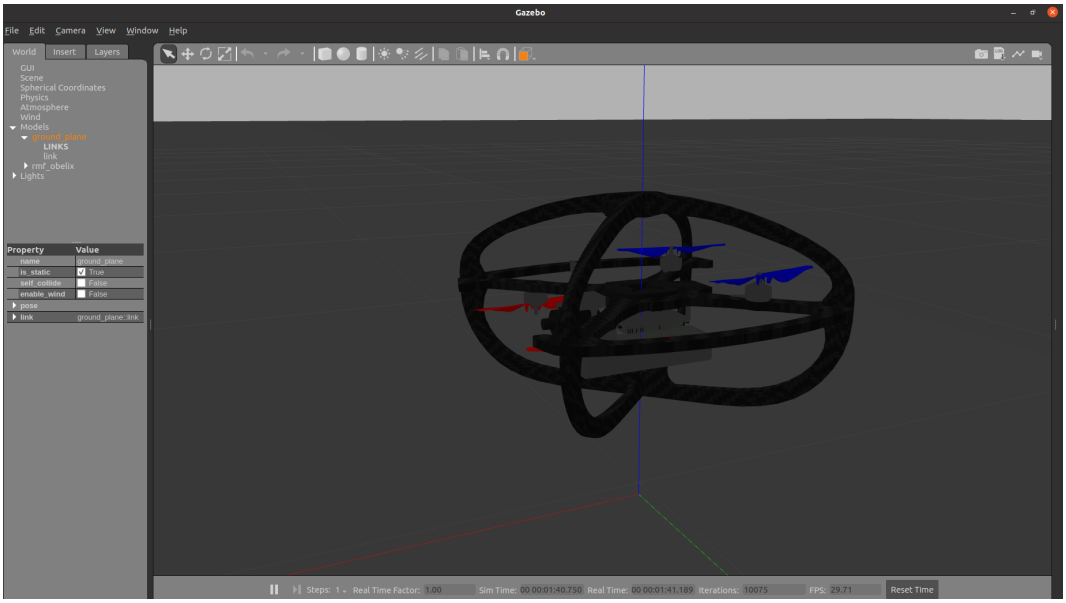

Figure 3.3: Gazebo simulation environment.

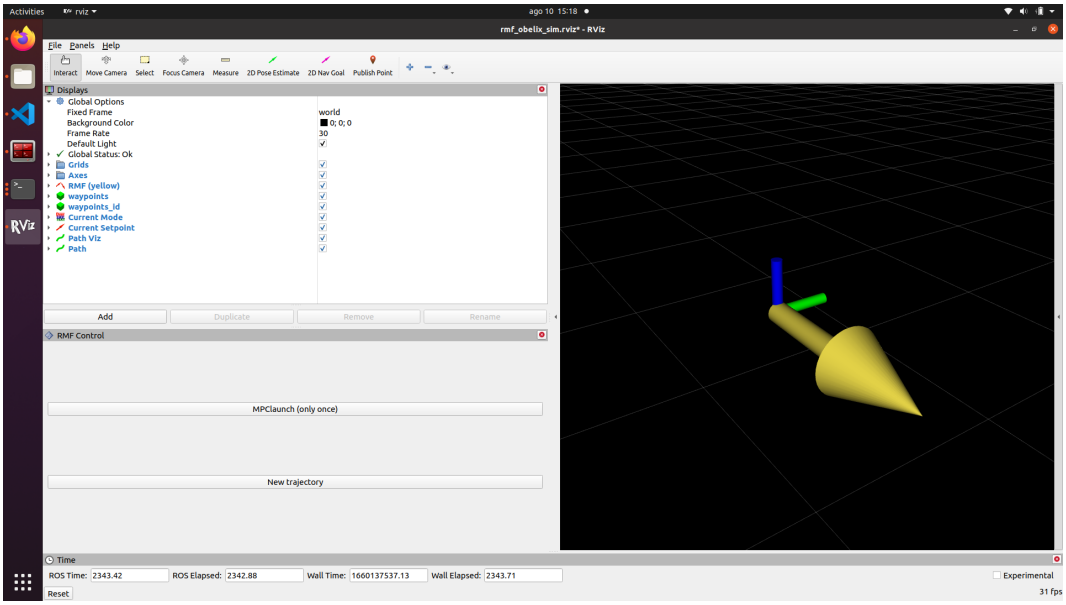

Figure 3.4: Rviz 3-D visualization software tool and relative user interface.

asynchronous streaming of data over **topics** (what I mostly used).

In figure 3.5, the ROS graph during simulation is showed.

Let's have a closer look at the picture and explain further. At the beginning

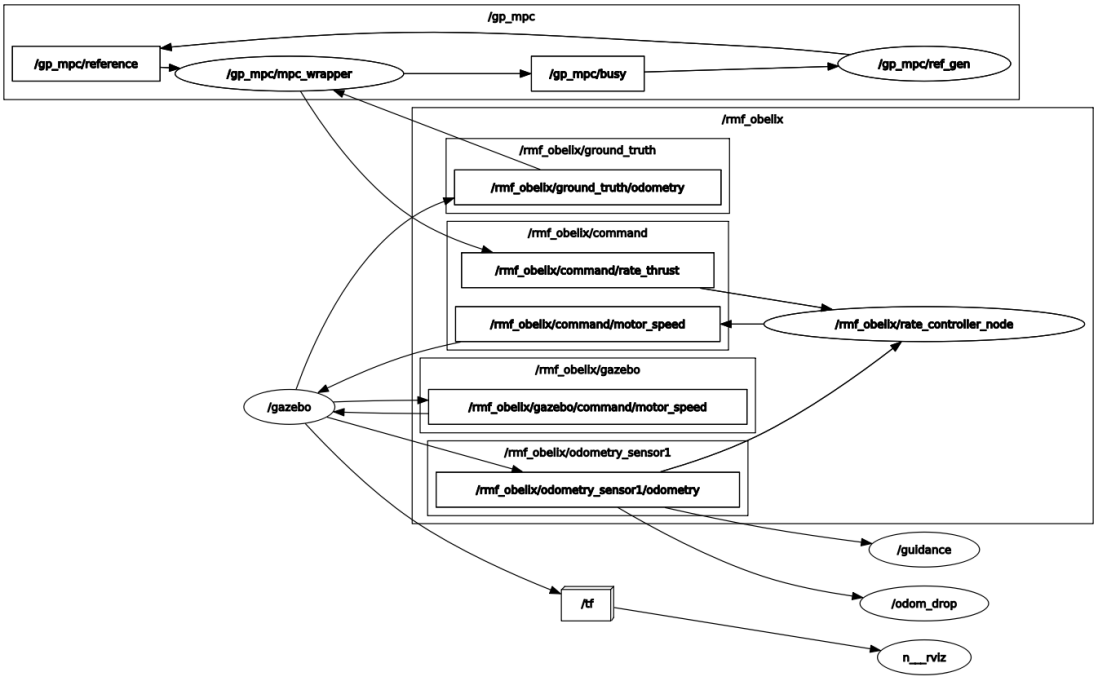

Figure 3.5: ROS graph of the ongoing processes (nodes) and the channels of data communication between them (topics).

of the simulation the reference trajectory is computed (by /ref_gen node) and fed to the MPC (through the /reference topic). At this point the drone starts moving. The MPC node (/mpc_wrapper) is fed with the odometry (current state: position, attitude, velocity, rate) by gazebo (through /odometry topic) and computes the body rates and total thrust commands. In principle it can also directly compute the four single thrusts of the rotors, but that would not be feasible when the hardware comes into play for safety reasons (it would require to by-pass some safety mechanism of the real autopilot). The MPC then passes the body rates and thrust commands (through /rate_thrust topic) to a body rate controller (/rate_controller_node). This controller basically simulates the real autopilot hardware and would be replaced by it in the lab implementation. This controller then feeds gazebo with the four motor speeds (equivalent to the single thrusts). To close the circle, gazebo feeds again the MPC with a new odometry and keep looping.

Note that in the real implementation, the gazebo simulator disappears. Instead, the four motor speeds are fed to the Electronics Speed Controller (ESC) of the motors for the Pulse Width Modulation (PWM) commands. And the odometry is given by a Motion Capture (Mocap) system.

Code example

```
def quad_dynamics(self, rdrv_d):
    """
    Symbolic dynamics of the 3D quadrotor model. The state consists on: [p_xyz, a_wxyz, v_xyz, r_xyz]^T, where p
    stands for position, a for angle (in quaternion form), v for velocity and r for body rate. The input of the
    system is: [u_1, u_2, u_3, u_4], i.e. the activation of the four thrusters.

    :param rdrv_d: a 3x3 diagonal matrix containing the D matrix coefficients for a linear drag model as proposed
    by Faessler et al.

    :return: CasADi function that computes the analytical differential state dynamics of the quadrotor model.
    Inputs: 'x' state of quadrotor (6x1) and 'u' control input (2x1). Output: differential state vector 'x_dot'
    (6x1)
    """

    x_dot = cs.vertcat(self.p_dynamics(), self.q_dynamics(), self.v_dynamics(rdrv_d), self.w_dynamics())
    return cs.Function('x_dot', [self.x[:13], self.u], [x_dot], ['x', 'u'], ['x_dot'])
def p_dynamics(self):
    return self.v
def q_dynamics(self):
    return 1 / 2 * cs.mtimes(skew_symmetric(self.r), self.q)
def v_dynamics(self, rdrv_d):
    """
    :param rdrv_d: a 3x3 diagonal matrix containing the D matrix coefficients for a linear drag model as proposed
    by Faessler et al. None, if no linear compensation is to be used.
    """

    f_thrust = self.u * self.quad.max_thrust
    g = cs.vertcat(0.0, 0.0, 9.81)
    a_thrust = cs.vertcat(0.0, 0.0, f_thrust[0] + f_thrust[1] + f_thrust[2] + f_thrust[3]) / self.quad.mass

    v_dynamics = v_dot_q(a_thrust, self.q) - g

    if rdrv_d is not None:
        # Velocity in body frame:
        v_b = v_dot_q(self.v, quaternion_inverse(self.q))
        rdrv_drag = v_dot_q(cs.mtimes(rdrv_d, v_b), self.q)
        v_dynamics += rdrv_drag

    return v_dynamics
def w_dynamics(self):
    f_thrust = self.u * self.quad.max_thrust

    y_f = cs.MX(self.quad.y_f)
    x_f = cs.MX(self.quad.x_f)
    c_f = cs.MX(self.quad.z_l_tau)
    return cs.vertcat(
        (cs.mtimes(f_thrust.T, y_f) + (self.quad.J[1] - self.quad.J[2]) * self.r[1] * self.r[2]) / self.quad.J[0],
        (-cs.mtimes(f_thrust.T, x_f) + (self.quad.J[2] - self.quad.J[0]) * self.r[2] * self.r[0]) / self.quad.J[1],
        (cs.mtimes(f_thrust.T, c_f) + (self.quad.J[0] - self.quad.J[1]) * self.r[0] * self.r[1]) / self.quad.J[2])
```

Figure 3.6: Sample of CasADi snippet containing the definition of the model dynamics 2.1

In figure 3.6 there is an example of the CasADi code in the implementation of the optimizer. The sample contains the definition of the model dynamics as described in 2.1 in CasADi Python-interface symbolic language.

MPC simulation results

The results coming from this simulation setup validate the use of an MPC as controller for agile drone flight control. For this kind of application, the performance achieved are quite satisfactory from a wide range of top velocities. In simulation,

we can achieve top velocities of 15 m/s, without compromising the stability of the system and without having a drastic decrease in performance. This is valid for both lemniscate and loop trajectories.

Briefly, as a general result of these extensive simulations, we come to the conclusion that MPC represent already a very good technology for this kind of problem, confirmed by our specific model dynamics.

Further details will be shown in the results chapter. Just to give the reader a general idea of the performance achieved, the RMSE for the lemniscate trajectory with 10 m/s as top speed is about 0.20 m. Similar results are achieved in the loop case, with a RMSE close to 0.23 m.

3.2.2 RL implementation

Once set up the MPC simulation environment, the next step in the project was to try to set up a Reinforcement Learning framework, in order to see if the performance could be improved even further. As discussed in the theory chapter, the basics of this method is to adjust some specific parameters of the MPC formulation through learning based on the data collected experimentally. In our case, the choice on the parameters to be learned from experience fell on the values of the diagonal matrices (Q and R) specified in the cost function. This two matrices identify the trade-off between performance and input effort. Their choice heavily impact the behaviour of simulation. They are naturally subject to manual tuning. But they comprise a large number of parameters, so the tuning can be cumbersome and it is not guaranteed to provide optimal results.

The two matrices dimensions are related to the state and input dimension. So, for the state matrix Q, the dimensions is 13x13 diagonal. Whilst for the input matrix R is 4x4 diagonal. This amounts to a total of 17 parameters subject to tuning. This should provide a parametrization vast enough in order to justify the use of the RL framework described previously. Theoretical foundations can be found in [13].

Another set of parameters that could be subject to training could be the parameters belonging to the model dynamics. This study could be interesting, since the model dynamics is of course subject to approximation and mismatches. In this work, we haven't implemented the RL on the model parameters. This is something that could be left for future work as well.

A curious entity to be specified in this framework is the reward function. In reinforcement learning problems, this is a pivotal choice. Reward function can have many shapes and forms. In our case, a common choice is to chose a reward function of the same form of the cost function in the definition of the MPC. Basically, we are given rewards on the basis of the performance in the reference tracking and

on the basis of the input effort. The smaller the effort and the better the tracking error, the higher is the reward. In figure 3.7, an example of reward plot of one simulation experiment is shown.

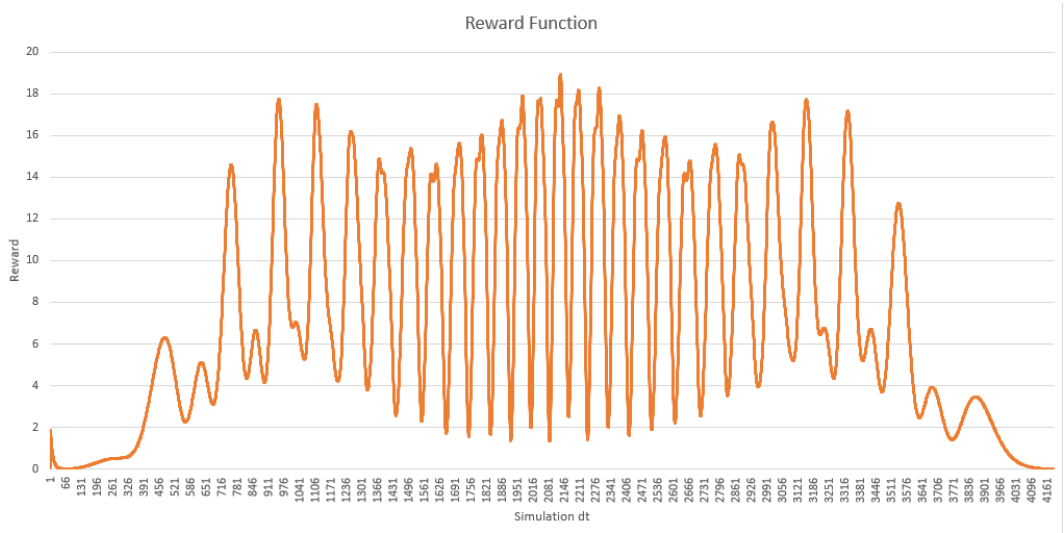

Figure 3.7: The pictures shows the trend of rewards across one simulation experiment.

The workflow of this simulation environment is quite straightforward. Basically, on top of the MPC simulation, a RL procedure is instantiated upon completion of every trajectory tracking experiment. Parameters are updated accordingly and a new experiment can be undertaken. So the flow is the interleaving of experiment and training, mutually influencing each other.

Regarding the practical implementation, Python code containing the definition of the training function, lagrangian and sensitivity computation plus other relevant functionalities, was added to the original MPC code base.

To sum up, what we get eventually is a well functioning framework, that allows to easily iterate between simulation with data collection and training with parametrs update.

RL simulation results

Regarding the results of this framework, things start to get a bit shaky. The implementation of the whole framework was quite challenging in terms of development and many things were to be taken under account. This fatiguing implementation was followed by an even more nasty debug phase. Unfortunately, some bug is still to be fixed, and the correct behaviour is not perfectly achieved yet. Basically we

cannot really appreciate a performance increase due to learning. The RMSE is not decreased, and a feasible parameter update require a too small learning rate such that barely any learning is present. However, it must be taken under account that the behaviour of the drone at high speeds looks more stable. So, this may be indicative of the fact that we are anyway in the right direction.

So, the results of this part are not satisfactory yet. More debugging is needed. Unfortunately due to a lack of time, this work is not able to show that learning is actually happening in the right direction. Anyway, results coming from this framework are shown in the results chapter, for the curious reader to explore.

Chapter 4

Lab experiments

Lastly, we here explain the part of the project related with real flight experiment. As already discussed, this part became unfeasible eventually, but anyway a lot of work was put in preparation for this step. So, we left here following a discussion of the method we would have been used to tackle the problem. One could also look at it as a guide for a possible future work.

The goal is to compare the performance of our RL/MPC on a real quadrotor. We use a custom quadrotor. We run the controller on a laptop computer and send control commands in the form of collective thrust and desired body rates to the quadrotor through wifi. The quadrotor flies in an indoor arena equipped with an optical tracking system that provides pose estimates at 100 Hz. As in the simulation experiments, we compare the tracking error along both circle and lemniscate trajectories.

4.1 Setup

4.1.1 Drone

The type of drone used for the experimental part is the Resilient Micro Flyer (RMF) (figure 4.1), a new type of collision-tolerant small aerial robot tailored to traversing and searching within highly confined environments. The robot is particularly lightweight and agile, while it implements a collision-tolerant design which renders it resilient during forcible interaction with the environment. It weighs less than 500g and is capable of 15min of endurance. [15]

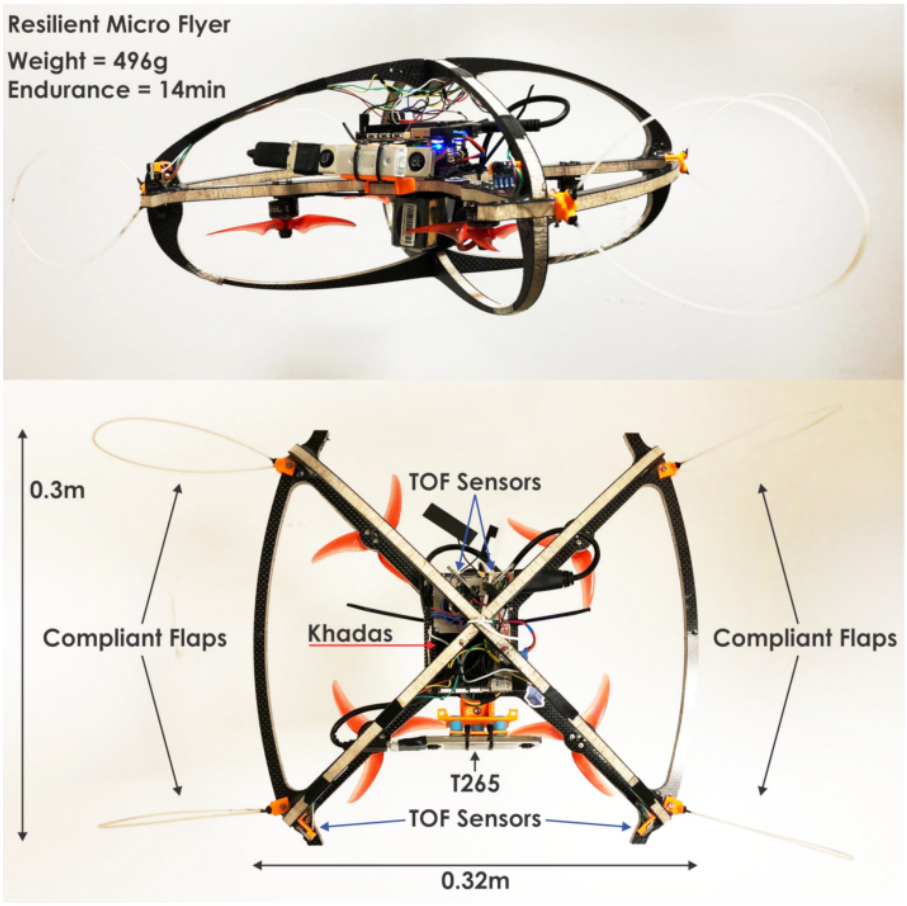

Figure 4.1: Resilient Micro Flier [15].

4.1.2 PX4 autopilot and Khadas vim3

Px4

The "brain" of the drone is called an autopilot. It consists of flight stack software running on vehicle controller ("flight controller") hardware, often a Pixhawk (also in our case). In our case it implements the body rate controller. It receives the body rates commands from the MPC via wifi and computes the motors speed to be fed to the ESCs. PX4 is a professional autopilot for drone control. It is developed by world-class developers from industry and academia, and supported by an active world wide community. It powers various kinds of vehicles from racing and cargo drones to ground vehicles and submersibles. [16]

Figure 4.2: Px4 autopilot module

Offboard Control: On-board processor and wifi link to ROS

The idea behind off-board control is to be able to control the PX4 flight stack using software running outside of the autopilot: a small computer mounted onto the vehicle connected to the autopilot through a UART to USB adapter while also having a wifi link to a ground station running ROS. This is done through the MAVLink protocol and MAVROS package.

MAVlink and MAVROS

MAVlink MAVLink is a very lightweight messaging protocol that has been designed for the drone ecosystem. PX4 uses MAVLink to communicate with the ground station and as the integration mechanism for connecting to drone components outside of the flight controller: companion computers, MAVLink enabled cameras etc. The protocol defines a number of standard messages and microservices for exchanging data.

MAVROS MAVROS is a ROS package that enables MAVLink extendable communication between computers running ROS for any MAVLink enabled autopilot, ground station, or peripheral. MAVROS is the "official" supported bridge between ROS and the MAVLink protocol.

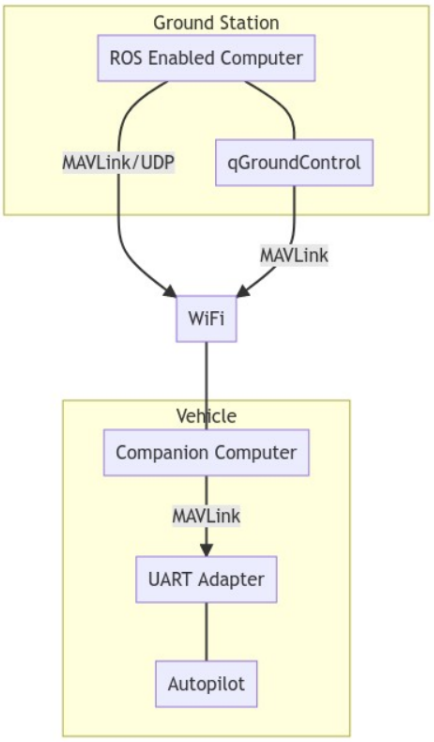

Figure 4.3: Scheme of Offboard Control.

Khadas Vim3 companion computer

PX4 can connect to companion computers (Khadas Vim 3 in our case) using any configurable serial port. Message are sent over the link using the MAVLink protocol. In order to receive MAVLink, the companion computer needs to run some software talking to the serial port. In our case it runs the MAVROS node to communicate via wifi to ROS nodes of the ground station.

The VIM3 single board computer (SBC) is the latest addition to the popular Khadas VIM series. It has a powerful Amlogic A311D SoC: x4 Cortex A73 performance-cores (2.2Ghz) and x2 Cortex A53 efficiency-cores (1.8Ghz) are merged into a hexa-core configuration and fabricated with a 12nm process to maximise performance, thermal and electrical efficiency. It is the size of a credit-card with everything already built-in.

4.1.3 Motion Capture

Computer vision techniques enable computers to use visual data to make sense of their environment. PX4 uses computer vision systems in order to support

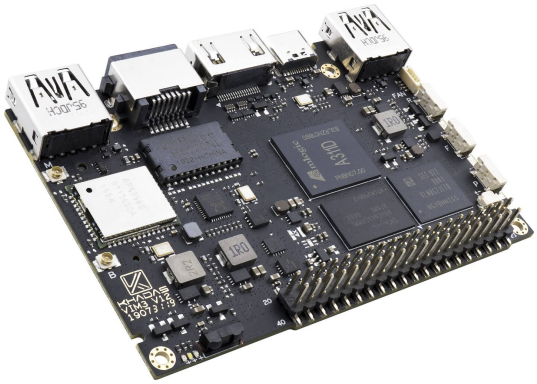

Figure 4.4: Khadas vim3 board

pose/velocity estimation. Motion Capture (MoCap) is a computer vision technique for estimating the 3D pose of a vehicle using a positioning mechanism that is external to the vehicle. It is primarily used for indoor navigation. It is commonly used to navigate a vehicle in situations where GPS is absent (e.g. indoors), and provides position relative to a local coordinate system. MoCap systems most commonly detect motion using infrared cameras, but other types of cameras, Lidar or Ultra Wideband (UWB) may also be used. In general, it is highly recommended to send motion capture data via an onboard computer for reliable communications.

4.1.4 ROS

We can explain how all what is said before translates into the ROS graph at runtime when running the experiment in the MoCap lab. Body rates are sent to mavros node which then sends via mavlink to autopilot. Note that mavros node runs onto the companion computer. And the MoCap provides the odometry through a /qualisys odometry topic.

Figure 4.5: Picture of a typical Motion Capture laboratory setup

Chapter 5

Results

5.0.1 MPC results

As already noted, the results deriving from the simulation setup with the MPC prove the validity of the utilization of an MPC as controller for agile drone flight control. For the specific application considered, we are able to achieve more than satisfactory performance with top velocities up to 15 m/s. This is obtained with great stability properties of the system under inspection and with performance that do not see drastic decreases. All this is proven for both types of trajectories.

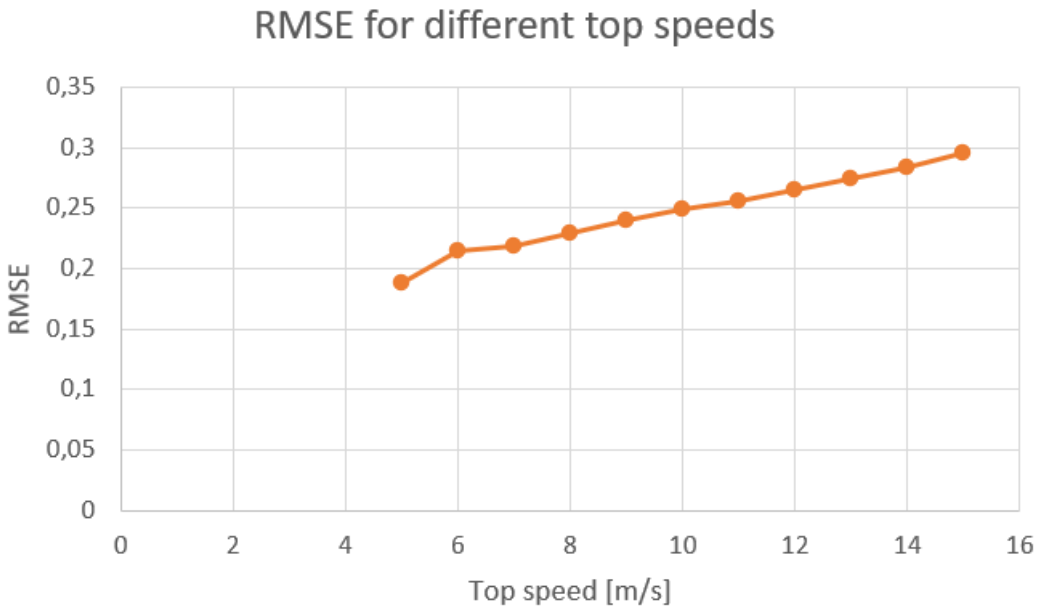

Figure 5.1: The picture shows the trend of RMSE in relation with different top speed simulations.

As discussed, the main observation to be casted is that MPC represent an extremely valid type of controller for this kind of problem. It is proven that MPC is able to guarantee all the characteristics than it possesses, such as reliabilty, performance, safety. All this while satisfying constraints on the input.

Further details are here shown by means of some simple plots. Note that the results for the lemniscate trajectory are presented, but a very similar discussion can be obtained from the results of the loop trajectory. We start by examining the RMSE at different velocities for the lemniscate trajectory. As we can see from figure 5.1, the overall errors are quite good in term of precision and performance. It can be clearly observed how the performance decreases at higher speeds. This comes directly from the fact that at high speeds, the turbulences and disturbances coming from unknown effects to the models have an increasing impact.

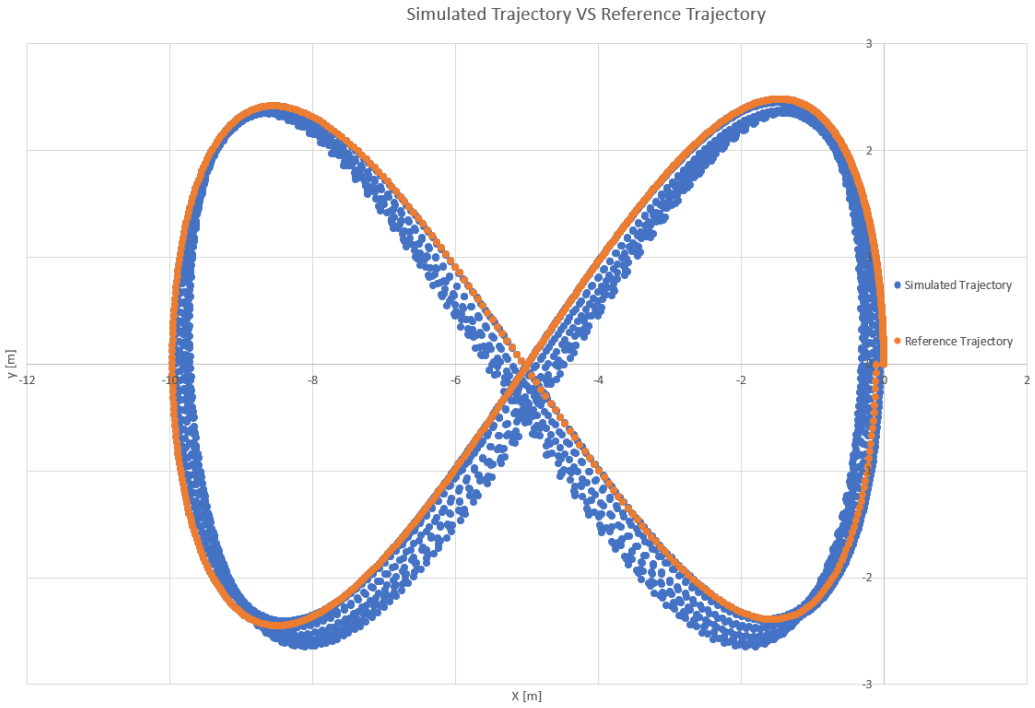

Figure 5.2: The picture shows a comparison between the reference trajectory and the simulated one in the x-y plane.

An another interesting plot is the one in figure 5.2 where the reference trajectory in the x-y plane is compared with the simulated one. In orange we have the reference, while the simulated trajectory is in blue. The trajectory shown here is computed for a top speed of 10 m/s. Note that the simulated trajectory has multiple loops since it start from having zero velocity up to top velocities and

then decelerates back to zero. The whole execution may take a few loops before completion.

Overall, we can conclude that the performance of the MPC is really satisfying since we have to take into account the fact that the drone is moving at extremely high speeds and accelerations compared to the standard application. The drone is normally employed for navigation task that are not concerned with extreme performance at high speeds. Small errors of about 0.25m on average are something far more than acceptable.

5.0.2 RL results

As already stated, the results of the RL framework are not really achieving the desirable outcome yet. The first desired outcome would be to have an MPC that trains itself from experience updating its parameters. Further, the second desired outcome would be to actually observe an improvement in the performance of our simulation. The first outcome is achieved. We obtained a nice running framework for training an MPC through RL by using experimental data. The second objective, however, is not observable yet. More debugging is probably still needed in order to appreciate an increase in the performance.

The implementation of the whole framework was quite challenging in terms of development and many things were to be taken under account. On top of this tiring construction, a lot of debugging was needed for it to work properly. Unfortunately, some bug is still to be fixed, and the correct behaviour is not perfectly achieved yet. However, it must be noted that the behaviour of the model at high speeds seems to have a more stable behaviour. So, this may be proof of the fact that we are heading in the right direction.

To sum up, we are not really able to see a performance increase due to the learning process. The RMSE is not improved over the learning iterations. Further, in order to make the training feasible for the solvers, it is required a too small learning rate such that barely any learning can be considered to be present.

So, the results of this part are not satisfactory yet. More work would be needed in order to fix the code. Due to lack of time however, this work is not able to show that learning is actually happening in the right direction.

Anyway, results coming from this framework are shown in figure 5.3. We can see how the RMSE is barely affected after five training instances. This is true also for a larger number of training iterations.

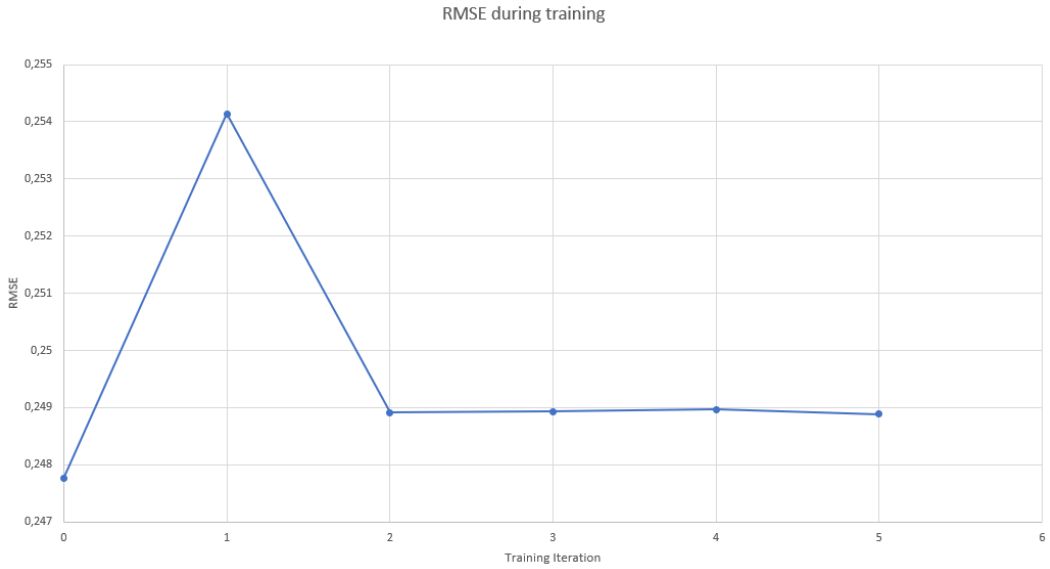

Figure 5.3: The picture shows the trend of the RMSE during across 5 training iterations.

Chapter 6

Conclusion

Aerodynamic forces make quadrotors trajectory tracking at high-speed extremely challenging. High-speed turbulences have in fact a major impact on performance loss.

Learning based MPC has been proven to be powerful for tackling different control problems. The goal of this work was to see if performance could be improved by adopting this kind of framework. MPC together with RL is used to tackle the problem. We make use of RL to offline tune the MPC formulation using the data obtained from the system.

We have been proven that MPC alone is a suitable solution for this kind of applications, providing good and satisfactory results. We prove that MPC is an optimal control method that exploits a dynamic model of the platform and provides constraint satisfaction.

Further, a great amount of work was related with optimization. Optimization is a central tool for many modern applications in technology, industry and research. So, having the privilege to learn how to identify, formulate and solve complex non-linear constrained optimization problems is something that should not go unnoticed.

Regarding the learning part of the project, conclusions are a bit harder to be come by. The framework is extremely challenging and provides in theory an extremely interesting tool to deal with problems facing an agent with goals cast into an environment. In relation to our specific application, the drone behaviour at high speed does look more stable. However, the performance on trajectory tracking is not perceptible yet. So, practical results proving the effectiveness of the framework are not reached.

Anyway, the objective of implementing a running RL/MPC framework was achieved, which is maybe the most relevant thing for a master thesis project.

Possible future work could be related with the real-flight implementation, as described in chapter 4, and with the fixing/tuning of the RL code.

Bibliography

[1] Guillem Torrente and Elia Kaufmann and Philipp Foehn and Davide Scaramuzza. «Data-Driven MPC for Quadrotors». In: *CoRR* abs/2102.05773 (2021). arXiv: {2102.05773}. URL: %7Bhttps://arxiv.org/abs/2102.05773%7D (cit. on pp. 1, 4, 6, 20).

[2] Yunlong Song and Davide Scaramuzza. «Policy Search for Model Predictive Control with Application to Agile Drone Flight». In: *CoRR* abs/2112.03850 (2021). arXiv: 2112.03850. URL: https://arxiv.org/abs/2112.03850 (cit. on pp. 2, 11).

[3] Simon Newman John M. Seddon. *Basic Helicopter Aerodynamics, 3rd Edition*. Wiley, 2011 (cit. on p. 3).

[4] Robert Mahony, Vijay Kumar, and Peter Corke. «Multirotor Aerial Vehicles: Modeling, Estimation, and Control of Quadrotor». In: *IEEE Robotics Automation Magazine* 19.3 (2012), pp. 20–32. DOI: 10.1109/MRA.2012.2206474 (cit. on p. 3).

[5] Stephen J. Wright Jorge Nocedal. *Numerical Optimization*. Springer, 2006 (cit. on pp. 5, 13).

[6] Moritz M. Diehl James B. Rawlings David Q. Mayne. *Model Predictive Control: Theory, Computation, and Design, 2nd Edition*. Nob Hill Publishing, 2022 (cit. on pp. 5, 10).

[7] Max Schwenzer, Muzaffer Ay, Thomas Bergs, and Dirk Abel. «Review on model predictive control: an engineering perspective». In: *The International Journal of Advanced Manufacturing Technology* 117 (Nov. 2021), pp. 1–23. DOI: 10.1007/s00170-021-07682-3 (cit. on p. 5).

[8] Moritz Diehl, Hans Bock, Holger Diedam, and Pierre-Brice Wieber. «Fast Direct Multiple Shooting Algorithms for Optimal Robot Control». In: *Lecture Notes in Control and Information Sciences* 340 (July 2007). DOI: 10.1007/978-3-540-36119-0_4 (cit. on p. 6).

[9] Robin Verschueren et al. «acados: a modular open-source framework for fast embedded optimal control». In: (Oct. 2019) (cit. on pp. 6, 18, 19).

[10] Joel Andersson, Joris Gillis, Greg Horn, James Rawlings, and Moritz Diehl. «CasADi: a software framework for nonlinear optimization and optimal control». In: *Mathematical Programming Computation* 11 (July 2018). DOI: 10.1007/s12532-018-0139-4 (cit. on pp. 6, 15, 16).

[11] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2015 (cit. on pp. 7, 8).

[12] Arash Bahari Kordabad, Hossein Nejatbakhsh Esfahani, Anastasios M. Lekkas, and Sébastien Gros. «Reinforcement Learning based on Scenario-tree MPC for ASVs». In: *2021 American Control Conference (ACC)*. 2021, pp. 1985–1990. DOI: 10.23919/ACC50511.2021.9483100 (cit. on pp. 11–13).

[13] Sebastien Gros and Mario Zanon. «Data-driven Economic NMPC using Reinforcement Learning». In: *IEEE Transactions on Automatic Control* PP (Apr. 2019), pp. 1–1. DOI: 10.1109/TAC.2019.2913768 (cit. on pp. 12, 13, 26).

[14] Robin Verschueren, Gianluca Frison, Dimitris Kouzoupis, Niels van Duijkeren, Andrea Zanelli, Rien Quirynen, and Moritz Diehl. «Towards a modular software package for embedded optimization». In: *IFAC-PapersOnLine* 51 (Jan. 2018), pp. 374–380. DOI: 10.1016/j.ifacol.2018.11.062 (cit. on pp. 17, 19).

[15] Paolo De Petris, Huan Nguyen, Mihir Kulkarni, Frank Mascarich, and Kostas Alexis. «Resilient Collision-tolerant Navigation in Confined Environments». In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021, pp. 2286–2292. DOI: 10.1109/ICRA48506.2021.9561999 (cit. on pp. 29, 30).

[16] https://docs.px4.io/main/en/. In: () (cit. on p. 30).