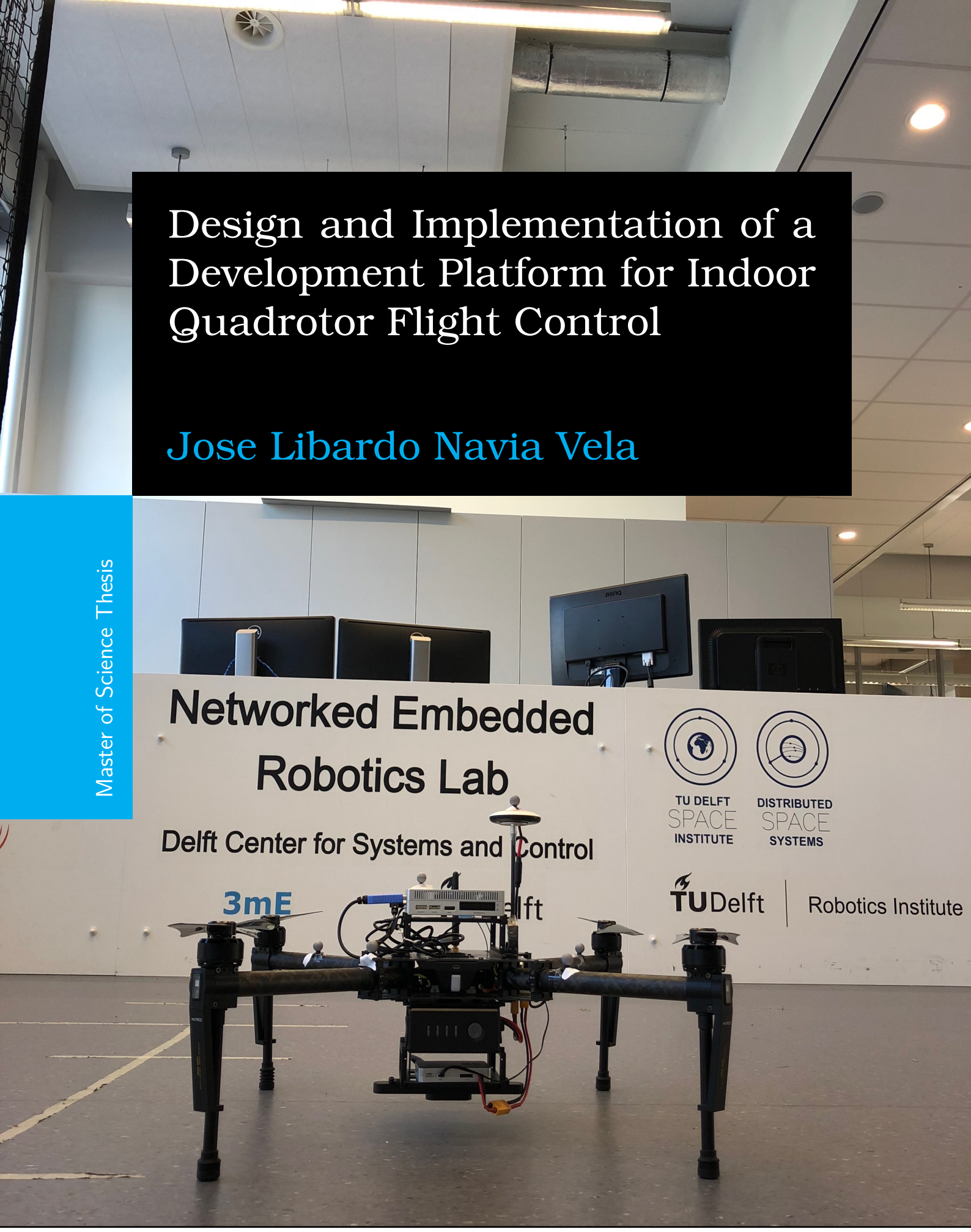
Design and Implementation of a Development Platform for Indoor Quadrotor Flight Control

Jose Libardo Navia Vela

Master of Science Thesis



TU Delft Delft University of Technology

Delft Center for Systems and Control

Design and Implementation of a Development Platform for Indoor Quadrotor Flight Control

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Jose Libardo Navia Vela

September 17, 2018

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Delft University of Technology
Department of
Delft Center for Systems and Control (dcsc)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical, Maritime and Materials Engineering (3mE) for acceptance a thesis entitled

Design and Implementation of a Development Platform for Indoor Quadrotor Flight Control

by

Jose Libardo Navia Vela

in partial fulfillment of the requirements for the degree of

Master of Science Systems and Control

Dated: September 17, 2018

Supervisor(s):

dr.ir. T. Keviczky (chair)

ir. Z Zheng

Reader(s):

dr. S. Baldi

dr. J. Guo

Abstract

Research and development of flight control of quadrotor Unmanned Aerial Vehicles (UAVs) have gained popularity during the past few years due to their deployment flexibility and wide range of applications: agriculture, entertainment, cinematography, package delivery, search and rescue, etc. This thesis project follows up this interest aiming to build a real-time development platform for experimentation and test of indoor flight controllers in the Networked Embedded Robotics in Delft laboratory. For this goal, we employ a Matrice 100 quadrotor from DJI alongside its ROS C++ Software Development Kit (SDK), develop a system identification routine to model its autopilot, vertical and horizontal displacement, and based on the attained results, we design and implement a control topology for real-time position control of the UAV inside the laboratory by means of a Model Predictive Controller (MPC) and a Linear-Quadratic Regulator. Finally, we enunciate and discuss potential applications in which this development platform can be used.

Table of Contents

List of Figures

List of Tables

Acknowledgements

This thesis is the result of a 2-year adventure far from home during which I met many people, visited diverse places for the first time, saw life fade out and reborn twice, lived alone, felt alone but also loved from the distance, understood what *missing someone* means, arrived to the strongest conclusion my mind has ever generated, and learned as never before about myself. However, this journey would not have been possible without the help of many people whom I would like to thank here:

- To my mother and father who have been there to support and guide *every step I take* since I was born.
- To my uncle Crisanto who trusted in my capacities to pay back my study loan and who has followed my progress since the very beginning.
- To my cousins Juan Manuel, Diana, Laura, and Jeisson who were there to read and sometimes discuss all my crazy philosophical analyses, problems, and joyful moments. One day we will be able to travel the world together.
- To all my Colombian and LATITUD friends with whom I shared uncountable unforgettable moments, even before coming to The Netherlands.
- To Dr. Iván Mondragón and Dr. Jairo Hurtado for believing in and giving me their recommendation letters.
- To my supervisor Dr.ir. Tamás Keviczky and co-supervisor ir. Zixuan Zheng for entrusting me this project and their resources.
- To my friend Srinath with whom I gladly worked most of the first year and kept on sharing during the second.
- To all *bloggers*, programmers, and people who respond in forums such as Stack Overflow. Without their indirect help, I would not have been able to resolve multiple implementation issues that popped up during this thesis.
- To all open-source and academia-free code developers, specially to Dr. Jacob Mattingley, for allowing me to use their software free of charge.

- To Colfuturo and, therefore, to all honest Colombians who pay taxes. I could not have come to Delft without your study loan.
- To Robert, Boaz, Nikhil, Iñigo, and Hai who were open to answer my questions in the DCSC Lab.
- To Baptiste for his constant interest in my project and, more importantly, for telling me about CVXGEN, which I used to solve my final implementation challenge.
- To engineer César Marín and SITE S.A.S. in Pereira, Colombia, for employing me in-between the end of my bachelor and the start of my master's degree. Similarly to Mrs. Ana María Piedrahita for welcoming me in her house during that period.

On the other hand and as I did in my bachelor thesis, I would like to thank engineers Juan Carlos Giraldo and Daniel Campos who recognized my skills and potential at early stages of my bachelor. I will never forget your words and will always be thankful for them.

I also want to express my gratitude to all my relatives who have tracked my progress and sent me all their positive energy and love from the distance.

Last but not least, I want to thank my grandmother for all the love she gave me while she was alive and whom I miss every day.

Delft, University of Technology — Jose Libardo Navia Vela
September 17, 2018

"To my grandmother who will always be by my side...'

Let this thesis be a statement to all girls and boys, specially to my little cousins Amelia, Nicolás and Juan Pablo, proving that dreams come true but not by themselves. One needs to work for them, put a lot of effort, and make tough decisions while always respecting everyone. And remember that dreams can only be achieved with the help of others, irrespective of who you are or where you come from, so be thankful with those who have assisted you throughout your life.

Chapter 1

Introduction

A quadrotor is a type of aircraft within the category: *rotary wing*, of the so-called Unmanned Aerial Vehicles (UAVs) [4]. Its most relevant features are: Vertical Takeoff and Landing (VTOL) and hovering, which make them suitable for applications where the available space for takeoff and landing is limited and/or where vertical flight or hovering are important assets, e.g. exploration of uneven terrain or mountains. These two characteristics have made of quadrotors a popular platform in research and development of flight controllers during the past few years [5, 4]. In fact, they are now employed in fields such as agriculture [6], search and rescue [7], entertainment [8] and cinematography [9], for example.

The continuous advancement of embedded systems and the aforementioned popularity of UAVs (specially quadrotors) have encouraged researchers to explore, exploit, and test the capabilities of the named Model Predictive Control (MPC), whose applications used to be limited to the process industry [10], on this type of aircraft. This is how, for instance, Viana et al. [11] proposed a formation control strategy for a group of UAVs to avoid collisions and obstacles, in their strategy each aircraft had a decentralized MPC capable of following a given trajectory while complying with a set of constraints that prevented crashes against each other and different shapes of obstacles. Similarly, Wang et al. [12] utilized an MPC to include the aircraft's constraints and to track the reference computed by a trajectory generator. Hafez et al. [13] explored a multi-UAV dynamic encirclement, where each aircraft had an MPC in charge of generating the drone's trajectory while satisfying the defined encirclement constraints.

However, researchers usually test their control algorithms in simulation models only, disregarding the challenges that may arise when porting their solutions to real setups: computational power, connectivity between the different components of the control scheme, sampling time, model mismatch, safety measures, available sensors and complexity of the resulting controllers, for instance. Still, real-life experiments can be found in the literature, for example, Nägeli et al. [14] controlled a Parrot Bebop 2 via MPC for aerial videography, Potdar [15] also utilized a Parrot Bebop 2 and an MPC for trajectory generation of a UAV plus payload system including obstacle avoidance, Sa et al. [16] developed an MPC for horizontal control of a Matrice 100 from DJI. These works tried to explore the feasibility of MPC in real

quadrotors, at the same time, they serve to illustrate the importance of having a development platform on/with which researchers and engineers can test their flight control algorithms.

1-1 Thesis Objective and Contribution

Considering the current necessity of counting with a reliable and practical setup where to test various control algorithms within the facilities of the Networked Embedded Robotics in Delft laboratory, this MSc. thesis aims to to build a development platform using a (quadrotor) Matrice 100 from DJI to provide future students or interested engineers with the fundamental software, hardware and position controller that enable them to easily (or readily) implement their own control strategies. For this purpose, a typical procedure of modeling, system identification, controller design and testing will be followed.

The to-be-presented development platform differs from previous ones, mainly, in the control formulation. Here we divide the control problem in two: yaw (heading) regulation, i.e. maintaining the yaw angle of the Matrice 100 at 0 rad, by means of an LQR, and horizontal and vertical position control through a linear MPC. In addition, the dynamics of the drone's autopilot are accounted for in the MPC problem, unlike other implementations where they are either completely omitted or used indirectly. The more intuitive drawback of the latter feature is a more complex optimization problem.

On the other hand, the software is implemented in ROS C++, making it suitable for (eventual) onboard processing, contrary to other software usually programmed in MATLAB, thus, not intended for embedded systems.

1-2 Document Outline

The structure of this document is as follows:

- In Chapter 2, the dynamical model of the Matrice 100 is derived,
- Chapter 3 describes the equipment and setup utilized in the practical implementation of this project, including the aerial platform and the laboratory resources,
- In Chapter 4, the system identification routine, tests, and results are given,
- In Chapter 5, the proposed control topology is introduced and the identified system is utilized to design two Kalman observers, one LQR, and one MPC, for which the simulation and experimental results are discussed,
- Finally, Chapter 6 presents the conclusions of this document, the recommendations regarding the current state of the attained platform, and enunciates the future work.

Chapter 2

Quadrotor Modeling

In this chapter, a dynamic model of the Matrice 100 is derived while considering that the aircraft can be controlled via a Software Development Kit (SDK) from its manufacturer (DJI). For this purpose, the required reference systems and attitude angles and defined first, then, the dynamics is obtained by modeling the onboard controllers (embedded in the autopilot of the UAV), followed by the analysis of the drone's displacement in the horizontal plane.

2-1 Reference Systems

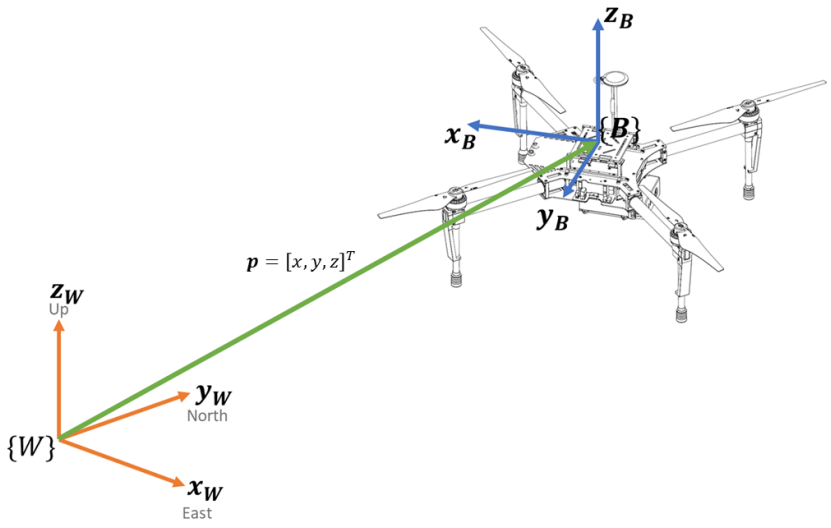


Figure 2-1: World and body coordinate systems, {W} and {B}, respectively. (Adapted from [1]).

Prior to start analyzing the dynamics of the quadrotor, the coordinate systems on which it is expressed need to be defined. Figure 2-1 presents the frames to be employed throughout this

document: i) the so-called East-North-Up (ENU) world reference system, {W}, where $\mathbf{x_W}$ points to the East, $\mathbf{y_W}$ to the North, and $\mathbf{z_W}$ completes the right-hand system; and ii) the body-centered forward-left-up (FLU) frame, {B}, where $\mathbf{x_B}$ points to the forward direction of movement, $\mathbf{y_B}$ to the left, and $\mathbf{z_B}$ completes the right-hand system. The origin of the second frame expressed in {W} is given by $\mathbf{p} = \begin{bmatrix} x & y & z \end{bmatrix}^T$ and corresponds to the absolute position of the drone. Whilst this body frame moves with the aircraft, the world reference system, {W}, is always fixed. These two coordinate systems comply with ROS' *Standard Units of Measure and Coordinate Conventions* [17], on which DJI's ROS-SDK is developed [18].

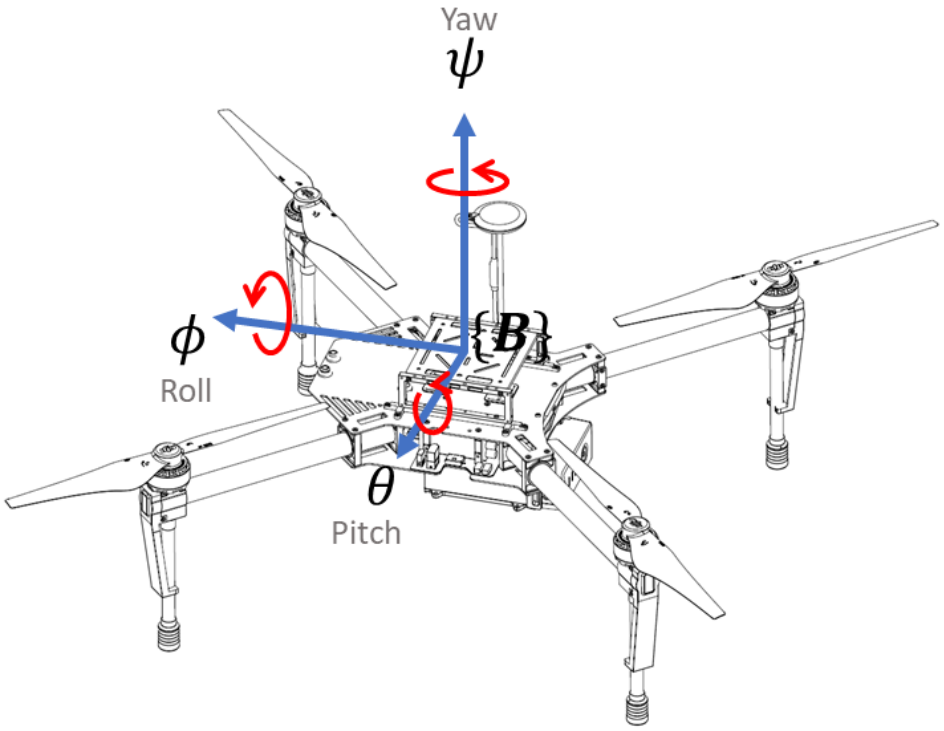


Figure 2-2: Rotation axes of the attitude angles: roll (ϕ), pitch (θ), and yaw (ψ). (Adapted from [1]).

Besides the world and body-centered reference systems, there are also the well-known Euler angles which are used to describe the orientation of the aircraft. Figure 2-2 depicts the rotation axes of these angles, namely: roll (ϕ), pitch (θ), and yaw (ψ); in the picture it can also be seen that these angles are expressed w.r.t. the body frame, {B}. In other words, a non-zero roll angle represents a rotation about $\mathbf{x_B}$, pitch about $\mathbf{y_B}$, and yaw about $\mathbf{z_B}$.

2-2 Dynamic Model

The Matrice 100 from DJI is controlled by an onboard autopilot, which can be accessed through DJI's SDK, which contains a set of functions that allows the user to send commands to and retrieve telemetry data from the drone. By means of the available commands, one can set desired roll and pitch angles, yaw rate, and vertical velocity, each of which utilizes an independent controller.

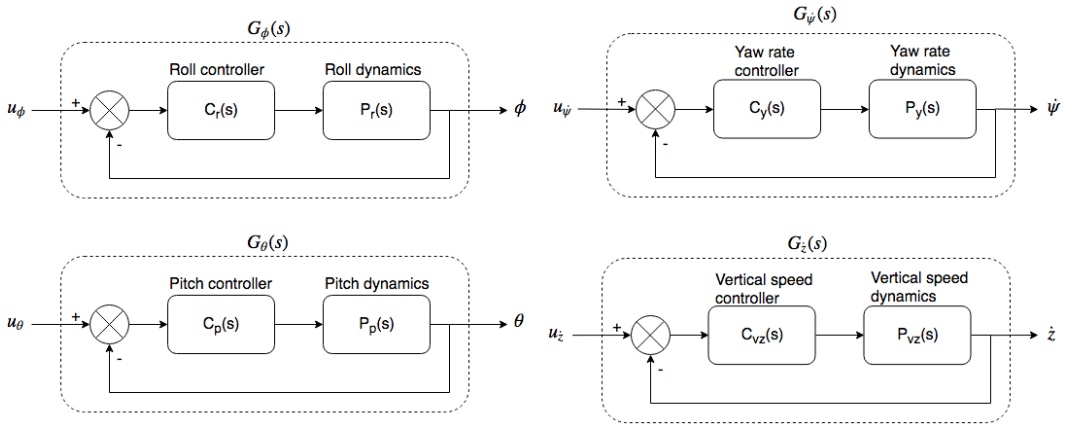


Figure 2-3: Low-level controller representation.

DJI does not provide any information regarding the internal implementation of the Matrice 100's autopilot, hence, the embedded controllers need to be modeled and identified based on input-output (I/O) data. A possible representation of the low-level controllers is shown in Figure 2-3, where u_ϕ, u_θ, $u_{\dot\psi}$, and $u_{\dot z}$ are the desired roll, pitch, yaw rate, and vertical velocity, respectively, and ϕ, θ, $\dot\psi$, and $\dot z$ are the actual measurements. According to [19, 16], we can assume second-order dynamics for the aforementioned pitch and roll controllers, and a first-order response for the vertical velocity and yaw rate, this assumption will be discussed in detail in Chapter 4. Consequently, the following set of transfer functions are derived:

$$\begin{aligned} G_\phi(s) &= \frac{\Phi}{U_\phi} = \frac{b_{0,\phi}}{s^2 + a_{1,\phi}s + a_{0,\phi}}, \\ G_\theta(s) &= \frac{\Theta}{U_\theta} = \frac{b_{0,\theta}}{s^2 + a_{1,\theta}s + a_{0,\theta}}, \\ G_{\dot\psi}(s) &= \frac{s\Psi}{U_{\dot\psi}} = \frac{b_{0,\psi}}{s + a_{0,\psi}}, \\ G_{\dot z}(s) &= \frac{sZ}{U_{\dot z}} = \frac{b_{0,z}}{s + a_{0,z}}, \end{aligned} \tag{2-1}$$

where Ψ, Θ, Ψ, Z, U_ϕ, U_θ, $U_{\dot\psi}$, $U_{\dot z}$ are the Laplace transforms of ϕ, θ, ψ, z, u_ϕ, u_θ, $u_{\dot\psi}$, and $u_{\dot z}$, respectively; $a_{i,j}$ and $b_{0,j}$ for $i = \{0, 1\}$ and $j = \{\phi, \dot\phi, \theta, \dot\theta, \dot\psi, \dot z\}$ are (non-zero) coefficients to be identified, see Chapter 4. Their equivalent time-domain differential equations are:

$$\begin{cases} \ddot\phi = -a_{0,\phi}\phi - a_{1,\phi}\dot\phi + b_{0,\phi}u_\phi \\ \ddot\theta = -a_{0,\theta}\theta - a_{1,\theta}\dot\theta + b_{0,\theta}u_\theta \\ \ddot\psi = -a_{0,\psi}\dot\psi + b_{0,\psi}u_{\dot\psi} \\ \ddot z = -a_{0,z}\dot z + b_{0,z}u_{\dot z} \end{cases}. \tag{2-2}$$

Although these equations can be grouped into one state-space model, for control purposes it is more practical to have two models –this statement is elaborated later on this chapter as well as in Chapter 5. Therefore, let us define $\mathbf{q_1} = \begin{bmatrix} \phi & \dot\phi & \theta & \dot\theta & z & \dot z \end{bmatrix}^T$, $\mathbf{r} = \begin{bmatrix} \psi & \dot\psi \end{bmatrix}$, the

input vector $\mathbf{u} = \begin{bmatrix} u_\phi & u_\theta & u_{\dot{z}} \end{bmatrix}^T$, and assume all states to be measurable –see Chapter 3 for details about the available sensors. Then the subsequent state-space models are obtained:

$$\begin{aligned} \dot{\mathbf{q}}_\mathbf{1} &= A_1 \mathbf{q}_\mathbf{1} + B_1 \mathbf{u}, \\ \mathbf{y}_\mathbf{1} &= C_1 \mathbf{q}_\mathbf{1}, \end{aligned} \tag{2-3}$$

with matrices:

$$\begin{aligned} A_1 &= diag\left(\begin{bmatrix} 0 & 1 \\ -a_{0,\phi} & -a_{1,\phi} \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ -a_{0,\theta} & -a_{1,\theta} \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 0 & -a_{0,z} \end{bmatrix} \right), \\ B_1 &= diag\left(\begin{bmatrix} 0 \\ b_{0,\phi} \end{bmatrix}, \begin{bmatrix} 0 \\ b_{0,\theta} \end{bmatrix}, \begin{bmatrix} 0 \\ b_{0,z} \end{bmatrix} \right), \\ C_1 &= diag\left(1, 1, 1, 1, 1, 1\right). \end{aligned}$$

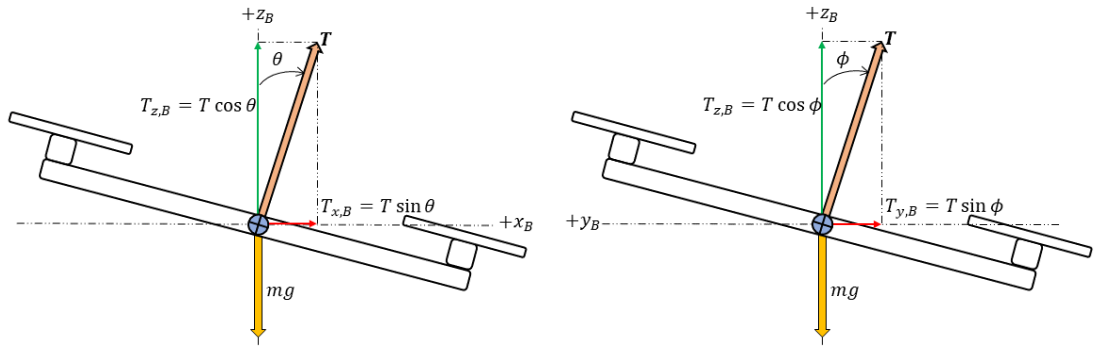
And,

$$\begin{aligned} \dot{\mathbf{r}} &= A_2 \mathbf{r} + B_2 u_{\dot{\psi}}, \\ \mathbf{y}_\mathbf{r} &= \mathbf{r}, \end{aligned} \tag{2-4}$$

with matrices

$$\begin{aligned} A_2 &= \begin{bmatrix} 0 & 1 \\ 0 & -a_{0,\psi} \end{bmatrix}, \\ B_2 &= \begin{bmatrix} 0 \\ b_{0,\psi} \end{bmatrix}, \end{aligned}$$

where $diag(\cdot)$ is a (block) diagonal matrix.



(a) Quadrotor's free-body diagram for non-zero pitch angle, θ.

(b) Quadrotor's free-body diagram for non-zero roll angle, ϕ.

Figure 2-4: Quadrotor's free-body diagrams for non-zero pitch and roll angles, θ and ϕ, respectively.

The individual control of roll and pitch angles, yaw rate, and vertical velocity allows us to analyze the lateral displacement dynamics of the quadrotor separately [5, 20]. In addition, we can assume the drone to be at hovering position, i.e. at the equilibrium point: $\dot{z}_e = 0$, $\theta_e = \phi_e = 0$, $\dot{\psi}_e = 0$, where the vertical forces are in equilibrium [16, 20, 21, 22].

Figure 2-4(a) illustrates the free-body diagram of the aircraft for a non-zero pitch angle referenced to the body-centered coordinate system, {B}, where $\mathbf{T}$ is the thrust vector resulting from the rotation of the propellers, T is its magnitude, $T_{z,B}$ and $T_{x,B}$ are the projection of $\mathbf{T}$ on $\mathbf{x_B}$ and $\mathbf{z_B}$, m is the mass of the UAV, and g is the acceleration of the gravity. From this diagram we have:

$$\begin{cases} T_{z,B} = T\cos\theta \\ T_{x,B} = T\sin\theta \\ T_{z,B} = mg \\ m\ddot{x}_B = T_{x,B} \end{cases}. \tag{2-5}$$

By rearranging these equations we arrive to:

$$\ddot{x}_B = g\tan\theta. \tag{2-6}$$

A similar analysis is performed using Figure 2-4(b), which depicts the free-body diagram for a non-zero roll angle, yielding:

$$\ddot{y}_B = -g\tan\phi. \tag{2-7}$$

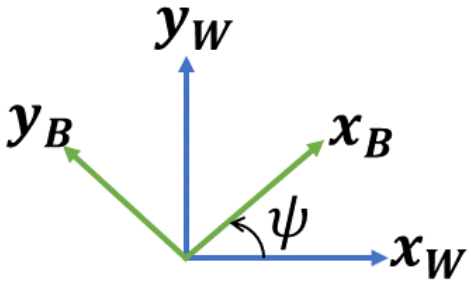


Figure 2-5: 2D-view of the body-centered coordinate system, {B}, and the world frame, {W}, after a yaw rotation, ψ, about the z-axis.

Regardless of the position of the quadrotor, $\mathbf{z_B}$ is always parallel to $\mathbf{z_W}$. Hence, we can transform the dynamics from {B} to {W} by applying the rotation matrix $R(\psi)$ [14] given in Eq. (2-8) [23], which represents a rotation about $\mathbf{z_W}$ (also $\mathbf{z_B}$) by a yaw angle ψ, see Figure 2-5.

$$R(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix}. \tag{2-8}$$

One more effect to be considered is the aerodynamic drag, which according to [24] can be modeled as a proportional linear force on the quadrotor given that it will be operated at

relatively low speeds. Combining this effect and the aforementioned dynamics in the world frame, {W}, yields:

$$\begin{aligned}\begin{bmatrix}\ddot{x}\\ \ddot{y}\end{bmatrix} &= R(\psi)\begin{bmatrix}\tan\theta\\ -\tan\phi\end{bmatrix}g + k_d\begin{bmatrix}\dot{x}\\ \dot{y}\end{bmatrix}\\ &= \begin{bmatrix}\cos\psi\tan\theta + \sin\psi\tan\phi\\ \sin\psi\tan\theta - \cos\psi\tan\phi\end{bmatrix}g + k_d\begin{bmatrix}\dot{x}\\ \dot{y}\end{bmatrix}.\end{aligned} \tag{2-9}$$

where k_d is the drag coefficient, whose value will be identified in Chapter 4.

Eq. (2-9) has infinite equilibrium points, to linearize it, a value for ψ has to be chosen and it needs to be constant while the quadrotor is moving in the horizontal plane –the control challenge due to this requirement is discussed in Chapter 5. Besides, the pitch and roll angles (θ and ϕ) have to be small, which in reality is the case because the aircraft is going to be operated at low speeds. By complying with these two conditions and naming ψ_0 the value of ψ during the horizontal displacement, the following linear equations are attained:

$$\begin{cases}\ddot{x} = (\cos\psi_0\theta + \sin\psi_0\phi)\,g + k_d\dot{x}\\ \ddot{y} = (\sin\psi_0\theta - \cos\psi_0\phi)\,g + k_d\dot{y}\end{cases}. \tag{2-10}$$

By introducing a new state vector $\mathbf{q} = \begin{bmatrix}\mathbf{q_1}^T & x & \dot{x} & y & \dot{y}\end{bmatrix}$, taking into account Eq. (2-3), and recalling that $\mathbf{u} = \begin{bmatrix}u_\phi & u_\theta & u_{\dot{z}}\end{bmatrix}^T$, a new state-space model is constructed:

$$\begin{aligned}\dot{\mathbf{q}} &= A_q\mathbf{q} + B_q\mathbf{u},\\ \mathbf{y_q} &= C\mathbf{q},\end{aligned} \tag{2-11}$$

whose matrices are:

$$A_q = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
-a_{0,\phi} & -a_{1,\phi} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & -a_{0,\theta} & -a_{1,\theta} & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & -a_{0,z} & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\
g\sin\psi_0 & 0 & g\cos\psi_0 & 0 & 0 & 0 & 0 & k_d & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1\\
-g\cos\psi_0 & 0 & g\sin\psi_0 & 0 & 0 & 0 & 0 & 0 & 0 & k_d
\end{bmatrix},$$

$$B_q = \begin{bmatrix} 0 & 0 & 0 \\ b_{0,\phi} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & b_{0,\theta} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & b_{0,z} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

$$\begin{aligned} C &= diag\left(C_1, \begin{bmatrix} 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \end{bmatrix}\right) \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \end{aligned}$$

The upcoming chapter presents the setup and equipment to be used in this thesis, including the Matrice 100 whose dynamic model was derived here.

Chapter 3

Equipment and Setup Description

This chapter describes the equipment and setup utilized in this thesis project in three sections: i) the aerial platform, i.e. the aircraft and the onboard accessories; ii) the ground station, which consists of two desktop PCs, an emergency system, a remote control (RC), a Wi-Fi router, and a motion capture system (OptiTrack); and iii) the block diagram of the connections between each of these resources.

3-1 Aerial Platform

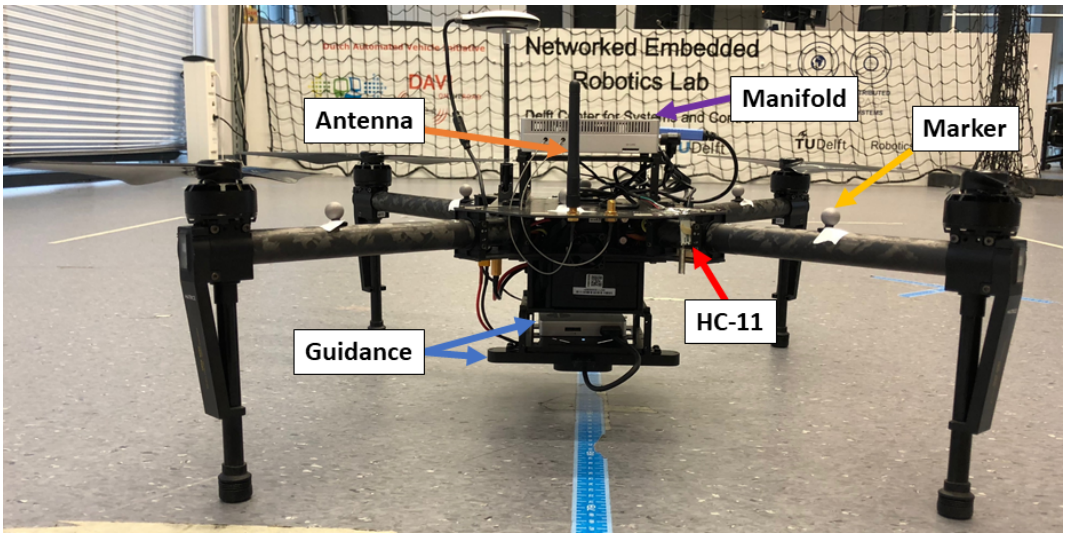
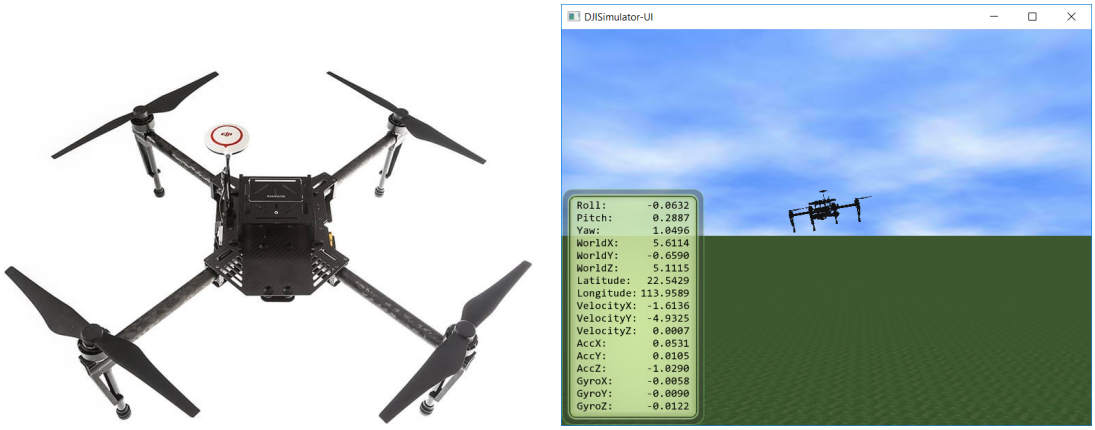


Figure 3-1: Matrice 100 with Manifold, HC-11 wireless serial port module, dual-band antenna, OptiTrack markers, and Guidance.

Figure 3-1 displays the aerial platform of this project: *Matrice 100* from DJI. Onboard, there are: i) an embedded computer called *Manifold* from DJI; ii) a visual sensing system

named *Guidance* also from DJI; iii) a radio frequency (RF) wireless serial module, HC-11; iv) OptiTrack markers; and v) a 2 dB dual-band antenna –all the electronics are powered by a TB47D battery from DJI [25]. These devices are described next.

3-1-1 Matrice 100



(a) Matrice 100 [26]. (b) DJI simulator.

Figure 3-2: Matrice 100 and DJI simulator.

The aircraft (alone) is presented in Figure 3-2(a). The relevant specifications of this quadrotor are [18]:

Feature	Value
Max. yaw rate	$5\pi/6$ rad/s
vertical velocity limits	-5 to 5 m/s
Max. roll and pitch angles	0.611 rad

Table 3-1: Specifications Matrice 100.

Among the onboard sensors, those we use are: i) the inertial measurement unit (IMU), which provides measurements of the attitude angles (ϕ, θ, and ψ), and their rates ($\dot{\phi}$, $\dot{\theta}$, and $\dot{\psi}$) at 100 Hz; and ii) the barometer for altitude (z) and vertical velocity ($\dot{z}$) estimation (when there is no GPS signal nor Guidance available) at 50 Hz. It is important to mention that the yaw angle (ψ) is measured with respect to the North Magnetic Pole as internally indicated by the compass calibration.

As it was said in Chapter 2, DJI offers an open-source Software Development Kit (SDK) (in C++ and ROS C++), which can be installed in an external Linux computer. By doing so and connecting it via UART to the drone's autopilot, the user can program his/her own routines to access the onboard sensor data and/or to send commands (e.g. desired attitude angles, vertical velocity, or yaw rate) to the quadrotor.

DJI also provides a hardware-in-the-loop simulator within its *DJI Assistant 2*, which reads and imports the configuration of the autopilot. A screenshot of the interface is shown in Figure 3-2(b); within the measurements therein, those of our interest are: roll, pitch, and

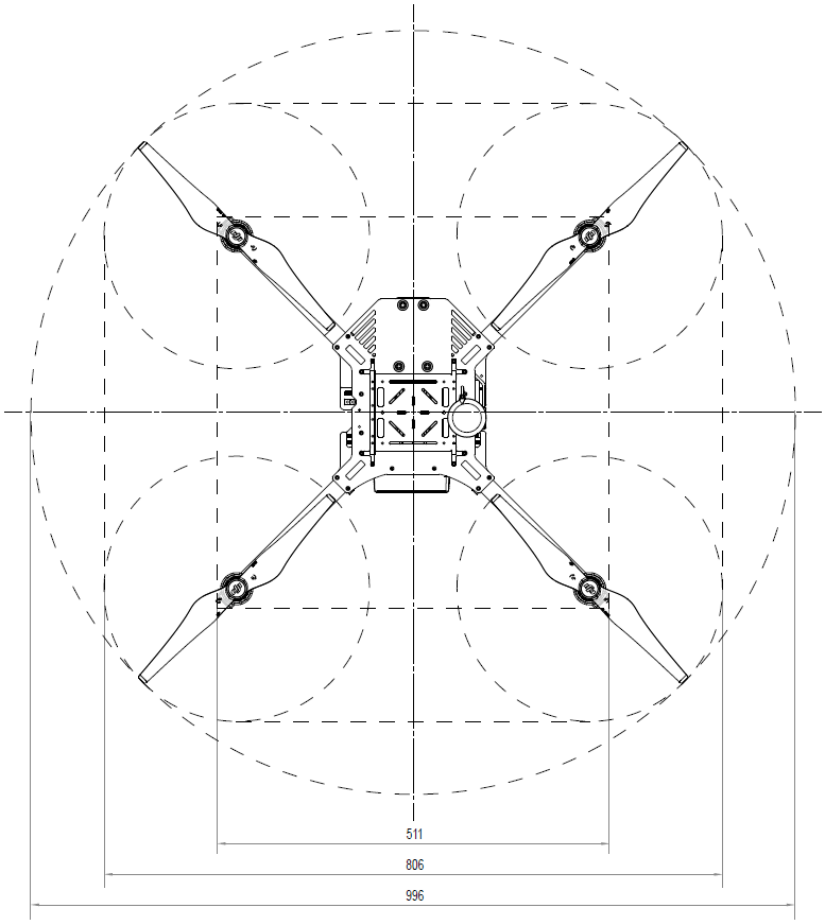


Figure 3-3: Dimension of the Matrice 100 (in millimeters) [1].

yaw angles, velocity in the z-axis, and angular rates (GyroX, GyroY, and GyroZ). Although initial tests can be executed in this simulator, different real-life results are expected because the actual platform has a few extra onboard accessories which are not part of the simulation model, see Figure 3-1, and the physical conditions differ from those in this software.

Figure 3-3 shows the top view of the Matrice 100 with the propellers attached. According to the measurements therein, the diameter of the UAV is 996 mm (0.996 m), which is comparable to the available workspace, see Section 3-2-1; thence, special attention must be paid when running experiments to avoid collisions.

3-1-2 Manifold

An onboard embedded computer, *Manifold*, with Ubuntu, ROS, and DJI's (Onboard and ROS) SDK installed, is connected via UART to the autopilot of the quadrotor; its connector overview is displayed in Figure 3-4. A USB-to-TTL converter is hooked up to one of the USB ports to connect a wireless serial module, HC-11; an Intel® Dual Band Wireless-AC 7260 (model 7260HWM) is installed on the mini PCIe Slot for Wi-Fi communication; a 2 dB dual-band antenna is connected to the wireless adapter for signal reception and transmission.

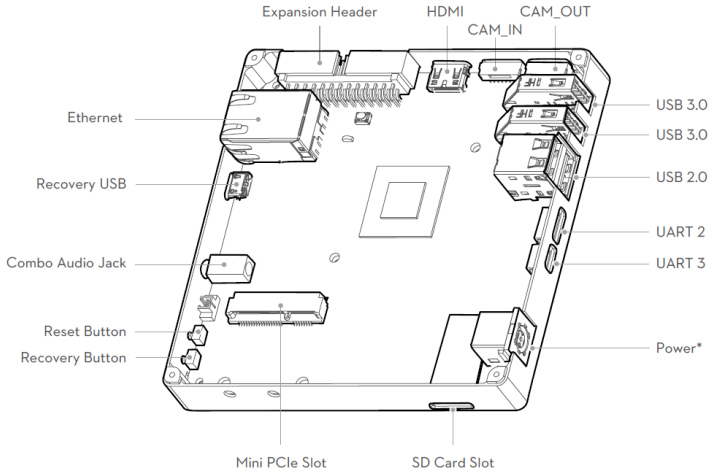


Figure 3-4: Manifold's connector overview [2].

Other specifications of this computer are listed in Table 3-2.

Feature	Value
Processor	Quad-core, 4-Plus-1™ARM®
Graphics processor	Low-power NVIDIA Kepler™-based GeForce®
RAM	2 GB DDR3L
Storage	16 GB eMMC 4.51

Table 3-2: Specifications DJI Manifold.

3-1-3 Guidance

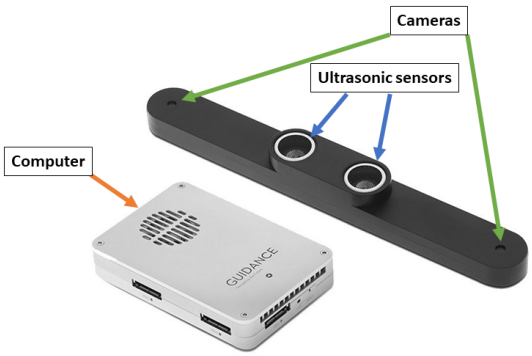


Figure 3-5: Guidance [3].

An onboard visual system, *Guidance*, is connected to the quadrotor's autopilot via its CAN-Bus. As presented in Figure 3-5, this system consists of a central computer that processes the signals from a measurement unit, which comprises a stereo camera pair and two ultrasonic sensors. The entire system has five (5) units, but for the purposes of this project, only one (pointing downwards) is used now that it is sufficient to generate the vertical velocity estimate

for the autopilot, which replaces that of the barometer [1]. The signal processing required to compute this estimate is already implemented by DJI and is sent directly to the autopilot, making it a plug and play accessory. However, the cameras have to be calibrated before using them.

3-1-4 Wireless Serial Module

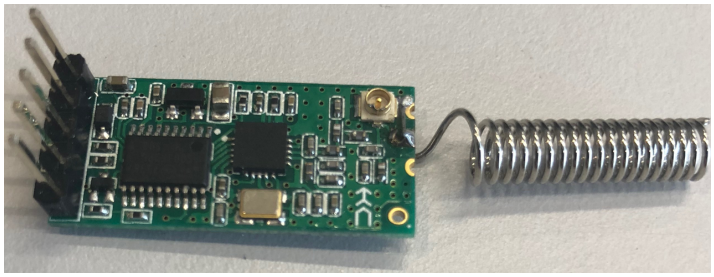


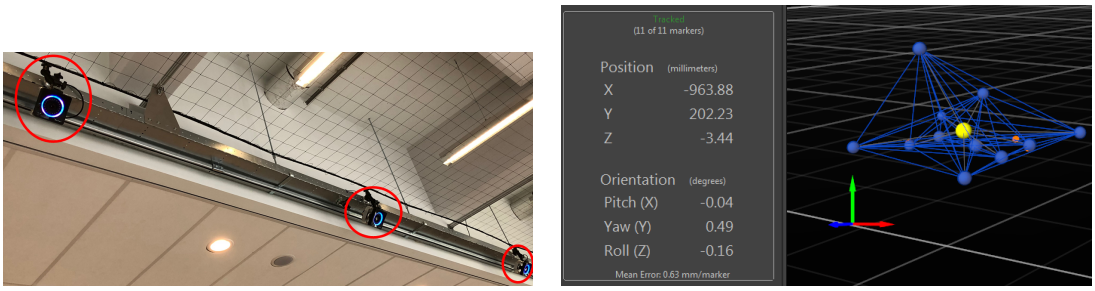
Figure 3-6: HC-11.

The RF wireless serial module HC-11, shown in Figure 3-6, allows the user to send data from one of these devices to another using the 434 MHz band. In this project, a pair of HC-11 is employed to send an emergency command from the ground station to the drone's onboard computer. The settings of both modules are: 4800 baud, 8-bit data, no check, one stop bit.

3-2 Ground Station

As mentioned at the beginning of this chapter, the ground station comprises: two desktop PCs, an emergency system, a remote control (RC), a Wi-Fi router, and a motion capture system (OptiTrack). The functional description of each component is given next.

3-2-1 OptiTrack and Motive



(a) OptiTrack cameras (encircled). (b) Position measurement in Motive.

Figure 3-7: OptiTrack and Motive

The motion capture system consists of ten (10) OptiTrack cameras, see Figure 3-7(a), that track reflective markers, such as those in Figure 3-1, within a workspace of 6.0 × 3.0 × 2.6 m

(length × width × height). These cameras are connected to a Windows (desktop) computer where a software called *Motive* processes the data and, according to its calibration, determines the absolute position of the aircraft (x, y, and z), see Figure 3-7(b). In addition, this software also computes the orientation of the drone in the world frame, i.e. ϕ_W, θ_W, and ψ. These angles are set to zero (0) at the moment the rigid body is created in Motive and can be reset at any time by the user. Thus it is better to use OpiTrack's yaw angle than the IMU's (from the drone). Although the readings in Motive do not comply with ROS' coordinate standard, the ROS package *mocap_otitrack* applies the required rotation matrix to express them in ENU coordinates. Motive also broadcasts the resulting data onto the local network.

3-2-2 Emergency System

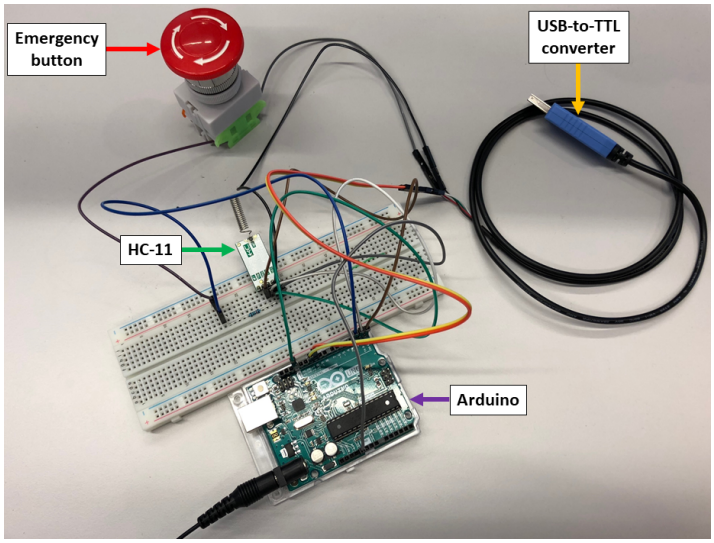


Figure 3-8: Emergency system: Emergency button, Arduino, HC-11, and USB-to-TTL converter.

Figure 3-8 shows the different elements of the emergency system: A push-button, an Arduino, an HC-11, and a USB-to-TTL converter. These devices operate as follows: one terminal of the button is connected to the 5 V end of the converter and the other to one interrupt pin of the Arduino; while the button is released, the voltage in the interrupt pin is logical LOW, but when it is pressed, it is HIGH. The Arduino is programmed to trigger a function that sends an (ASCII) A to the HC-11 and the USB-to-TTL converter whenever a rising edge is detected at that pin.

3-2-3 On-ground PC, RC, and Router

The second desktop computer runs Ubuntu Mate, the ROS master and a pair of ROS packages: the main algorithm (for control or identification) and *mocap_optitrack*. The latter reads the broadcasted OptiTrack data, rotates the coordinate system (to comply with ROS standards) and publishes the result into a ROS topic. The main algorithm subscribes to that topic and utilizes the measurements. The specifications of this desktop computer are listed in Table 3-3.

Feature	Value
Processor	Intel® Xeon® E5603 1.60 GHz
RAM	4 GB DDR3
Storage	35 GB

Table 3-3: Specifications on-ground PC.

The RC, on the other hand, is employed whenever manual control of the Matrice 100 is needed. Besides, it has to be ON during the entire flight because it is required by the autopilot.

Finally, the router manages the local network to which all three computers (Manifold and desktop PCs) are connected. While the ground station's computers are hooked up via Ethernet, the Manifold uses a 5.0 GHz Wi-Fi connection.

3-3 Block Diagram of the Complete Setup

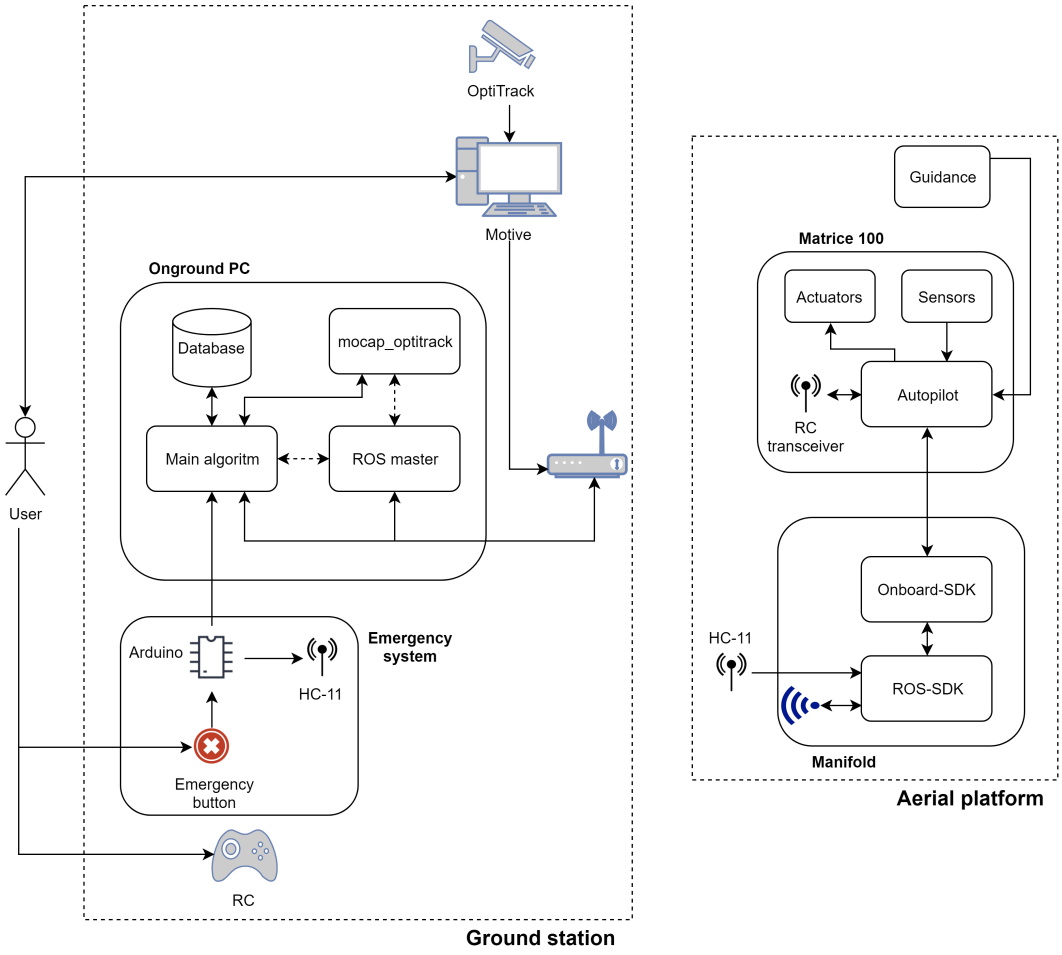


Figure 3-9: Block diagram of the entire setup: ground station to the left and aerial platform to the right.

The entire interconnection between the different elements of this thesis' setup is presented in Figure 3-9, where it can be seen that the user is responsible for operating the ground station, i.e. he/she can push the emergency button, set up Motive, and utilize the on-ground PC. In addition, he/she receives feedback from the main algorithm as on-screen messages, and monitors the current status of the OptiTrack cameras and tracked objects on Motive.

On the other hand, the USB-to-TTL converter of the emergency system is connected to one of the USB ports of the on-ground PC where the transmitted data is read by the main algorithm which internally activates the required procedure to stop sending commands to the Matrice 100 when an emergency action is triggered.

The main algorithm can also import an input signal for identification or previously computed matrices and vectors contained in text files stored in the database. Furthermore, it generates the required log files for post-processing analyzes. This algorithm interacts with the ROS master –which informs all ROS nodes about the existence of the others when they are started–, mocap_optitrack and ROS-SDK (indirectly through the router). Although our implementation runs the main algorithm in the on-ground PC, one may decide to port it to the Manifold and execute it there. However, the emergency action procedure would have to be revised (or bypassed which is highly discouraged) because there would not be a direct connection between the emergency system and the algorithm.

As mentioned in the previous section, the (two-way) communication between the on-ground PC and Manifold is achieved via Wi-Fi. The Manifold sends the available sensor measurements to the main algorithm through the ROS-SDK, which at the same time reads the messages sent from the emergency system and activates the emergency action if needed. ROS-SDK uses DJI's Onboard SDK to access the data from and to send commands to the autopilot.

Finally, the autopilot controls the actuators (propellers' rotors), processes the onboard sensor measurements (from the IMU and Guidance), and executes the commands received from the RC and/or the Onboard SDK.

The setup described here is employed in the subsequent chapters for system identification and control.

Chapter 4

System Identification

This chapter presents and explains the algorithms used for system identification of the low-level controllers of the Matrice 100's autopilot and the drag coefficient (k_d), introduced in Chapter 2. Besides, the attained results are also given and discussed.

Due to the possibility of unexpected behavior while testing and to prevent accidents, the emergency system presented in Section 3-2-2 is essential when running an experiment. Thus, we start by describing how the emergency actions are executed in both: the ground station and the onboard computer (Manifold). In order to do so, the onboard procedure is treated first and the on-ground action is explicated next in combination with the rest of the algorithm used during the experiments.

4-1 Onboard Emergency Action

The flow chart of the onboard emergency action is depicted in Figure 4-1 –this process is added to the (original) ROS-SDK, see Figure 3-9. The first step is to start the dij_sdk node. Afterwards, the USB port where the HC-11 is connected to is opened and configured. Then, the algorithm enters a loop where, every iteration, the USB data is read and compared with an (ASCII) *A*, which corresponds to the emergency triggering command. The loop can be stopped at any time by pressing *Ctrl+C*. If the emergency action is triggered, the control of the drone is requested and the landing command is sent –due to a bug in the SDK, this instruction may be given twice. Thenceforth, the algorithm waits until the aircraft is on ground. At that moment, it releases the control of the drone, closes the USB port and ends the execution by shutting down the node.

4-2 Algorithm for Input-Output (I/O) Data Collection

The algorithm utilized to obtain the necessary data for system identification is displayed in the form of a flow chart in Figure 4-2. The procedure is as follows: first, the (identification)

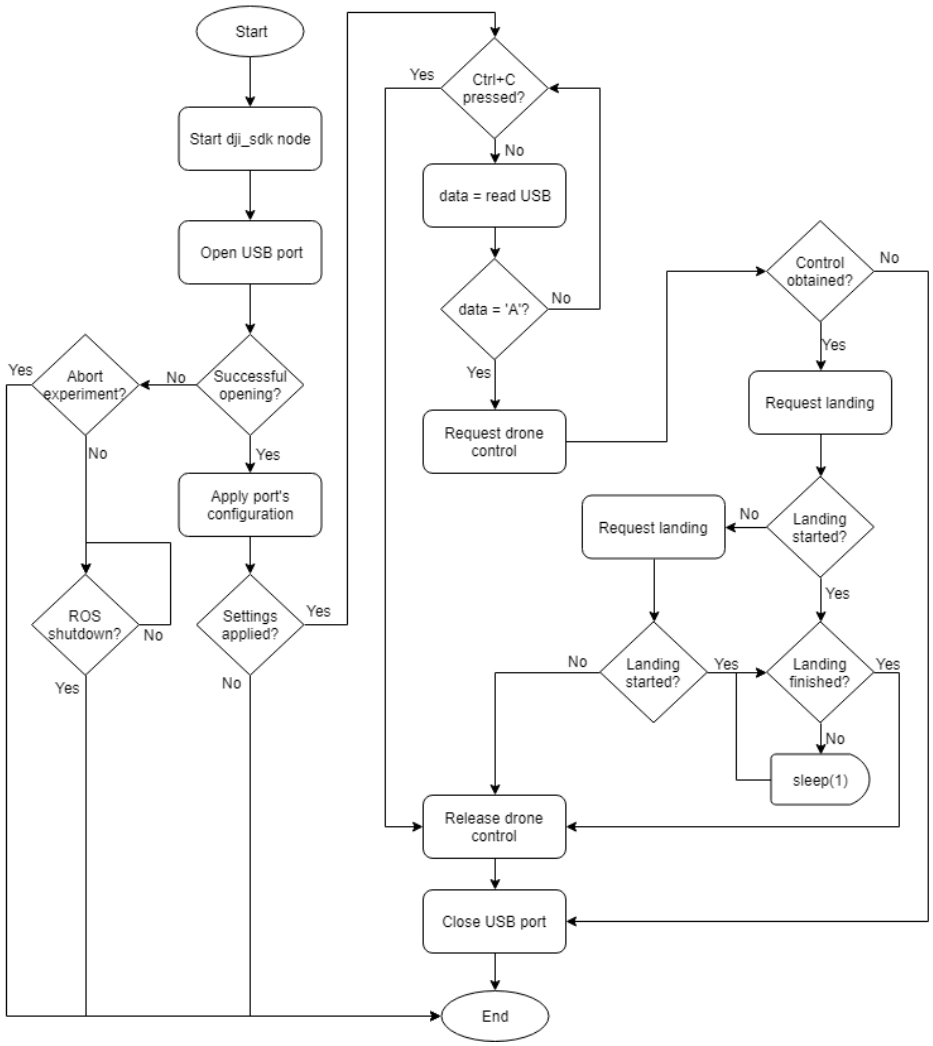


Figure 4-1: Onboard emergency action.

input signals (one per low-level controller) are imported. Then, the number of samples (N), the number of inputs (m), and the sampling time (h) of the signals are retrieved from the imported data. Next, the log files are created (to store the measurement data: roll, pitch, yaw, yaw rate, position, and vertical velocity). Afterwards, the USB port, where the USB-to-TTL converter from the emergency system is connected, is opened to *listen* to the emergency call. Subsequently, the (ROS) node is initiated and the subscribers and publishers to interact with the drone (via ROS-SDK) and mocap_optitrack are instantiated. Following that, the control of the drone is requested and, if obtained, the algorithm waits for the user to pilot the takeoff. Once the user considers the drone is at a position that minimizes the chances of crashing, he/she can command the start of the experiment. Now that the quadrotor may be slightly tilted at that point, the attitude angles are reset. Afterwards, the time, sample instant (k), loop rate, and the previous time stamp (prev_stamp) –that indicates at what moment the last measurement was taken–, are initialized. The algorithm then enters a loop which is in charge of: i) reading the USB to determine if the emergency action has to be activated; ii) getting the current time stamp (current_stamp) and updating the time variable if needed;

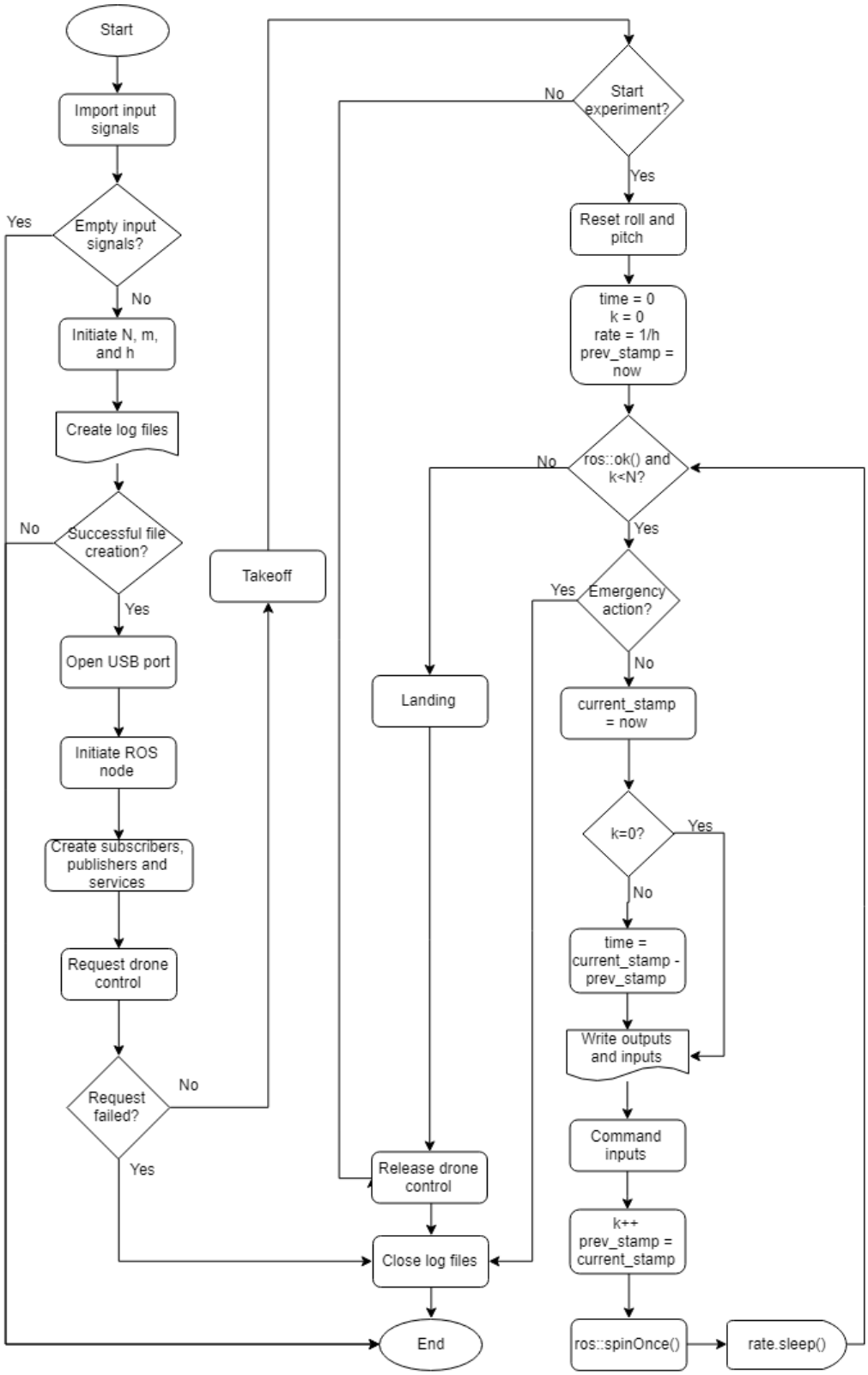


Figure 4-2: Flow chart of the algorithm for I/O data acquisition.

iii) logging the current outputs (measurements) and inputs; iv) commanding the inputs to the autopilot and updating k and the previous stamp; v) calling the function *ros::spinOnce()* to update all measurement data; and vi) sleeping for as long as it is needed to comply with the desired rate of the loop. When the entire input signals, i.e. the N samples, have been commanded or the user stops the loop, the drone lands and the log files are closed, finishing the experiment.

4-3 Identification of the Low-level Controllers

Once the input-output (I/O) data has been obtained (using the algorithm explained above), a grey-box identification approach is followed. This method consists on computing the value of the parameters of a pre-derived mathematical model in order to fit its response to that of the actual system [27]. The goal of this section is to identify the value of the coefficients in Eq. (2-1), which as explained in Chapter 2 models the I/O behavior of the low-level controllers. This equation is repeated here for readability:

$$
\begin{aligned}
G_{\phi}(s) &= \frac{b_{0,\phi}}{s^2 + a_{1,\phi}s + a_{0,\phi}}, \\
G_{\theta}(s) &= \frac{b_{0,\theta}}{s^2 + a_{1,\theta}s + a_{0,\theta}}, \\
G_{\dot{\psi}}(s) &= \frac{b_{0,\psi}}{s + a_{0,\psi}}, \\
G_{\dot{z}}(s) &= \frac{b_{0,z}}{s + a_{0,z}}.
\end{aligned}
$$

To attain such a goal, the acquired I/O data are imported into MATLAB, and then into Simulink's *Parameter Estimation* toolbox –a tutorial can be found in [28]. This toolbox computes the value of the parameters by solving the following (curve fitting) optimization problem [29]:

$$
\min_{\mathbf{p_{id}}} \quad \|F(\mathbf{p_{id}}, u_{id}) - y_{id}\|_2^2, \tag{4-1}
$$

where $\mathbf{p_{id}}$ are the parameters to identify (grouped into a vector), u_{id} contains the input signal, y_{id} the sensor measurements (response of the system), and $F()$ is a function internally built by the toolbox from a Simulink block model that represents the dynamics of the to-be-identified system. Figure 4-3 depicts the diagram utilized in this case which is simply a transfer function block whose numerator and denominator change depending on the controller of interest. In other words, each transfer function is estimated individually.

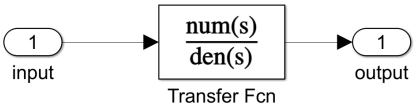
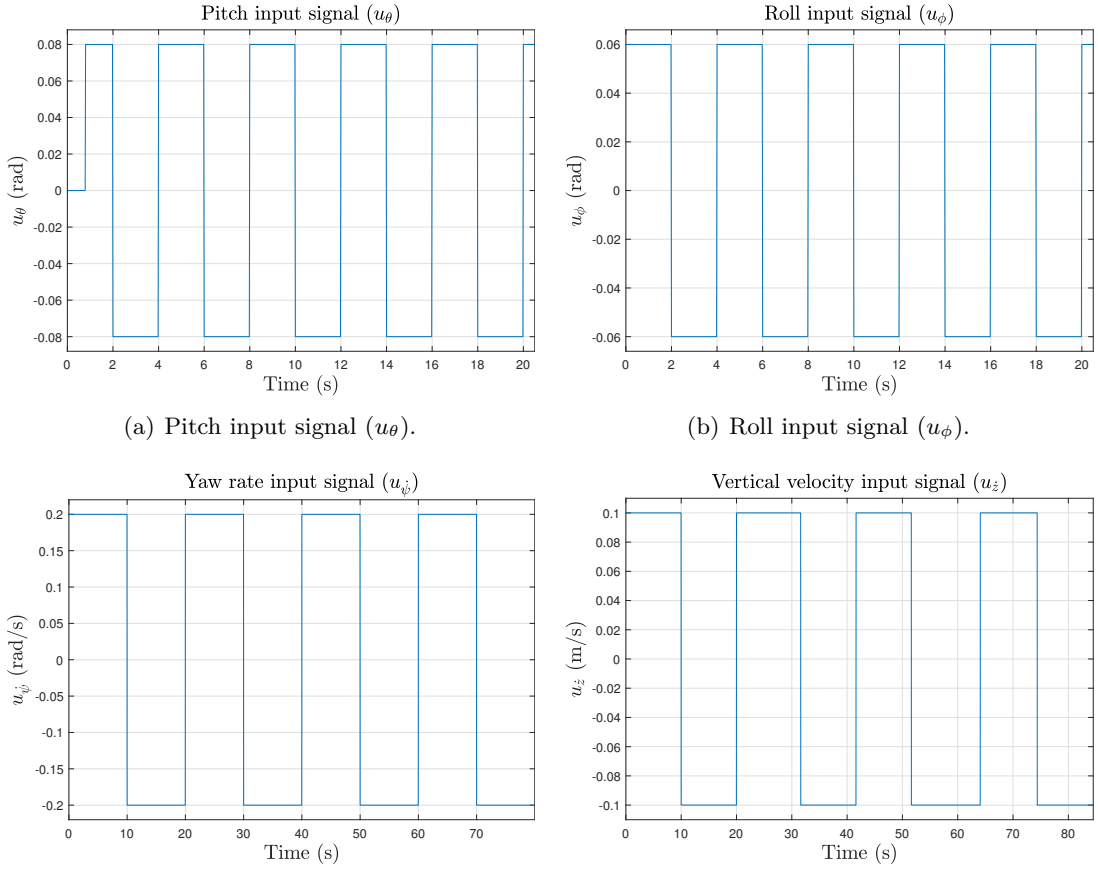


Figure 4-3: Transfer function in Simulink.



(a) Pitch input signal (u_θ).

(b) Roll input signal (u_ϕ).

(c) Yaw rate input signal ($u_{\dot{\psi}}$).

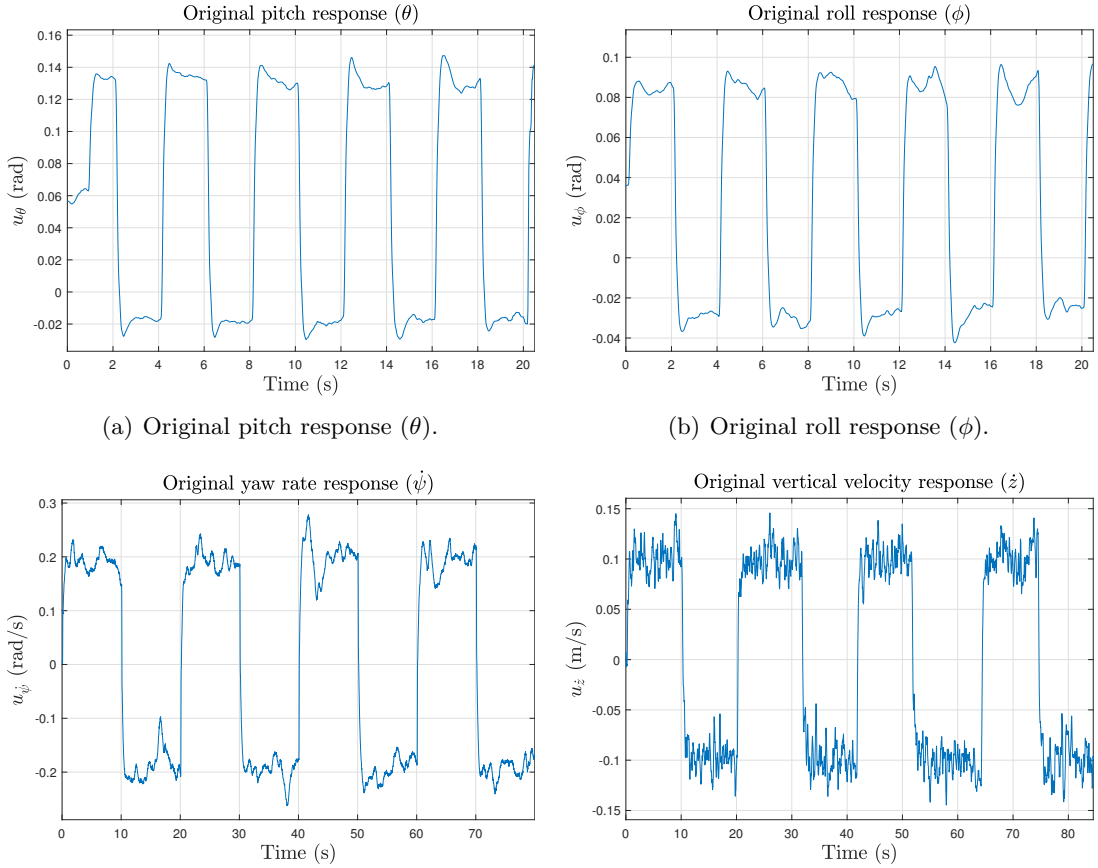
(d) Vertical velocity input signal ($u_{\dot{z}}$).

Figure 4-4: Identification input signals.

Square signals are used as inputs for identification because we are working with either first or second-order systems, which can be fully characterized with such type of inputs. In addition, their sampling time, h, is 0.02 s, which corresponds to a sampling frequency of 50 Hz (frequency at which the vertical velocity estimate is broadcasted by the quadrotor's autopilot, see Chapter 3). Figure 4-4 presents the employed input signals, notice that their amplitude and frequency are such that when applied to the quadrotor, it does not crash into the windows, net, or ceiling around the workspace, while still capturing the dynamic response of the system.

4-3-1 Results

The corresponding responses of the low-level controllers are presented in Figure 4-5. Figure 4-5(a) and Figure 4-5(b) exhibit a second-order-like response, confirming the previously made assumption regarding the order of the pitch and roll controllers, and in both cases there is a trend in the signals, which may correspond to the operation point of these controllers. On the other hand, the yaw rate and vertical velocity measurements in Figure 4-5(c) and Figure 4-5(d), respectively, are noisy, yet, a first-order like response can be distinguished.



(a) Original pitch response (θ).

(b) Original roll response (ϕ).

(c) Origianl yaw rate response ($\dot{\psi}$).

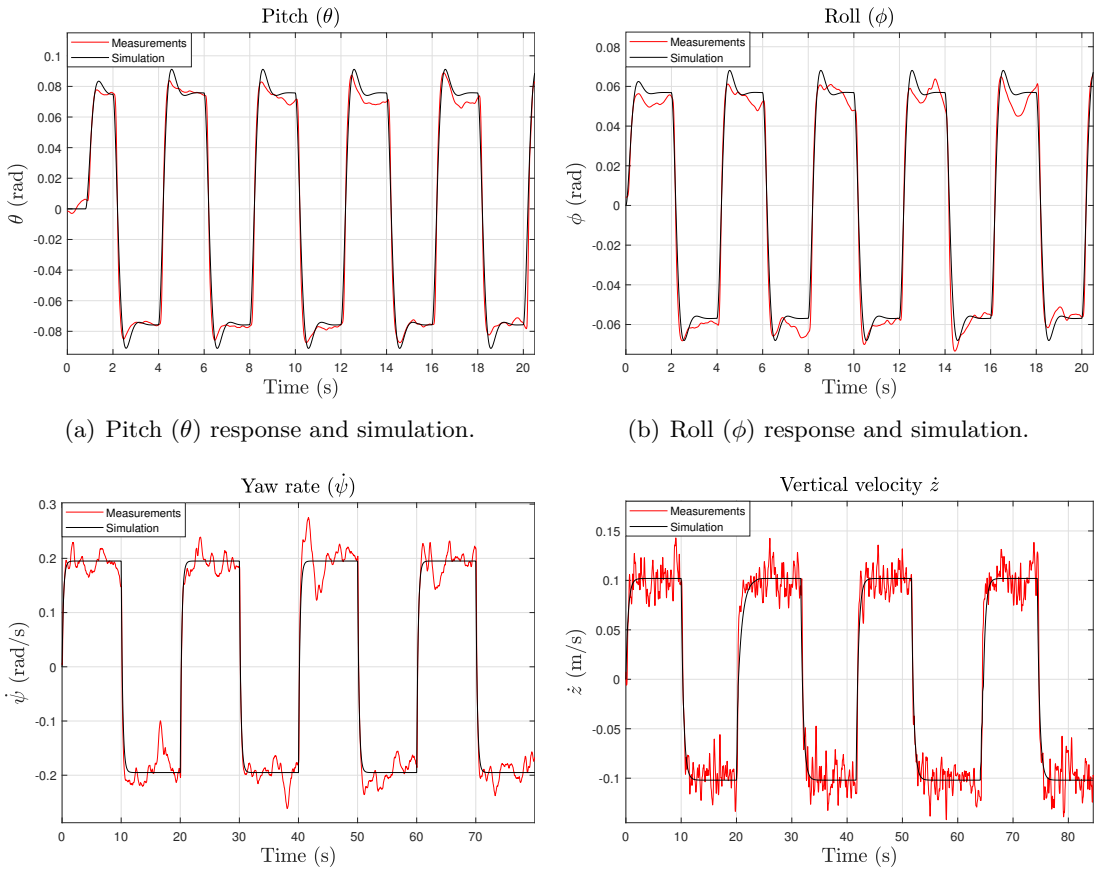
(d) Original vertical velocity response ($\dot{z}$).

Figure 4-5: System identification: Original measurements.

Before estimating the parameters of each controller, it is important to filter the measurement data and detrend the pitch and roll signals [30]. By doing so, we can expect to obtain accurate results and to actually identify linear models. The selected filter is a moving average with span 5 –applied using the function *smooth()* in MATLAB [31]–, which is chosen because it is optimal for reducing random noise, making it ideal for time domain signals [32].

The processed measurements and simulations of the identified models are displayed in Figure 4-6, where it can be observed that in each picture both signals are alike. In order to have a metric to determine whether the obtained results are reliable, the so-called Variance accounted for (VAF), see Eq. (4-2), is computed for each controller, this value represents the degree of closeness of the simulation to the experimental data [30]. The results are summarized in Table 4-1 and lead us to the conclusion that the identification is successful now that all VAFs are above 92%. The attained parameters are listed in Table 4-2

$$\text{VAF}(y_{id}(k),\, \hat{y}_{id}(k)) = \max\left(0, \left(1 - \frac{\frac{1}{N}\sum_{k=1}^{N} \|y_{id}(k) - \hat{y}_{id}(k)\|_2^2}{\frac{1}{N}\sum_{k=1}^{N} \|y_{id}(k)\|_2^2}\right) \cdot 100\%\right). \tag{4-2}$$



(a) Pitch (θ) response and simulation.

(b) Roll (ϕ) response and simulation.

(c) Yaw rate ($\dot{\psi}$) response and simulation.

(d) vertical velocity ($\dot{z}$) response and simulation.

Figure 4-6: System identification: simulation vs detrended and/or filtered measurements.

In Eq. (4-2), y_{id} is the measured data, $\hat{y}_{id}$ is the outcome of the simulation, and N is the total number of samples.

Controller	VAF (%)
Roll (ϕ)	98.5
Pitch (θ)	98.9
Vertical velocity ($\dot{z}$)	92.4
Yaw rate ($\dot{\psi}$)	98.4

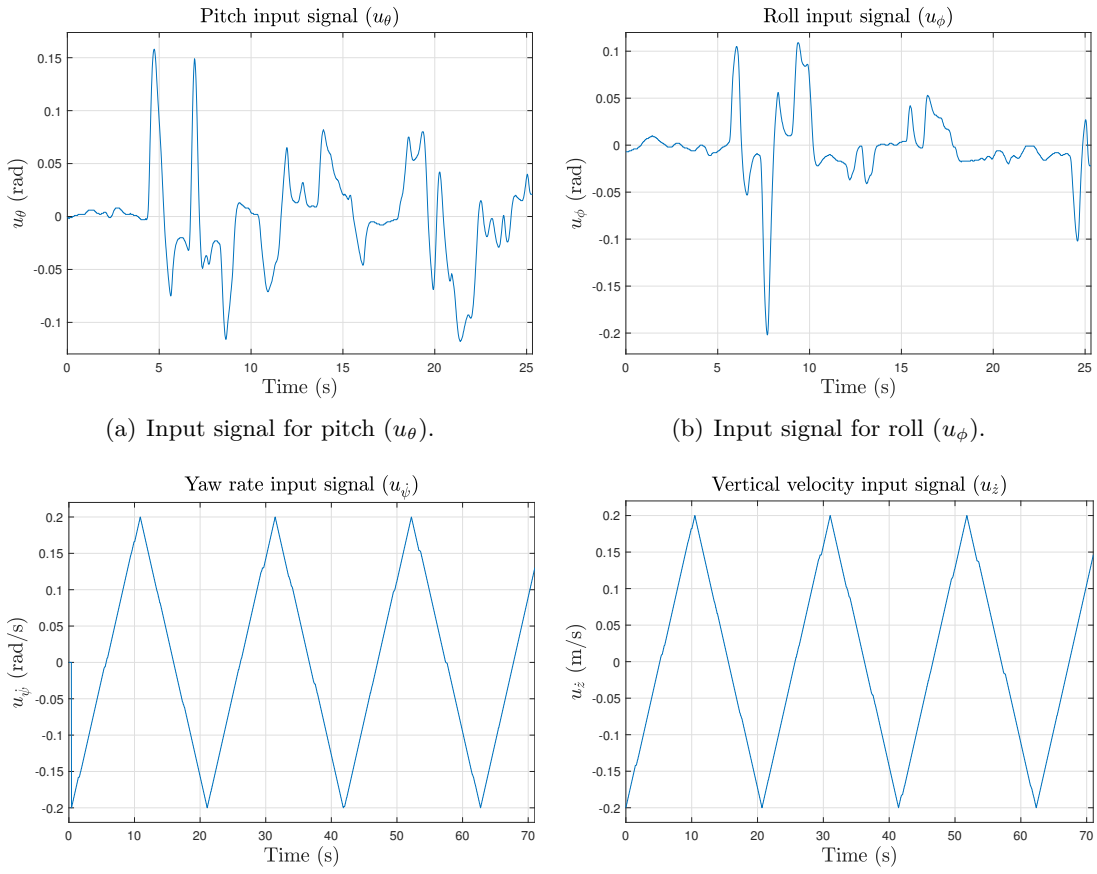
Table 4-1: System identification VAF.

4-3-2 Validation

New experiments are executed to validate the attained parameters. The input signals are displayed in Figure 4-7. In this opportunity, a sawtooth signal is chosen as input for the yaw rate and vertical velocity controllers ($u_{\dot{\psi}}$ and $u_{\dot{z}}$, respectively), moreover, these two are applied at the same time in order to observe if there is coupling between both controllers. Similarly,

Parameter	Value
$b_{0,\phi}$	47.3101
$a_{1,\phi}$	8.3898
$a_{0,\phi}$	49.9070
$b_{0,\theta}$	42.9685
$a_{1,\theta}$	7.9179
$a_{0,\theta}$	45.4188
$b_{0,\psi}$	4.0083
$a_{0,\psi}$	4.1094
$b_{0,z}$	2.3269
$a_{0,z}$	2.2820

Table 4-2: Identified parameters of the low-level controllers.



(a) Input signal for pitch (u_θ).

(b) Input signal for roll (u_ϕ).

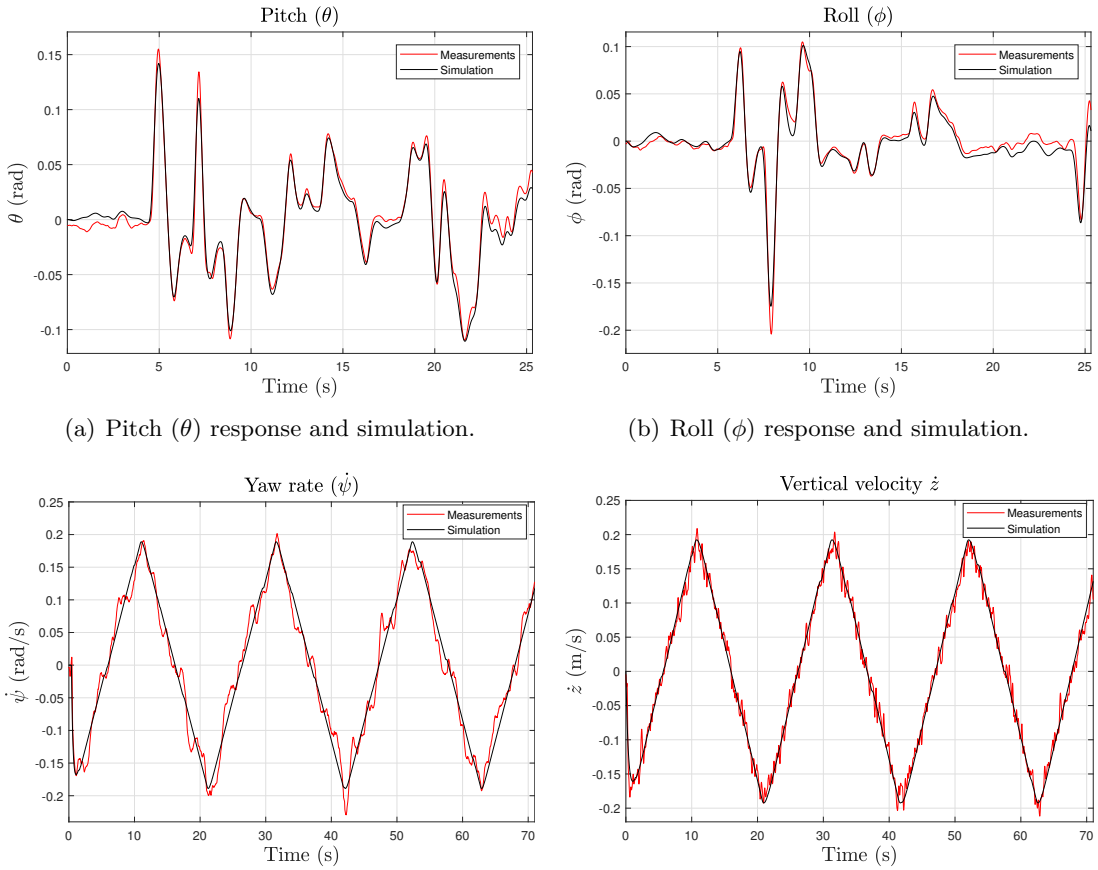
(c) Input signal for yaw rate ($u_{\dot{\psi}}$).

(d) Input signal for vertical velocity ($u_{\dot{z}}$).

Figure 4-7: Validation input signals.

the roll and pitch inputs (u_ϕ and u_θ) are commanded at the same time, while keeping the other two inputs at 0 for safety reasons, as combining all four inputs without a high-level controller can result in severe collisions against the windows or the net (surrounding the workspace) since the movement can be too fast for the user to push the emergency button on time. The

latter two signals were generated by manually piloting the drone and recording the measured angles, which are then post-processed (as explained in Section 4-3-1) and stored. By doing so, we can expect the drone not to crash immediately after starting the experiment, giving time to the user to react if required; besides, the resulting signal has a non-periodic shape, allowing us to better test the identified models.



(a) Pitch (θ) response and simulation.

(b) Roll (ϕ) response and simulation.

(c) Yaw rate ($\dot{\psi}$) response and simulation.

(d) vertical velocity ($\dot{z}$) response and simulation.

Figure 4-8: Validation system identification: simulations vs detrended and/or filtered measurements.

The measured data is detrended, filtered and plotted in Figure 4-8. As expected, both: the simulations and the measurements are alike, despite having thoroughly different input signals compared to those used for identification. Once again, the VAF is used as metric to determine how close the simulation is to the experimental data, the results are presented in Table 4-3, given that all of them are above 96%, we conclude that the identification of the low-level controllers has been validated satisfactorily.

4-3-3 Identification of the Roll and Pitch Trends

As mentioned in Section 4-3-1, the roll and pitch measurements have a trend, here we determine its value. To do so, we take the mean (linear trend) of these two outputs for each

Controller	VAF (%)
Roll (ϕ)	96.4
Pitch (θ)	97.7
Vertical velocity ($\dot{z}$)	98.8
Yaw rate ($\dot{\psi}$)	97.1

Table 4-3: System identification (validation) VAF.

of the experiments showed in the previous sections and, then, we compute their average. This procedure is resumed in Table 4-4. The computed trends (0.0219 in roll, ϕ, and 0.0501 in pitch ,θ) need to be subtracted from the measurements when simulating the system, as explained before, and when using these angles for control. As it can be seen in Table 4-4, the trends change slightly between experiments, thus the user should remember that the calculated averages are only guesses of the real values.

	1	2	3	4	5	6	7	8	Average
Roll (ϕ)	0.0288	0.0223	0.0182	0.0242	0.0183	0.0262	0.0220	0.0149	0.0219
Pitch (θ)	0.0596	0.0632	0.0592	0.0548	0.0439	0.0587	0.0545	0.0483	0.0501

Table 4-4: Roll and pitch trends

4-4 Drag Coefficient Identification

In Chapter 2, it was said that there is a drag coefficient, k_d, that needs to be identified as it is part of Eq. (2-9), which describes the horizontal displacement of the quadrotor and is repeated here for readability:

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} \cos\psi\tan\theta + \sin\psi\tan\phi \\ \sin\psi\tan\theta - \cos\psi\tan\phi \end{bmatrix} g + k_d \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix}.$$

Now that the parameters of the low-level controllers have been found and based on Eq. (2-9), we can utilize grey-box identification, similarly as in Section 4-3, to obtain the value of k_d. For this purpose, the Simulink model displayed in Figure 4-9 is built, where the block called *ddx* contains the non-linear expression of $\ddot{x}$, the *G_* blocks are the identified transfer functions, the input is all four available commands (u_ϕ, u_θ, $u_{\dot{z}}$, and $u_{\dot{\psi}}$), and the output is the position in x.

4-4-1 Results

The identification experiment consists in commanding the pitch signal (u_θ) shown in Figure 4-10(a) while keeping the other three inputs at 0 with the aim of exciting the dynamics in the x-axis. As it can be seen in Figure 4-10(b), the difference between the simulation and the measurements increases with time. In order to analyze the cause of this behavior, the recorded position in x is differentiated twice and the results are plotted alongside the simulation in Figure 4-11. Unsurprisingly, the discrepancy between the simulated and indirectly measured

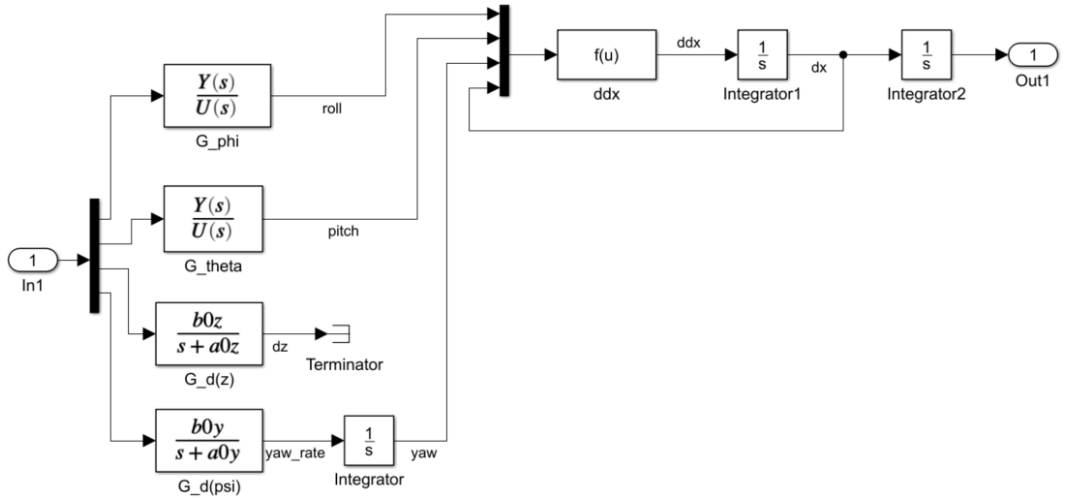
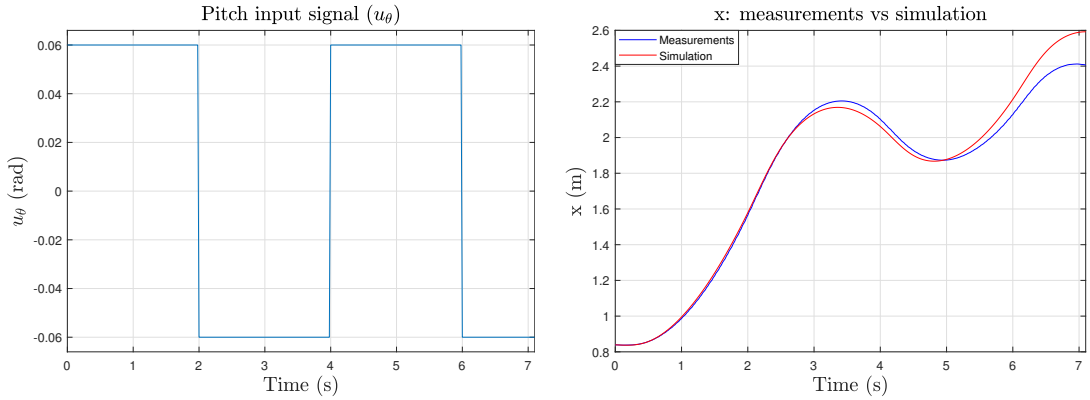


Figure 4-9: Simulink model for the identification of k_d.



(a) Input signal.

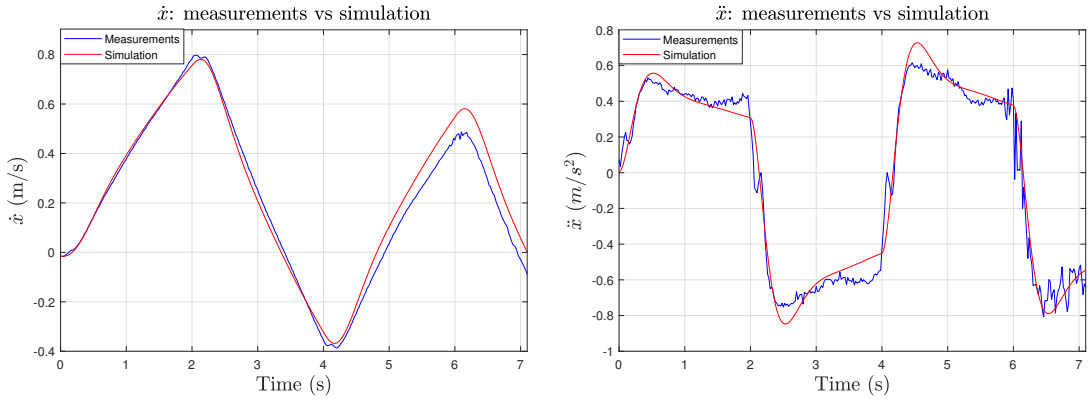
(b) Position x

Figure 4-10: k_d identification experiment: Input, measurements, and simulation.

acceleration is small during the entire experiment, see Figure 4-11(b), in fact the VAF equals 93.6%. Likewise, when observing the velocity plots in Figure 4-11, the two signals have the same shape and almost the same magnitude up to 4 s, from that moment on, the difference is noticeable. Thus, the reasoning is that the *errors* in the acceleration are accumulated as the signal is integrated. However, for the purpose of this project, it is sufficient that the response of the model and the real system are comparable for a short time span because the predictions of the (to-be-designed) MPC will be limited to a given amount of samples. Lastly, the obtained value of k_d is 0.3293.

4-4-2 Validation

To corroborate the value of k_d, the data collected during the validation experiment of the pitch and roll low-level controllers is utilized, i.e. the same inputs are employed, see Figure 4-7(a) and Figure 4-7(b). Besides, the non-linear expression of $\ddot{y}$, see Eq. (2-9) on page 8, is



(a) Velocity $\dot{x}$.

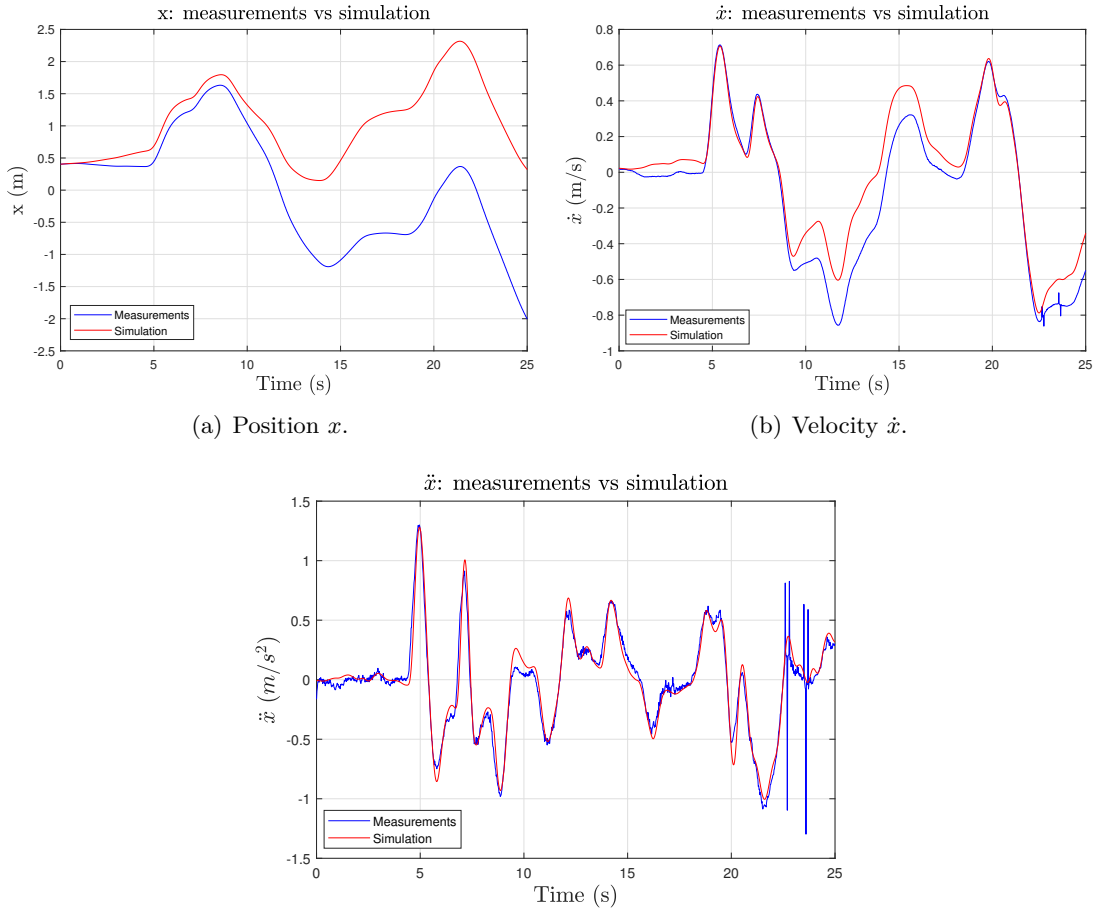
(b) Acceleration $\ddot{x}$.

Figure 4-11: k_d identification experiment: Velocity and acceleration.

included in the simulation.

Regarding the dynamics in x, it can be seen in Figure 4-12(c) that the two signals (measurements and simulation) have mostly the same values, moreover, the VAF between them is 87.9%, which allows us to state that the identification is successful –notice that the sudden peaks in the measurements are due to taking direct derivatives of the position. However and as explained before, once the acceleration is integrated to obtain the velocity displayed in Figure 4-12(b), the amplitude of the simulation differs compared to the measurements, and this effect is more evident in the position. Yet, the shapes are conserved.

On the other hand, the responses in y are plotted in Figure 4-13. Similarly to what happened in x, the simulated and measured acceleration are alike and have a VAF of 83.3%, but discrepancies can be observed in the velocity, see Figure 4-13(b), and even more prominently in Figure 4-13(a). Given that the shapes of all simulated signals are essentially the same as the measured ones and that during the first seconds their amplitudes are similar, we can conclude that the derived models have been validated.

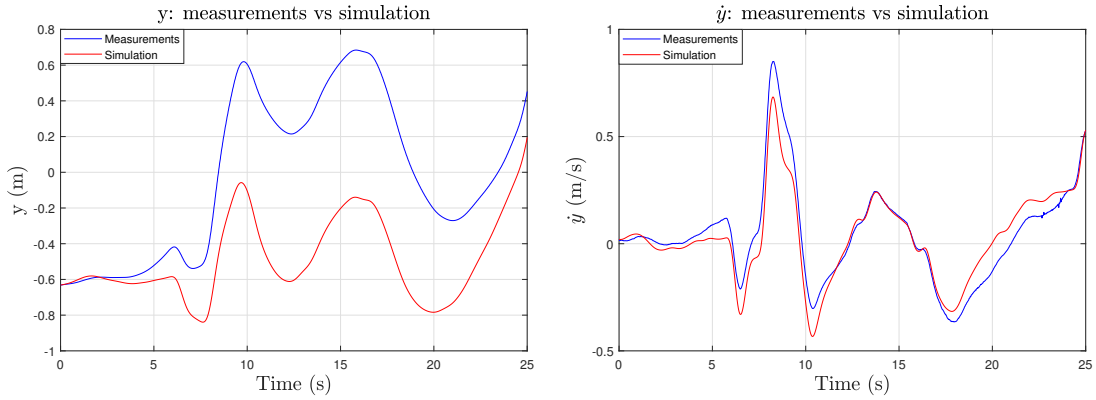


(a) Position x.

(b) Velocity $\dot{x}$.

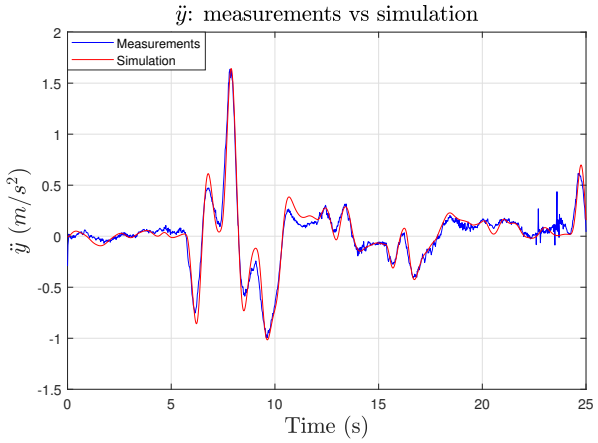
(c) Acceleration $\ddot{x}$.

Figure 4-12: k_d Validation experiment: x dynamics.



(a) Position y.

(b) Velocity $\dot{y}$.



(c) Acceleration $\ddot{y}$.

Figure 4-13: k_d Validation experiment: y dynamics.

Chapter 5

Controller Design and Implementation

In this chapter the control strategies: LQR, MPC, and Kalman observer, are explained and employed to design a position controller for the Matrice 100 using the models identified in Chapter 4. An application programming interface (API) in ROS C++ has been developed to implement the control topology and to interact with the UAV, a functional explanation of this software is also given here. Besides, the attained experimental results are presented and analyzed.

5-1 Discretization and Sampling Time Selection

Before introducing the control topology, let us discretize the models derived in Chapter 2:

$$\begin{aligned} \dot{\mathbf{q}} &= A_q\mathbf{q} + B_q\mathbf{u}, \\ \mathbf{y_q} &= C\mathbf{q}, \end{aligned}$$

and

$$\begin{aligned} \dot{\mathbf{r}} &= A_2\mathbf{r} + B_2 u_{\dot{\psi}}, \\ \mathbf{y_r} &= \mathbf{r}, \end{aligned}$$

where $\mathbf{q} = \begin{bmatrix} \phi & \dot{\phi} & \theta & \dot{\theta} & z & \dot{z} & x & \dot{x} & y & \dot{y} \end{bmatrix}^T$, $\mathbf{y_q} = \begin{bmatrix} \phi & \dot{\phi} & \theta & \dot{\theta} & z & \dot{z} & x & y \end{bmatrix}^T$, $\mathbf{r} = \begin{bmatrix} \psi & \dot{\psi} \end{bmatrix}^T$, and $\mathbf{u} = \begin{bmatrix} u_\phi & u_\theta & u_{\dot{z}} \end{bmatrix}^T$. As discretization method, we apply a zero-order hold approximation for a given sampling time, h, yielding [33]:

$$\begin{aligned} \mathbf{q}(k+1) &= A\mathbf{q}(k) + B\mathbf{u}(k), \\ \mathbf{y_q}(k) &= C\mathbf{q}(k), \end{aligned} \tag{5-1}$$

with matrices

$$A = e^{A_q h},$$
$$B = \int_0^h e^{A_q s} ds B_q,$$

where $e^{\cdot}$ is the matrix exponential and k is the time index. Similarly we have:

$$\begin{aligned} \mathbf{r}(k+1) &= A_r \mathbf{r}(k) + B_r u_{\dot{\psi}}(k), \\ \mathbf{y_r}(k) &= \mathbf{r}(k), \end{aligned} \tag{5-2}$$

with matrices,

$$A_r = e^{A_2 h},$$
$$B_r = \int_0^h e^{A_2 s} ds B_2.$$

In order to determine the sampling time, h, we can simulate the step response of the identified low-level controllers, obtain its rise time, T_r, and according to the rule of thumbs given in [33] choose h such that $\frac{T_r}{h} \approx 4$ to 10. The resulting rising times, $h = \frac{T_r}{4}$, and $h = \frac{T_r}{10}$ are listed in the following table:

Low-level controller	T_r (s)	$h = \frac{T_r}{4}$ (s)	$h = \frac{T_r}{10}(s)$
G_ϕ	0.2611	0.0653	0.0261
G_θ	0.2714	0.0678	0.0271
$G_{\dot{z}}$	0.9625	0.2406	0.0962
$G_{\dot{\psi}}$	0.5346	0.1336	0.0535

Table 5-1: Step response's rise time of the low-level controllers.

Based on Table 5-1 and the aforementioned rule of thumbs, we have $h_{min} = 0.0261$ s and $h_{max} = 0.0653$ s that translate into $f_{max} = 38$ Hz and $f_{min} = 15$ Hz. Recalling that the measurement of the vertical velocity, $\dot{z}$, is broadcasted at 50 Hz, see Chapter 3, we select a sampling frequency, f_s, of 25 Hz (a factor of 50 Hz), ergo, $h = 0.04$ s.

5-2 Control Topology

The control topology implemented in this thesis is displayed in Figure 5-1. To start explaining this diagram, the reader should remember that from the onboard sensors of the Matrice 100 we gather the following measurements (see Chapter 3): the roll and pitch angles ($\phi(k)$ and $\theta(k)$), the angular rates ($\dot{\phi}(k)$, $\dot{\theta}(k)$ and $\dot{\psi}(k)$), and the vertical velocity ($\dot{z}(k)$); whilst from the motion camera system OptiTrack, we obtain: the position of the drone ($x(k)$, $y(k)$, and $z(k)$) and the yaw angle ($\psi(k)$).

The implemented topology contains two Kalman observers/filters: $\hat{\mathbf{r}}$ *observer* and $\hat{\mathbf{q}}$ *observer*. The first one filters $\mathbf{y_r}(k)$ resulting in the filtered state vector $\hat{\mathbf{r}}(k)$; the second is used to estimate $\dot{x}(k)$ and $\dot{y}(k)$ while also filtering $\mathbf{y_q}(k)$, producing the state vector $\hat{\mathbf{q}}(k)$.

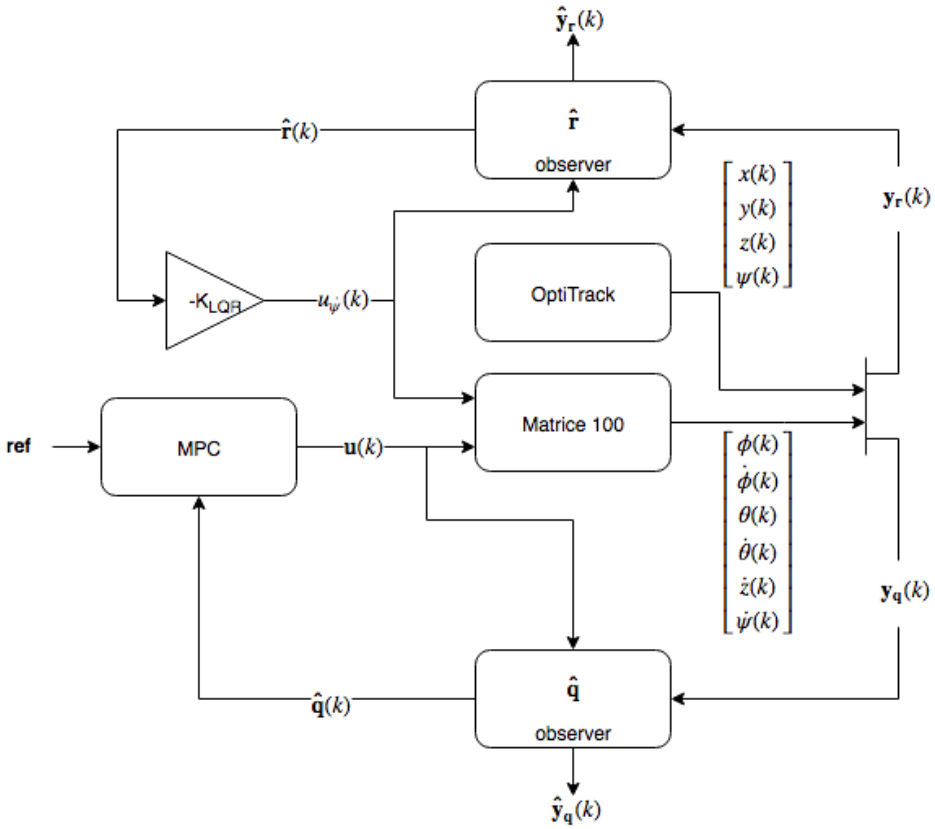


Figure 5-1: Implemented control topology.

As it was mentioned in Chapter 2, for control we need the yaw angle, ψ, to be constant because the linearization of the x-y dynamics depends on it. Hence, Eq. (5-1) is obtained for a particular value $\psi = \psi_0$, which we have chosen to be 0 rad. To satisfy this condition, we use an LQR controller that regulates the yaw angle, see the gain $-K_{LQR}$ in Figure 5-1.

The MPC in Figure 5-1 is in charge of computing the control action $\mathbf{u}(k)$ to track the reference $\mathbf{ref} = \begin{bmatrix} z_{ref} & x_{ref} & y_{ref} \end{bmatrix}^T$, i.e., it controls the position of the drone. The loop is closed by feeding back the estimated $\hat{\mathbf{q}}(k)$ to the MPC.

The output of both controllers is then fed to the drone (Matrice 100) and to the corresponding observers.

5-3 Application Programming Interface (API)

Figure 5-2 illustrates the application programming interface (API), coded in ROS C++, that has been developed to implement the described control topology in real-time and comprises four classes: LinearMPC, Observer, LQ, and SubAndPub, a general-purpose library: Utilities, and the CVXGEN-generated solver needed by the MPC, see Section 5-6-1. This interface has been coded such that, to the largest extend possible, it can be used not only in the particular control case treated in this thesis, but wherever any of the blocks in Figure 5-1

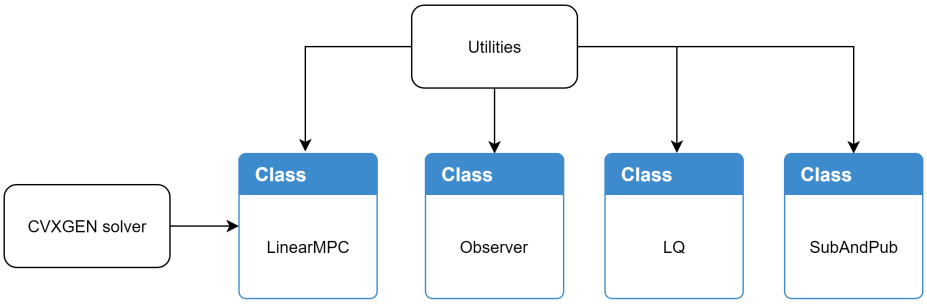


Figure 5-2: Developed API.

are required, i.e. an observer, an LQR/LQI, an MPC, and an interface with the Matrice 100, either outdoors (using GPS) or indoors (using OptiTrack). For this reason, each class is independent from the others, but all of them import the Utilities library and diverse header files from ROS packages. Despite the flexibility of this API, users are recommended to read the documentation (available alongside the source code upon request) before using it in order to know its limitations or specific conditions, e.g. format of the text files containing pre-computed matrices, observer structure, class constructors, etc.

The CVXGEN solver is an auto-generated library from CVXGEN necessary to execute the online optimization of the MPC. Consequently, it works only for the MPC formulation that will be presented in Section 5-6-2.

The LinearMPC class, per contra, is general enough to handle other kind of CVXGEN-generated MPCs or, at least, slight modifications or additions would be required for this purpose.

The Observer class can be directly used to implement any LTI observer as described in Section 5-5. However, this class does not compute the Kalman (observer) gain K, the user needs to calculate it beforehand and import it.

The LQ class allows us to implement either an LQR or a Linear-Quadratic-Integral control (LQI), see [34]. Alike to the Observer class, LQ does not compute the corresponding feedback gain K_{LQR} (or K_{LQI} in case of an LQI), so the user needs to use a third-party software for that. Now that we cannot include the actuator limits within the LQR or LQI formulation, this class incorporates a saturation check whose bounds can be set by the user or imported from a file.

Finally, the class SubAndPub handles the ROS subscribers, publishers, and services, making it the interface with the Matrice 100. Although not explicitly mentioned before, this class was also used in the algorithm for I/O data acquisition presented in Figure 4-2 on page 21.

The main algorithm in Figure 3-9 on page 17 utilizes all four classes and the Utilities library to execute the controller as well as to log the sensor data, observer estimates, control actions, and reference signals. These logs are imported in MATLAB afterwards to assess the controller's performance.

Now that both the control topology and the software implementation have been described, we move on to the observer and controller design, which is done in MATLAB via simulation using

the identified models and is implemented later by means of the API for real-life experiments with Matrice 100.

5-4 Kalman Observers

Before explaining how the observers in Figure 5-1 are tunned, we present the basics of an LTI (steady-state) Kalman observer. For explanation purposes, let us introduce additive process and sensor noises to the system model from Eq. (5-1):

$$\begin{aligned} \mathbf{q}(k+1) &= A\mathbf{q}(k) + B\mathbf{u}(k) + \mathbf{w_q}(k), \\ \mathbf{y_q}(k) &= C\mathbf{q}(k) + \mathbf{v_q}(k), \end{aligned} \tag{5-3}$$

where $\mathbf{q}(k) \in \mathbb{R}^n$, $\mathbf{u}(k) \in \mathbb{R}^m$, and $\mathbf{w_q}(k) \in \mathbb{R}^n$ and $\mathbf{v_q}(k) \in \mathbb{R}^m$ are zero-mean white noise sequences with joint covariance matrix:

$$E\left[\begin{bmatrix}\mathbf{w_q}(k)\\ \mathbf{v_q}(k)\end{bmatrix}\begin{bmatrix}\mathbf{w_q}(j)^T & \mathbf{v_q}(j)^T\end{bmatrix}\right] = \begin{bmatrix}Q_q & 0\\ 0 & R_q\end{bmatrix}\Delta(k-j) \geq 0, \tag{5-4}$$

$R_q > 0$ and $\Delta(k)$ is the unit pulse. The Kalman observer for this system takes the form of Eq. (5-5) [30]:

$$\begin{aligned} \hat{\mathbf{q}}(k+1) &= A\hat{\mathbf{q}}(k) + B\mathbf{u}(k) + K_q(\mathbf{y_q}(k) - \hat{\mathbf{y}}_\mathbf{q}(k)) \\ &= (A - K_qC)\hat{\mathbf{q}}(k) + B\mathbf{u}(k) + K_q\mathbf{y_q}(k), \\ \hat{\mathbf{y}}_\mathbf{q}(k) &= C\hat{\mathbf{q}}(k), \end{aligned} \tag{5-5}$$

where K_q is the so-called Kalman gain computed from the positive-definite solution, $P \in \mathbb{R}^{n\times n}$, of the following Riccati-type equation:

$$\begin{aligned} P &= APA^T + Q_q - APC^T(CPC^T + R_q)^{-1}CPA^T, \\ K_q &= APC^T(CPC^T + R_q)^{-1}. \end{aligned} \tag{5-6}$$

5-4-1 $\hat{\mathbf{q}}$ Observer Design

Based on the previous explanation, we design the $\hat{\mathbf{q}}$ *observer* shown in Figure 5-1. The Kalman gain, K_q, is found using MATLAB's *dare()* command (with A^T and C^T as the first two arguments, and transposing the returned gain matrix) [35], which solves the Riccati equation presented in Eq. (5-6). For this controller we employ the subsequent variance matrices:

$$\begin{aligned} R_q = diag(&1.4\times10^{-5}, 1.59\times10^{-4}, 1.59\times10^{-5}, 1.63\times10^{-4}, 5\times10^{-4}, 1.99\times10^{-4}, \\ &5\times10^{-4}, 7.5\times10^{-5}), \\ Q_q = diag(&1\times10^{-5}, 1\times10^{-5}, 1\times10^{-5}, 1\times10^{-5}, 1\times10^{-5}, 1\times10^{-5}, 1\times10^{-4}, 5\times10^{-5}, \\ &1\times10^{-5}, 5\times10^{-5}). \end{aligned} \tag{5-7}$$

In general, the diagonal values of R_q can be found by letting the drone hover with zero input, i.e. $u_\phi = u_\theta = u_{\dot{z}} = u_{\dot{\psi}} = 0$, and then computing the variance of the measurements. This method produces an estimate of the sensor's noise variance. Notwithstanding, there is no similar recipe to determine the variance matrix of the process noise, Q_q, the only hint we have is that it is most likely on the same order of magnitude as R_q or close to it, else, the noise would be too large and would be reflected in the measurements.

On the other hand, note that $\mathbf{y_q}(k)$ includes the roll and pitch angles whose trends were estimated in Chapter 4 and are subtracted by the observer before estimating the states.

5-4-2 $\hat{\mathbf{r}}$ Observer Design

Analogously to Eq. (5-5), the state-space description of the $\hat{\mathbf{r}}$ *observer* is:

$$\begin{aligned}\hat{\mathbf{r}}(k+1) &= A_r\hat{\mathbf{r}}(k) + B_r u_{\dot{\psi}}(k) + K_r(\mathbf{y_r}(k) - \hat{\mathbf{y}}_\mathbf{r}(k)) \\ &= (A_r - K_r)\hat{\mathbf{r}}(k) + B_r u_{\dot{\psi}}(k) + K_r\mathbf{y_r}(k), \\ \hat{\mathbf{y}}_\mathbf{r}(k) &= \hat{\mathbf{r}}(k),\end{aligned} \tag{5-8}$$

Once again, the *dare()* command from MATLAB is used to compute the Kalman gain, K_r. In this occasion, the variance matrices are:

$$\begin{aligned}R_r &= \begin{bmatrix} 6.9\times10^{-5} & 0 \\ 0 & 9\times10^{-5}\end{bmatrix}, \\ Q_r &= \begin{bmatrix} 1\times10^{-4} & 0 \\ 0 & 1\times10^{-4}\end{bmatrix}.\end{aligned} \tag{5-9}$$

The selection of the diagonal values of R_r and Q_r follows the same logic as per R_q and Q_q.

5-5 Linear-Quadratic Regulator (LQR)

The first controller we are interested in is an LQR to regulate the yaw angle, ψ, as commented in Section 5-2. The fundamentals regarding this type of controller are explicated first, followed by the design procedure and the corresponding simulation.

5-5-1 Fundamentals

To begin with, the implemented LQR is an optimal state feedback controller that applies to the system the solution of the subsequent optimization problem [33, 36]:

$$\begin{aligned}\min \quad & \sum_{k=0}^{\infty}\left(\mathbf{r}(k)^T Q_{LQR}\mathbf{r}(k) + u_{\dot{\psi}}(k)R_{LQR}u_{\dot{\psi}}(k)\right), \\ \text{s.t.} \quad & \mathbf{r}(k+1) = A_r\mathbf{r}(k) + B_r u_{\dot{\psi}}(k),\end{aligned} \tag{5-10}$$

where $Q_{LQR} \in \mathbb{R}^{2\times2}$ and $R_{LQR} > 0$. The solution of this problem is based on the unique positive-definite solution, $P_{LQR} \in \mathbb{R}^{2\times2}$, of the following discrete-time algebraic Riccati equation (DARE) [33, 35]:

$$A_r^T P_{LQR} A_r - P_{LQR} - A_r^T P_{LQR} B_r (B_r^T P_{LQR} B_r + R_{LQR})^{-1} B_r^T P_{LQR} A_r + Q_{LQR} = 0. \quad (5\text{-}11)$$

The optimal feedback gain, K_{LQR}, and the control action, $u_{\psi}(k)$, are given by:

$$\begin{aligned} K_{LQR} &= (B_r^T P_{LQR} B_r + R_{LQR})^{-1} B_r^T P_{LQR} A_r, \\ u_{\psi}(k) &= -K_{LQR}\mathbf{r}(k). \end{aligned} \quad (5\text{-}12)$$

By applying this optimal control input as presented in Figure 5-1, one can drive all states to the origin.

5-5-2 LQR Design

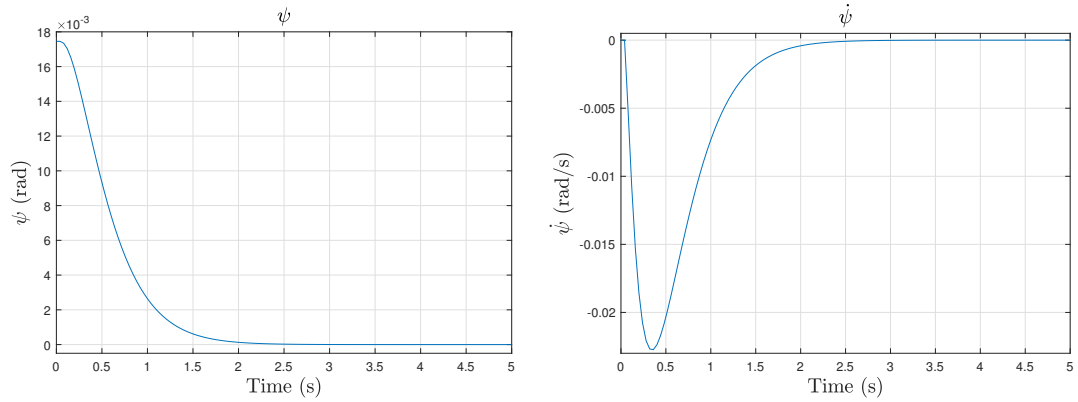
In order to tune this controller, we: i) vary the weighting matrices Q_{LQR} and R_{LQR}, ii) compute the corresponding feedback gain K_{LQR} using MATLAB's *lqr()* command [36], and iii) simulate the closed loop (including the $\hat{\mathbf{r}}$ *observer*) to determine if the achieved performance is satisfactory, if not, we repeat this procedure. The performance criteria are: i) ψ should converge softly to 0 rad, ii) the settling time should be less than 3 s, and iii) the control action should be as low as possible. Following these conditions, we arrived to the subsequent weighting matrices:

$$\begin{aligned} Q_{LQR} &= \begin{bmatrix} 20 & 0 \\ 0 & 1 \end{bmatrix}, \\ R_{LQR} &= 2. \end{aligned} \quad (5\text{-}13)$$

As it can be seen, Q_{LQR} has high penalty on the first state, $\psi(k)$, to ensure fast convergence to 0 rad. Additionally, the control action, u_{ψ}, is also penalized to reduce its magnitude. The system is simulated with initial condition $\mathbf{r}(0) = \begin{bmatrix} \pi/180 & 0 \end{bmatrix}^T$ and the results are plotted in Figure 5-3. The settling time is roughly 2 s, thus within the specified limit, and the control action's peak is 0.042 rad/s (or 2.4 °/s) that we consider small enough. Furthermore, the regulation is smooth as desired. This last characteristic is relevant to avoid abrupt rotations of the aircraft.

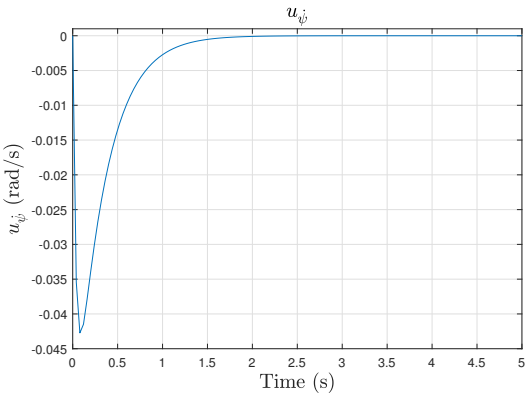
5-6 Model Predictive Control (MPC)

The second control strategy implemented in this thesis is the named Model Predictive Control (MPC), also known as Receding Horizon Control (RHC), which solves a finite horizon, open-loop optimal control problem online [37]. Every time instant, k, this controller computes the solution the following optimization problem:



(a) Simulated ψ

(b) Simulated $\dot{\psi}$



(c) Simulated u_{ψ}

Figure 5-3: LQR simulation.

$$
\begin{aligned}
\min_{\mathbf{q_{MPC}},\mathbf{U}} \quad & J(\mathbf{q_{MPC}}, \mathbf{U}), \\
\text{s.t.} \quad & \mathbf{q}(k+i) = A\mathbf{q}(k+i-1) + B\mathbf{u}(k+i-1) \quad \forall\, i = 1, \ldots, N_c, \\
& \mathbf{q}(k+i) = A\mathbf{q}(k+i-1) + B\mathbf{u}(k+Nc-1) \quad \forall\, i = N_c+1, \ldots, N_p, \\
& g(\mathbf{q_{MPC}}, \mathbf{U}) \leq 0,
\end{aligned}
\tag{5-14}
$$

where $N_c \leq N_p$ are the control and prediction horizons, respectively, $\mathbf{q_{MPC}} \in \mathbb{R}^{nN_p}$ and $\mathbf{U} \in \mathbb{R}^{mN_c}$ are vectors grouping all predicted states and inputs throughout the prediction and control horizons, the expressions for $\mathbf{q}(k+i)$ are the prediction models, $g(\mathbf{q_{MPC}}, \mathbf{U})$ is the set of state and/or input constraints, and $J(\mathbf{q_{MPC}}, \mathbf{U})$ is the cost function. Although the prediction model shown in Eq. (5-14) is presented as a discrete-time state-space system, any type of representation is also valid, e.g. transfer function or nonlinear system [10].

Figure 5-4 illustrates an example of the optimization performed by an MPC at time k, assuming there are only one input, $u(k)$, and one state, $q(k)$. The controller solves Eq. (5-14) based on the current state, $q(k)$, calculating the control input sequence $\mathbf{U} = [u(k) \quad u(k+1) \quad \cdots \quad u(k+N_c-1)]^T$ that minimizes the cost function, while employing the prediction model to construct the vector $\mathbf{q_{MPC}} = [q(k+1) \quad q(k+2) \quad \cdots \quad q(k+Np)]^T$ and satisfying

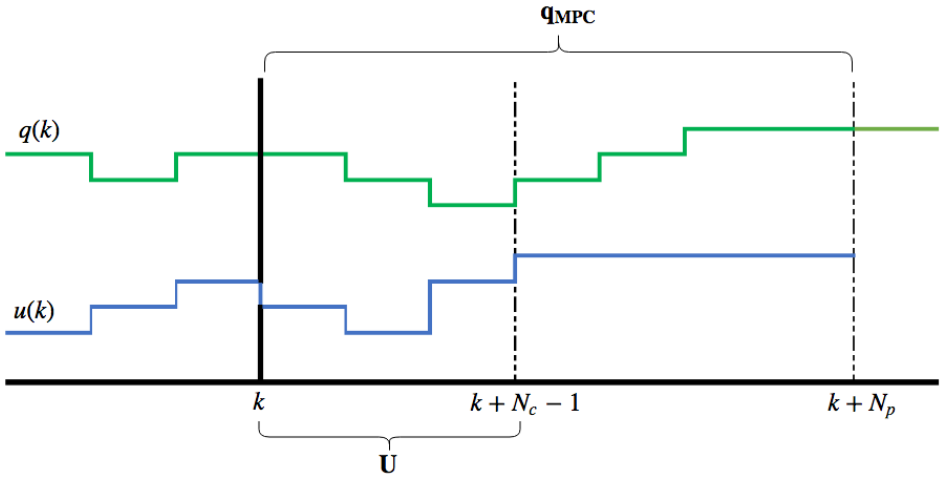


Figure 5-4: MPC example.

the inequality constraints. In the figure, one can also notice that the control action is held constant between $k+N_c-1$ and $k+N_p-1$, doing so the number of optimization variables and constraints is reduced, decreasing the problem's complexity.

Despite solving an open-loop optimization problem, the closed-loop response is achieved by updating the state vector, $\mathbf{q}(k)$, every sampling time, leading to the receding horizon control that consists in applying to the system the attained $\mathbf{u}(k)$ and solving the optimization problem again the next time instant with the new state vector [38]. The receding horizon approach is useful to incorporate possible model mismatch, time varying behavior, and disturbances [10] in a different fashion from other optimal controllers such as LQR, where the solution is found off-line, hence, taking the same form all time instances.

The selection of the cost function, on the other hand, depends on what type of solution we want. For instance, an ℓ_1 norm would produce a sparse result, but relatively high deviations can be expected; an ℓ_∞ norm penalizes more the maximum deviation but all deviations would have approximately the same value; in contrast, a quadratic cost would minimize the average deviation, resulting in rather small deviations [39, 40]. Irrespective of the selected type of penalty, it can be applied to the states, the outputs, or the inputs. Moreover, the cost function can also contain a final state penalty, which, if tunned properly, can reduce the complexity of the optimization problem as it may allow shorter-horizon controllers to still produce a similar performance to controllers with longer horizons [38].

Based on the above analysis, the cost function for this thesis' MPC is:

$$J(\mathbf{q_{MPC}}, \mathbf{U}) = \sum_{i=0}^{N_p-1} \|(C_{tracking}\mathbf{q}(k+i) - \mathbf{ref})\|^2_{Q_{MPC}} + \sum_{i=0}^{N_c-1} \|\mathbf{u}(k+i)\|^2_{R_{MPC}} + \|\mathbf{q}(k+N_p)\|^2_{Q_n}, \tag{5-15}$$

where $\|\mathbf{u}(k+i)\|^2_{R_{MPC}} = \mathbf{u}(k+i)^T R_{MPC}\mathbf{u}(k+i)$ and similarly for the other two expressions, $Q_{MPC} \in \mathbb{R}^{3\times3}$, $R_{MPC} \in \mathbb{R}^{3\times3}$, and $Q_n \in \mathbb{R}^{10\times10}$ are the weighting matrices –whose values will be presented in Section 5-6-2–,

$$C_{tracking} = diag\left(\begin{bmatrix}0 & 0 & 0 & 0 & 1\end{bmatrix}, \begin{bmatrix}0 & 1\end{bmatrix}, \begin{bmatrix}0 & 1 & 0\end{bmatrix}\right),$$
$$\mathbf{q_{MPC}} = \begin{bmatrix}\mathbf{q}(k+1)^T & \mathbf{q}(k+2)^T & \dots & \mathbf{q}(k+N_p)^T\end{bmatrix}^T,$$

and $\mathbf{U} = \begin{bmatrix}\mathbf{u}(k)^T & \mathbf{u}(k+1)^T & \dots & \mathbf{u}(k+N_c-1)^T\end{bmatrix}^T$.

Given the value of $C_{tracking}$, we have: $C_{tracking}\mathbf{q}(k) = \begin{bmatrix}z(k) & x(k) & y(k)\end{bmatrix}^T$. The reason for not using a separate vector, e.g. $\mathbf{y_{tracking}}(k) = \begin{bmatrix}z(k) & x(k) & y(k)\end{bmatrix}^T$, for this product is that it would require the definition of a third optimization variable in CVXGEN, which from its perspective would be a more complex problem.

By employing this cost function, one can penalize large deviations of the tracking outputs from the reference; furthermore the control input is also penalized to avoid large control actions; and finally, there is a cost for the final state, which as mentioned before, allows us to reduce the prediction horizon (N_p), whilst still achieving the desired performance.

5-6-1 CVXGEN

In order to implement the explained MPC, we need C/C++ code capable of solving Eq. (5-14) online and faster than the sampling time of the system. From the designer's perspective one has, basically, three options: i) to manually transform the optimization problem into any standard form, e.g. Linear Programming (LP) or Quadratic Programming (QP), and then use any available software (e.g. Gurobi or CIPLEX) to solve it; ii) to fully customize the solution if the problem cannot be recast into a standard form; or iii) to utilize an automatic solver generator to produce the desired solver (almost) directly from the original formulation of the problem. The first and second approaches require previous knowledge in optimization techniques and can be highly time consuming as the transformation or customization procedures have to be repeated every time the problem is changed (e.g. new constraints are added, the sizes of the matrices are varied, the cost function is modified, etc.); whereas the third option provides flexibility due to its high-level description, making it more appealing for application-driven projects, where the designer alters the original formulation multiple times before arriving to a final controller. However, the latter may be limiting for the intended application or the designer may prefer to minimize the computation time as much as possible, favoring then a custom solution. Now that this project focuses on the application side rather than in optimization itself, the third option is more convenient.

The code generator for embedded convex optimization CVXGEN is chosen to generate the solver because of its easy-to-use interface, which provides a framework for quick prototyping of sophisticated controllers, such as MPC [38]. To be more explicit, CVXGEN's editor consists of four sections: i) *dimensions*, where the user can define various constants, such as the prediction and control horizons, and the number of states, outputs and inputs; ii) *parameters*, where the inputs of the solver are declared, e.g. weighting matrices, current state vector, reference signal, etc; iii) *variables*, where the optimization variables are defined, e.g. $\mathbf{U}$ and $\mathbf{q_{MPC}}$ (using CVXGEN's notation, evidently); and iv) *minimize*, where the optimization problem is inserted in two parts: the cost function and the constraints (under the *subject to*

tag). The reader is referred to [41] for an exhaustive explanation of CVXGEN and to Listing (5.1) for a concrete illustration of its editor.

CVXGEN utilizes *disciplined convex programming* techniques to ensure that valid problem descriptions are convex [41]. Disciplined convex programming is made of a set of functions (known to be convex) grouped into a library named *atom* and a *convexity ruleset*, derived from the principles of convex analysis, that assesses convexity of a given problem constructed with functions from the atom library [42]. This means that CVXGEN's users are restrained to a collection of pre-defined functions to formulate their optimization problems and, more importantly, the current version of the software only support problems that can be recast into Quadratic Programming (QP) [41]. The other relevant limitation of CVXGEN is the number of non-zero entries in the so-called Karush–Kuhn–Tucker (KKT) matrix constructed by the software and whose form is explicated next.

CXVGEN transforms the high-level description of the optimization problem into a standard (or canonical) QP as shown in Eq. (5-16):

$$\begin{aligned} \min \quad & \frac{1}{2}\mathbf{x_{qp}}^T Q_{qp}\mathbf{x_{qp}} + q_{qp}^T\mathbf{x_{qp}}, \\ \text{s.t.} \quad & G_{qp}\mathbf{x_{qp}} \leq h_{qp}, \\ & A_{qp}\mathbf{x_{qp}} = b_{qp}, \end{aligned} \tag{5-16}$$

where $\mathbf{x_{qp}} \in \mathbb{R}^{n_{qp}}$ is the optimization variable, $Q_{qp} > 0 \in \mathbb{R}^{n_{qp}\times n_{qp}}$, $q_{qp} \in \mathbb{R}^{n_{qp}}$, $G_{qp} \in \mathbb{R}^{p_{qp}\times n_{qp}}$, $h_{qp} \in \mathbb{R}^{p_{qp}}$, $A_{qp} \in \mathbb{R}^{m_{qp}\times n_{qp}}$, and $b_{qp} \in \mathbb{R}^{m_{qp}}$. This problem is solved by CXVGEN via primal-dual interior point, therefore, it introduces a slack variable $\mathbf{s_{qp}} \in \mathbb{R}^{p_{qp}}$ and reformulates the problem as:

$$\begin{aligned} \min \quad & \frac{1}{2}\mathbf{x_{qp}}^T Q_{qp}\mathbf{x_{qp}} + q_{qp}^T\mathbf{x_{qp}}, \\ \text{s.t.} \quad & G_{qp}\mathbf{x_{qp}} + \mathbf{s_{qp}} = h_{qp}, \\ & A_{qp}\mathbf{x_{qp}} = b_{qp}, \\ & \mathbf{s_{qp}} \geq 0, \end{aligned} \tag{5-17}$$

with optimization variables $\mathbf{x_{qp}}$ and $\mathbf{s_{qp}}$. Then it inserts the dual variables $\mathbf{y_{qp}} \in \mathbb{R}^{m_{qp}}$ associated with the equality constraints, and $\mathbf{z_{qp}} \in \mathbb{R}^{p_{qp}}$ associated with the inequality constraints. The resulting KKT conditions are [41]:

$$\begin{aligned} G_{qp}\mathbf{x_{qp}} + \mathbf{s_{qp}} &= h, \\ A_{qp}\mathbf{x_{qp}} &= b_{qp}, \\ \mathbf{z_{qp}} &\geq 0, \\ Q_{qp}\mathbf{x_{qp}} + q_{qp} + G_{qp}^T\mathbf{z_{qp}} + A_{qp}^T\mathbf{y_{qp}} &= 0, \\ \mathbf{z_{qp}}_i\mathbf{s_{qp}}_i &= 0, \quad i = 1, \ldots, p_{qp}. \end{aligned} \tag{5-18}$$

Now, let $S_{qp} = diag(\mathbf{s_{qp}})$ and $Z_{qp} = diag(\mathbf{z_{qp}})$, then the KKT matrix equals [41]:

$$\begin{bmatrix} Q_{qp} & 0 & G_{qp}^T & A_{qp}^T \\ 0 & Z_{qp} & S_{qp} & 0 \\ G_{qp} & I & 0 & 0 \\ A_{qp} & 0 & 0 & 0 \end{bmatrix}, \tag{5-19}$$

where I is an identity matrix of appropriate size. CVXGEN is able to generate reliable and fast C-code as long as there are up to 4000 non-zero entries in the foregoing matrix. Although the mentioned limits may seem narrow, it will be shown in Section 5-6-2 that we are still capable of achieving good results using this code generator.

Besides the C code, CVXGEN creates a MATLAB MEX interface that allows us to use the resulting solver in MATLAB, through which we are able to design and simulate the MPC before testing it with the real Matrice 100.

5-6-2 MPC Design

Now that the MPC framework has been explained, we proceed to the design and simulation phases by first presenting the to-be-used optimization problem in Eq. (5-20), which uses the cost function from Eq. (5-15), the prediction model from Eq. (5-14) and defines the constraints therein:

$$\begin{aligned} \min_{\mathbf{q_{MPC}},\mathbf{U}} \quad & \sum_{i=0}^{N_p-1} \|(C_{tracking}\mathbf{q}(k+i) - \mathbf{ref})\|^2_{Q_{MPC}} + \sum_{i=0}^{N_c-1} \|\mathbf{u}(k+i)\|^2_{R_{MPC}} + \|\mathbf{q}(k+N_p)\|^2_{Q_n}, \\ \text{s.t.} \quad & \mathbf{q}(k+i) = A\mathbf{q}(k+i-1) + B\mathbf{u}(k+i-1) \quad \forall\, i = 1,\dots,N_c, \\ & \mathbf{q}(k+i) = A\mathbf{q}(k+i-1) + B\mathbf{u}(k+Nc-1) \quad \forall\, i = N_c+1,\dots,N_p, \\ & \mathbf{u_{min}} \le \mathbf{u}(k+i) \le \mathbf{u_{max}} \quad \forall\, i = 0,\dots,N_c-1, \\ & \mathbf{y_{min}} \le C_{tracking}\mathbf{q}(k+i) \le \mathbf{y_{max}} \quad \forall\, i = 1,\dots,N_p-8, \end{aligned} \tag{5-20}$$

where $\mathbf{u_{min}} = \begin{bmatrix} u_{\phi_{min}} & u_{\theta_{min}} & u_{\dot{z}_{min}} \end{bmatrix}^T$ and $\mathbf{u_{max}} = \begin{bmatrix} u_{\phi_{max}} & u_{\theta_{max}} & u_{\dot{z}_{max}} \end{bmatrix}^T$ are the actuator limits, $\mathbf{y_{min}} = \begin{bmatrix} z_{min} & x_{min} & y_{min} \end{bmatrix}^T$ and $\mathbf{y_{max}} = \begin{bmatrix} z_{max} & x_{max} & y_{max} \end{bmatrix}^T$ are the bounds of the tracking outputs.

Eq. (5-20) is entered in CVXGEN as follows:

Listing 5.1: MPC implementation in CVXGEN's editor.

```
dimensions
  n = 10 # states.
  m = 3 # inputs.
  l = 3 # tracked outputs.
  Np = 10 # Prediction horizon
  Nc = 4 # Control horizon
end

parameters
```

```
10    A (n,n) # dynamics matrix.
11    B (n,m) # transfer matrix.
12    C (l,n) # Tracking output matrix (C_tracking).
13    Q (l,l) psd # tracking cost (Q_MPC).
14    R (m,m) psd # input cost (R_MPC).
15    Qn (n,n) psd
16    q[0] (n) # initial state.
17    u_max (m) # control action upper limit.
18    u_min (m) # control action lower limit.
19    y_max (m) nonnegative # output upper limit
20    y_min (m) # output lower limit
21    ref (l) # reference signal
22  end
23
24  variables
25    q[k] (n), k=1..Np # states (q_{MPC})
26    u[k] (m), k=0..Nc-1 # inputs (U)
27  end
28
29  minimize
30    sum[k=0..Np-1](quad(C*q[k] - ref,Q)) + sum[k=0..Nc-1](quad(u[k],R)) +
          quad(q[Np],Qn)
31  subject to
32    q[k+1] == A*q[k] + B*u[k], k=0..Nc-1
33    q[k+1] == A*q[k] + B*u[Nc-1], k=Nc..Np-1
34    u_min <= u[k] <= u_max, k=0..Nc-1
35    y_min <= C*q[k] <= y_max, k=1..Np-8
36  end
```

In the above code, it can be easily noticed that porting the MPC formulation to CVXGEN is straightforward. The interested reader is invited to read CVXGEN's documentation [41] for a detailed description of the syntax.

In order to select the prediction horizon, we increase it until just before reaching the limits of CVXGEN (4000 non-zero KKT-matrix entries), while considering that the MPC should predict the states for a reasonable time frame and that the sampling time, h, equals 0.04 s, see Section 5-1. At that point, we start increasing the control horizon following the same logic. The reached values are: $N_p = 10$ and $N_c = 4$, consequently, the MPC predicts the states up to 0.4 s ahead and computes the control actions until 0.12 s into the *future*. As a remark, lower control horizons are capable of stabilizing the system, but unexpected oscillations occur after reaching "steady state", in fact, the lower the control horizon, the more prominent this effect is.

In Section 4-4, it was stated that the response of the identified model and the real drone are comparable for up to 4 s, which implies that the open-loop predictions of the MPC throughout the entire prediction horizon are reliable as it is one order of magnitude shorter.

On the other hand, having $N_p = 10$ means that the output constraints in Eq. (5-20) are imposed only on the first and second predicted set of outputs and not for the complete prediction horizon. Constraining more outputs would cause more than 4000 non-zero KKT-matrix entries. However, ensuring that those two sets are within the limits is sufficient to

secure that the drone will remain within the desired boundaries after applying the resulting optimal control input $\mathbf{u}(k)$.

To determine the actuator and output limits we consider the following aspects: the diameter of the Matrice 100 with the propellers attached is almost 1 m, the dimensions of the workspace are 6.0 × 3.0 × 2.6 m (length × width × height), see Chapter 3, and we assume the origin of the world reference system ({W}) to be on the floor (0 m height) and at the center of the workspace's horizontal plane. Then, we set $\mathbf{y_{max}} = [2.2 \quad 2.75 \quad 0.75]^T$, and $\mathbf{y_{min}} = [0.15 \quad -2.75 \quad -0.75]^T$ –notice that the minimum height is not 0 m since when the UAV is on the floor, its center of mass is at 0.2 m approximately–, $\mathbf{u_{max}} = [10\pi/180 \quad 15\pi/180 \quad 1.5]^T$, and $\mathbf{u_{max}} = -\mathbf{u_{max}}$. The limits of the control action can technically be larger (see Table 3-1 in 12), however, we prefer to utilize small angles due to the relatively small workspace.

To tune the weighting matrices Q_{MPC}, R_{MPC}, and Q_n, we define the following criteria: i) the overshoot of the tracking outputs (x, y, and z) should not be larger than 5% to prevent collisions against the windows or the net around the workspace; ii) the rise time should not be longer than 4 s, although this is not a hard constraint because the endurance of the Matrice 100 is between 8 and 10 minutes (with the payload shown in Chapter 3); and iii) the reference tracking should not be oscillatory.

Alike to the process followed to tune the LQR in Section 5-5-2, we simulate the closed-loop system using the model from Eq. (5-1) and the $\hat{\mathbf{q}}$ *observer*. By doing so we arrive to:

$$Q_{MPC} = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 2.5 & 0 \\ 0 & 0 & 5 \end{bmatrix}, \tag{5-21}$$

where $Q_{MPC}(1,1)$ affects the performance of z, $Q_{MPC}(2,2)$ the one of x, and $Q_{MPC}(3,3)$ that of y. Let us evaluate the effect of each of these entries: $Q_{MPC}(1,1) > 1$ produces a faster response and reduces the steady-state error, when it is greater or equal to 3 there is overshoot; $Q_{MPC}(2,2) > 3$ produces oscillations, in fact, the larger it becomes, the more prominent this effect is (to the point of causing instability). Increasing $Q_{MPC}(3,3)$ speeds up the response, but a larger overshoot and more oscillations (if any) may occur. The noticed behaviors are expected as Q_{MPC} penalizes the deviation of x, y, and z from the given set-point **ref**, i.e. it is logical to obtain a faster response for a larger penalty. With respect to the instability caused by a large $Q_{MPC}(2,2)$, it resembles the effect of a proportional controller, where a high enough gain can render the closed-loop response unstable.

The attained second weighting matrix equals:

$$R_{MPC} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0.2 \end{bmatrix}, \tag{5-22}$$

where $R_{MPC}(1,1)$ penalizes u_ϕ, $R_{MPC}(2,2)$ influences u_θ, and $R_{MPC}(3,3)$ affects $u_{\dot{z}}$. During the tunning process, it was found that increasing $R_{MPC}(1,1)$ causes more oscillations in y, a larger overshoot and longer settling time; Reducing $R_{MPC}(2,2)$ speeds up the response of x; and augmenting $R_{MPC}(3,3)$ slows down the response of z while reducing the overshoot (if any). Thence, we decided to favor a low penalty in $R_{MPC}(3,3)$. The effect of R_{MPC} is

unsurprisingly alike to that of R_{LQR} in the LQR controller now that both have a direct impact on the amplitude of the control action. However, it is important to stress on one of the key differences between the MPC and the LQR controllers: the possibility of including the limits of the control action in the MPC formulation, which results in less effort when tunning this weighting matrix.

The remaining weighting matrix is given by:

$$Q_n = diag(1, 1, 1, 1, 0.1, 1, 0.01, 1, 0.01, 2). \tag{5-23}$$

Regarding the response in y: $Q_n(1,1) < 1$ slightly reduces the overshoot; $Q_n(2,2) < 1$ produces a faster response and a smaller steady-state error at the cost of larger overshoot; $Q_n(9,9) < 1$ reduces the overshoot, increases the settling time and lessens the steady-state error; and $Q_n(10,10) < 1$ causes more oscillations, increasing the settling time. With respect to the response in z: $Q_n(5,5) < 1$ reduces the steady-state error by causing overshoot, while $Q_n(6,6) > 1$ has no effect. Concerning the performance in x: $Q_n(3,3) \geq 10$ produces overshoot, $Q_n(4,4) < 1$ has the same outcome, decreasing $Q_n(7,7)$ reduces the steady-state error, and $Q_n(8,8) < 1$ provokes instability. The values in Eq. (5-23) were found to balance all these aspects.

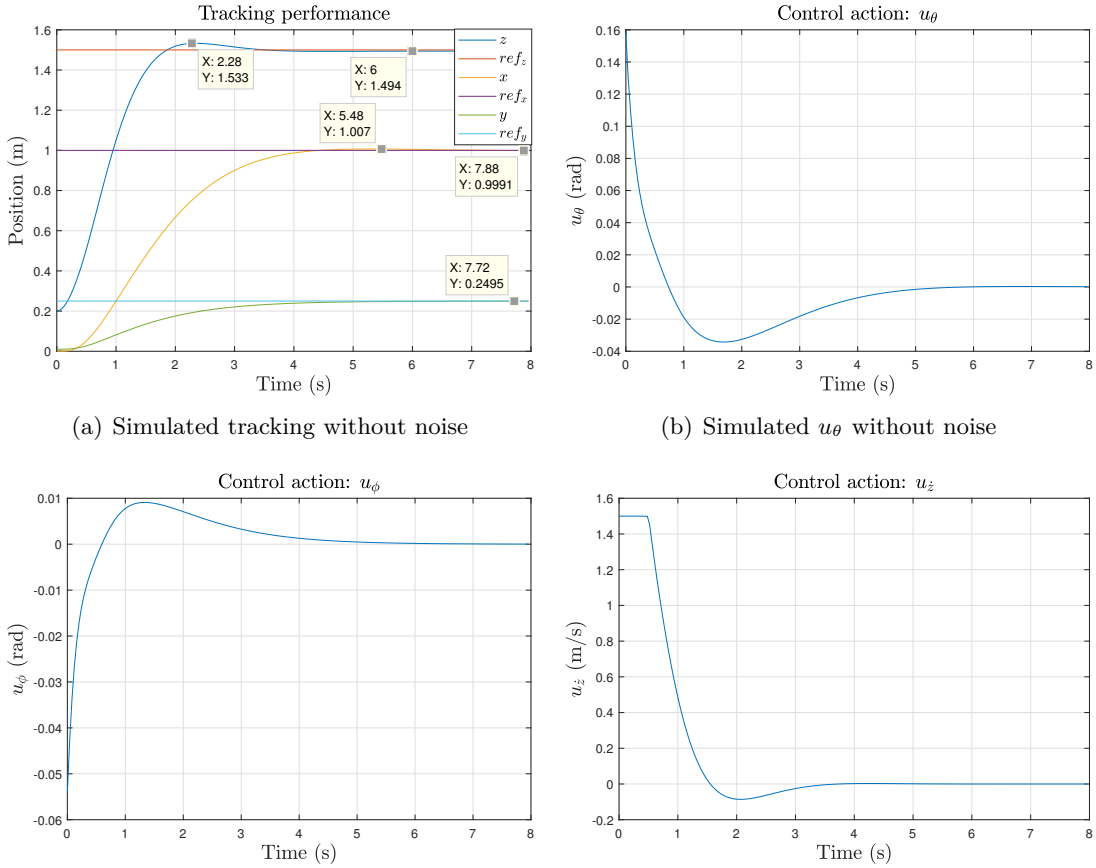
Two simulation cases are performed: with and without sensor noise, the initial condition in both cases is: $\mathbf{q}(0) = [0\ \ 0\ \ 0\ \ 0\ \ 0.2\ \ 0\ \ 0\ \ 0\ \ 0.01\ \ 0]^T$, and the set-point is: $\mathbf{ref} = [1.5\ \ 1\ \ 0.25]$.

Let us first present the outcome of the simulation without noise: Figure 5-5 illustrates the tracking performance alongside the control actions obtained by the MPC. In Figure 5-5(a) we can see that the peak of z is at 1.533 m and its steady-state value is 1.494 m, hence, the overshoot equals 3.9% and the steady-state error 0.006 m (6 mm), consequently the design specifications are met and the error of 6 mm is acceptable when compared to the size of the Matrice 100. The response in x presents an overshoot of 0.8% and its rise time is clearly within the specifications. No overshoot is observed in y and its rise time is also lower than 4 s. Regarding the control actions, during the first seconds, the signals reach their peaks to accelerate the drone and take it to the desired set-point. The effect of the overshoot in z can also be appreciated in $u_{\dot{z}}$, Figure 5-5(d), where it first reaches its maximum (as defined in $\mathbf{u_{max}}$), and later it goes negative until finally converging to 0 m/s to maintain the *drone's* altitude.

The results of the simulation with sensor noise are displayed in Figure 5-6. It can be observed that the MPC is capable of handling the noise to the point that the tracking performance is practically unaffected, de facto, the only noticeable modification is that the peak of x is shifted by approximately half a second. Moreover, the control actions are on the same order of magnitude than those in the noiseless scenario.

5-7 Experimental Results

Relying on the attained simulations, we venture into the real-life implementation where the complete setup and equipment described in Chapter 3, the developed API explained in Section 5-3, and the control topology presented in Section 5-2 are merged. It is important to



(a) Simulated tracking without noise

(b) Simulated u_θ without noise

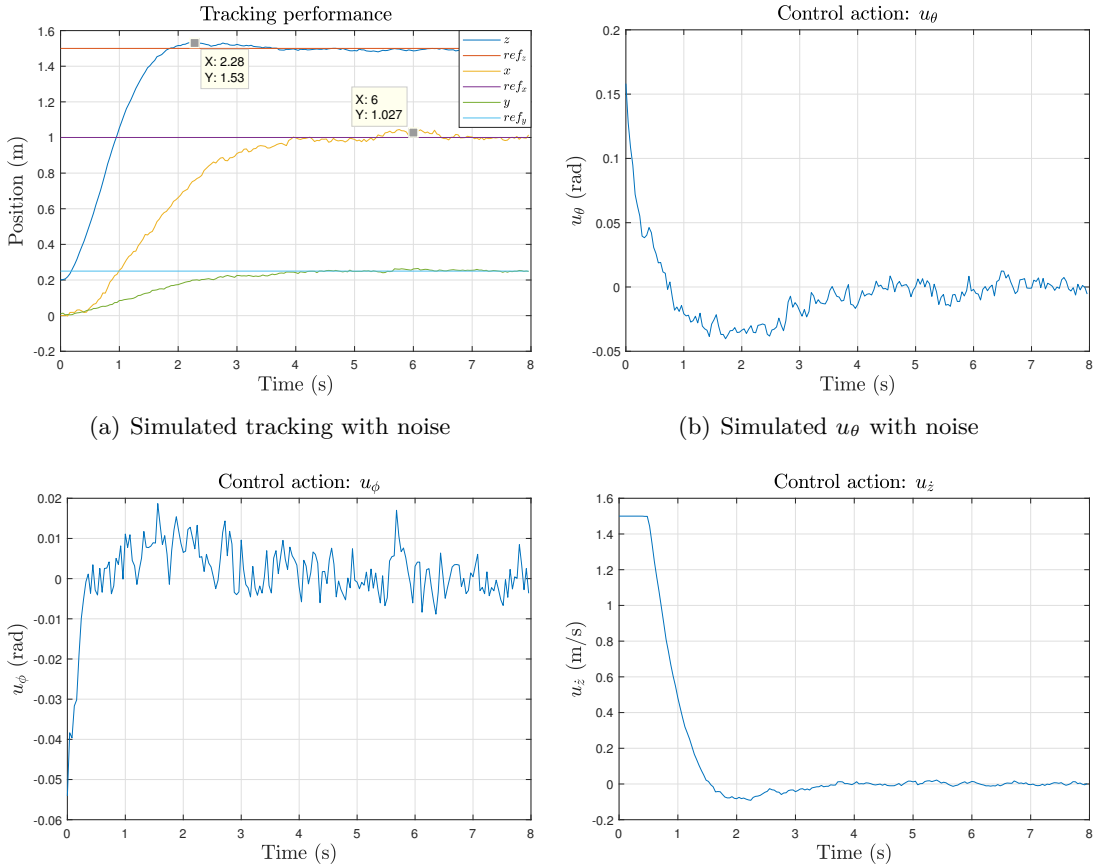
(c) Simulated u_ϕ without noise

(d) Simulated $u_{\dot{z}}$ without noise

Figure 5-5: MPC simulation without noise.

mention that in the real experiments we have $-0.1 \leq u_{\dot{\psi}} \leq 0.1$ (rad) to avoid unexpected fast yaw rates ($\dot{\psi}$) –these limits are set when instantiating the LQ class.

The results herein are acquired by following the procedure presented in Figure 5-7 –notice that the emergency action check, time update, and data logging are intentionally omitted for readability. The sequence of starting ROS and mocap_optitrack first, then ROS-SDK, and finally the main algorithm is mandatory because ROS-SDK is executed in the onboard computer, see Figure 3-9 on page 17, while the others run in the on-ground PC. Although it is actually possible to skip the test of the emergency action, it is strongly recommended, thus, included in the experiment protocol. When the main algorithm is started, it instantiates all the required objects, i.e. two Observer, one LinearMPC, one LQ, and one SubAndPub, by importing all matrices, vectors, dimensions, and sampling time from the text files stored in the *database*; besides, the algorithm sets the number of samples (N) for the experiment and creates the log files to record the measured outputs, state estimates, reference, and control inputs for post-processing. Afterwards, the main program is paused while the user takes off manually or finishes the experiment, when the user is ready, he/she commands the beginning of the experiment. As explained in Section 5-2, we need the yaw angle ψ to be 0 rad (or close to it) for the linear model employed in the MPC to be valid, the first part of the experiment



(a) Simulated tracking with noise

(b) Simulated u_θ with noise

(c) Simulated u_ϕ with noise

(d) Simulated $u_{\dot{z}}$ with noise

Figure 5-6: MPC simulation with sensor noise.

ensures this condition by setting the reference of the MPC to the current position and then regulating ψ until $|\psi| \leq \epsilon = 0.3\pi/180$ rad, as it can never be exactly 0 rad due to sensor noise. Once this condition is met, the reference is changed and the main control loop starts. In each iteration, the measurements and estimates are gotten (from SubAndPub and the observers), then it is checked if $k = 1500$ or 3000 to change the reference, later the control actions are computed (using both the MPC and the LQR as shown in Figure 5-1), following, the next states are estimated and k is increased by 1. Finally, when $k = N$, the drone is commanded to land.

5-7-1 Tracking

Figure 5-8 presents the tracking response of the real-setup. In Figure 5-8(a) we can appreciate that the response in x improves with time, in the sense that after the second change of reference (at 62 s), the settling time decreases; the steady-state error during the final 20 s is of approximately 0.1 m (almost 10% of the drone's diameter). The rise time of the response in y, shown in Figure 5-8(b), increases with time (going from 2.7 s to 9.8 s approx.), yet the steady-state error practically does not change with the reference and is roughly 0.05 m.

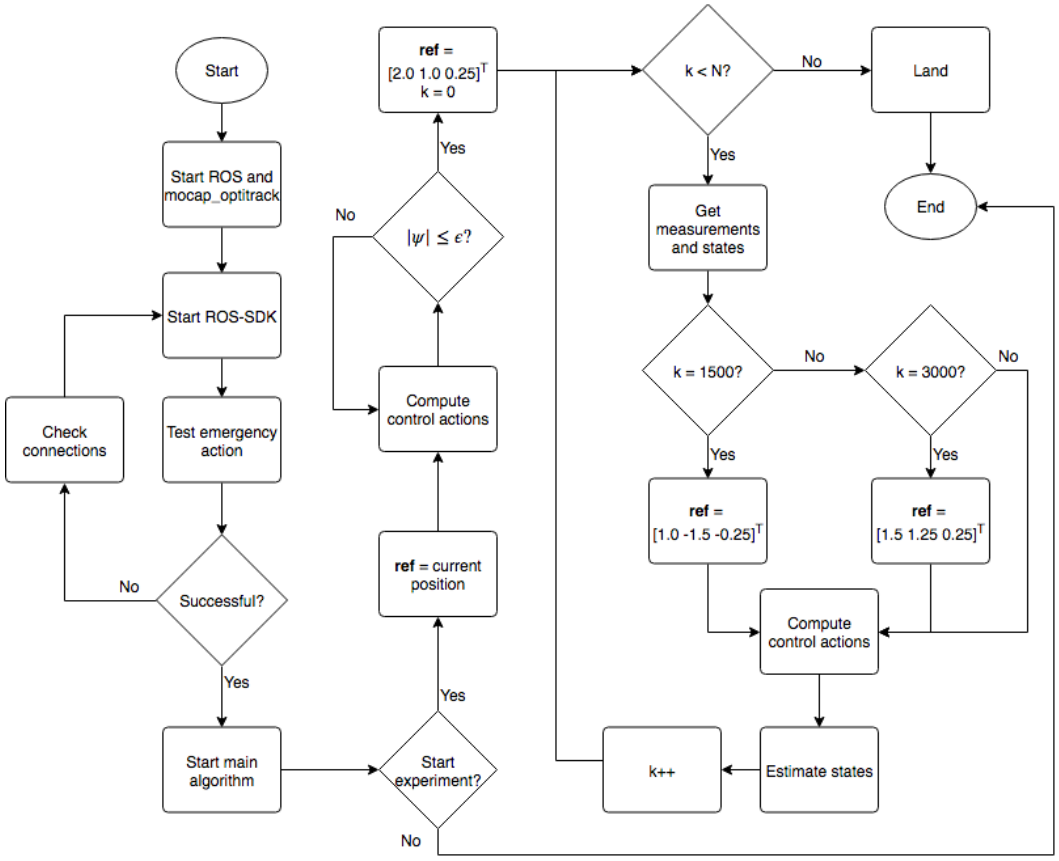
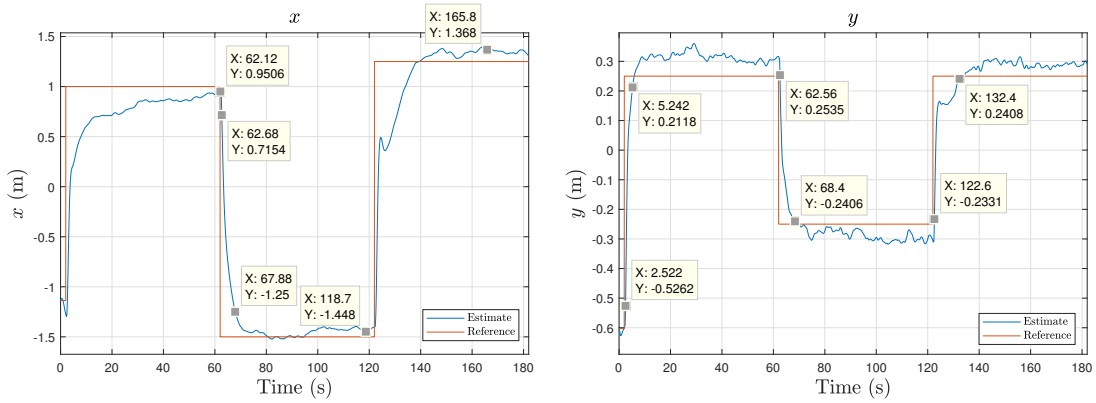


Figure 5-7: Control experiment work flow.

The performance of z is the best of the three as its rise time is about 1 s and its steady-state value is the closest to the reference out of the three tracking outputs and as expected, there is a small overshoot (of approximately 2%). All performances differ from those of the simulations, being z the only one that outperforms them; the reason why there is a larger discrepancy in x and y is the model mismatch explained in Chapter 4, where it was seen that the identified x and y dynamics are not perfect. Nevertheless, the experimental results confirm that the implemented control topology is capable of compensating for such a mismatch and the steady-state errors are small compared to the size of the Matrice 100.

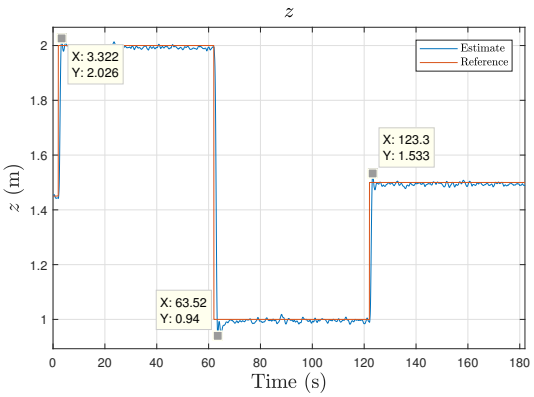
5-7-2 Control Inputs

The control actions computed by the MPC are presented in Figure 5-9(a), Figure 5-9(b), and Figure 5-9(c), whilst the control action of the LQR is plotted in Figure 5-9(d). As expected, the peaks in the first three inputs occur when there is a change in the reference signal to accelerate drone, we can also observe that the control actions are always within the defined limits ($\mathbf{u_{max}} = [0.17453293 \;\; 0.26179939 \;\; 1.5]^T$, $\mathbf{u_{min}} = -\mathbf{u_{max}}$ and $-0.1 \leq u_{\dot{\psi}} \leq 0.1$). With respect to $u_{\dot{\psi}}$, the most prominent peaks also happen when the reference changes, in general this response implies that when the drone is tilted, its yaw angle ψ changes slightly. This effect was not perceived during the identification experiments because the commanded u_ϕ



(a) Experimental x-response

(b) Experimental y-response



(c) Experimental z-response

Figure 5-8: Experimental tracking response.

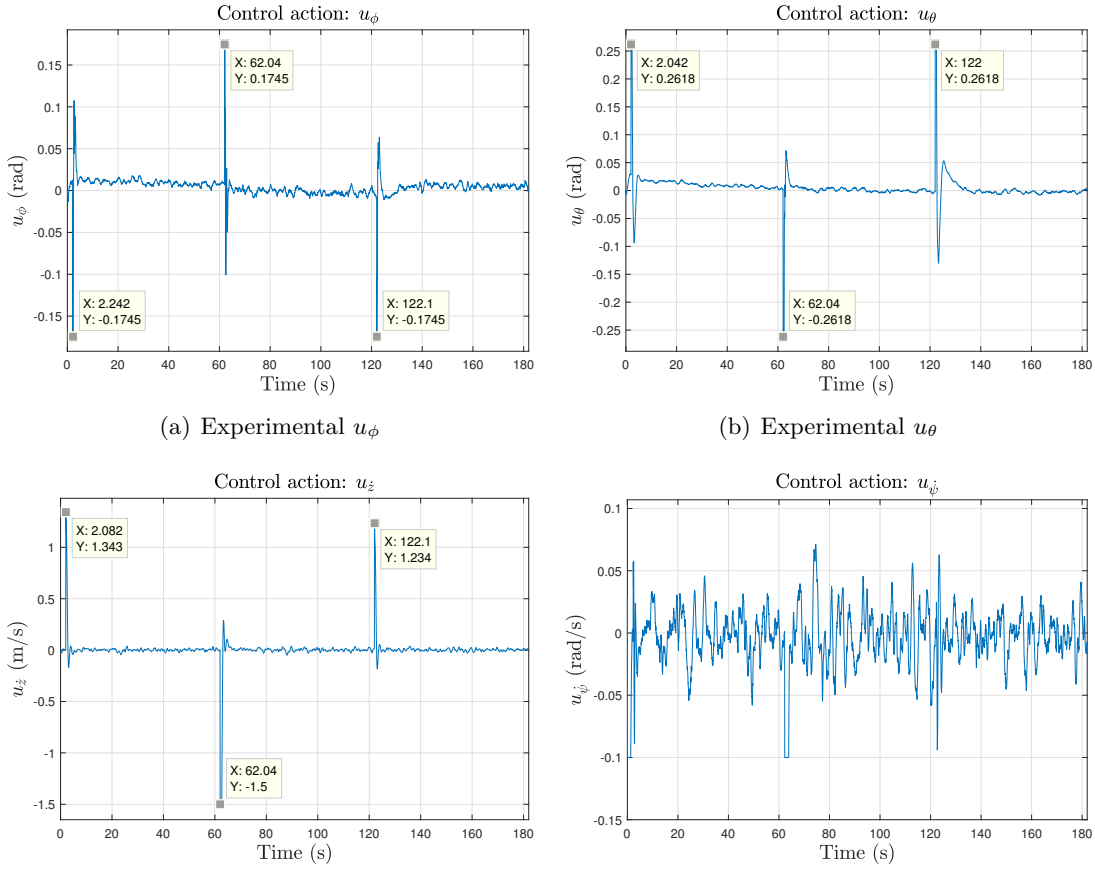
and u_θ had a smother shape, see Figure 4-7 on page 26. Nonetheless, the LQR controller is able to correct ψ along the experiment, guaranteeing the validity of the linear model used in the MPC.

5-7-3 Estimated States vs Measurements

To finalize the experimental results, we compare the output of the two observers with the sensor measurements to assess their performance.

The outcome of the $\hat{\mathbf{r}}$ *observer* is plotted in Figure 5-10. As it can be seen both signals (*Measurements* and *Estimate*) superpose throughout the experiment and the effect of the filter is more prominent in Figure 5-10(b). To be more precise, we contrast the variance of the measurements, $E[\mathbf{y_r}(k)\mathbf{y_r}(k)^T]$, and that of the observer, $E[\hat{\mathbf{y}}_\mathbf{r}(k)\hat{\mathbf{y}}_\mathbf{r}(k)^T]$:

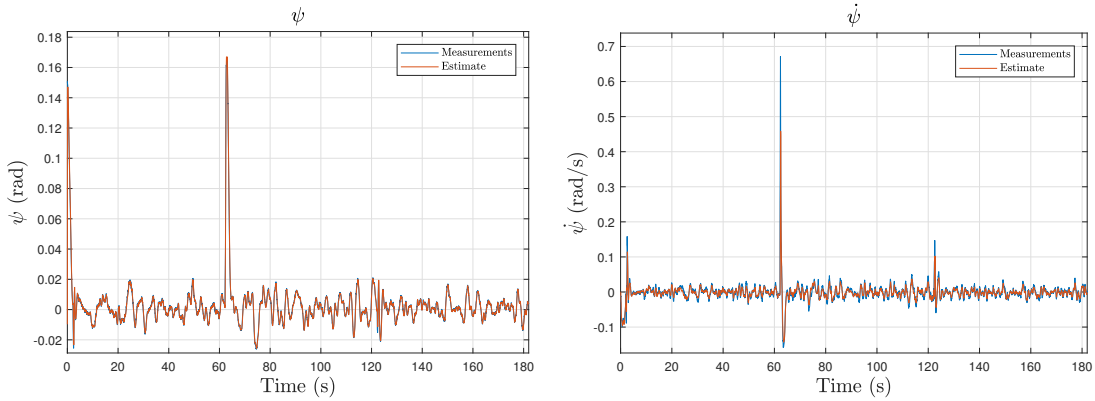
$$\begin{aligned} E[\mathbf{y_r}(k)\mathbf{y_r}(k)^T] &= \begin{bmatrix} 2.9029 \times 10^{-4} & 8.6362 \times 10^{-4} \end{bmatrix}, \\ E[\hat{\mathbf{y}}_\mathbf{r}(k)\hat{\mathbf{y}}_\mathbf{r}(k)^T] &= \begin{bmatrix} 2.8079 \times 10^{-4} & 5.1013 \times 10^{-4} \end{bmatrix}. \end{aligned} \tag{5-24}$$



(a) Experimental u_ϕ (b) Experimental u_θ

(c) Experimental $u_{\dot{z}}$ (d) Experimental $u_{\dot{\psi}}$

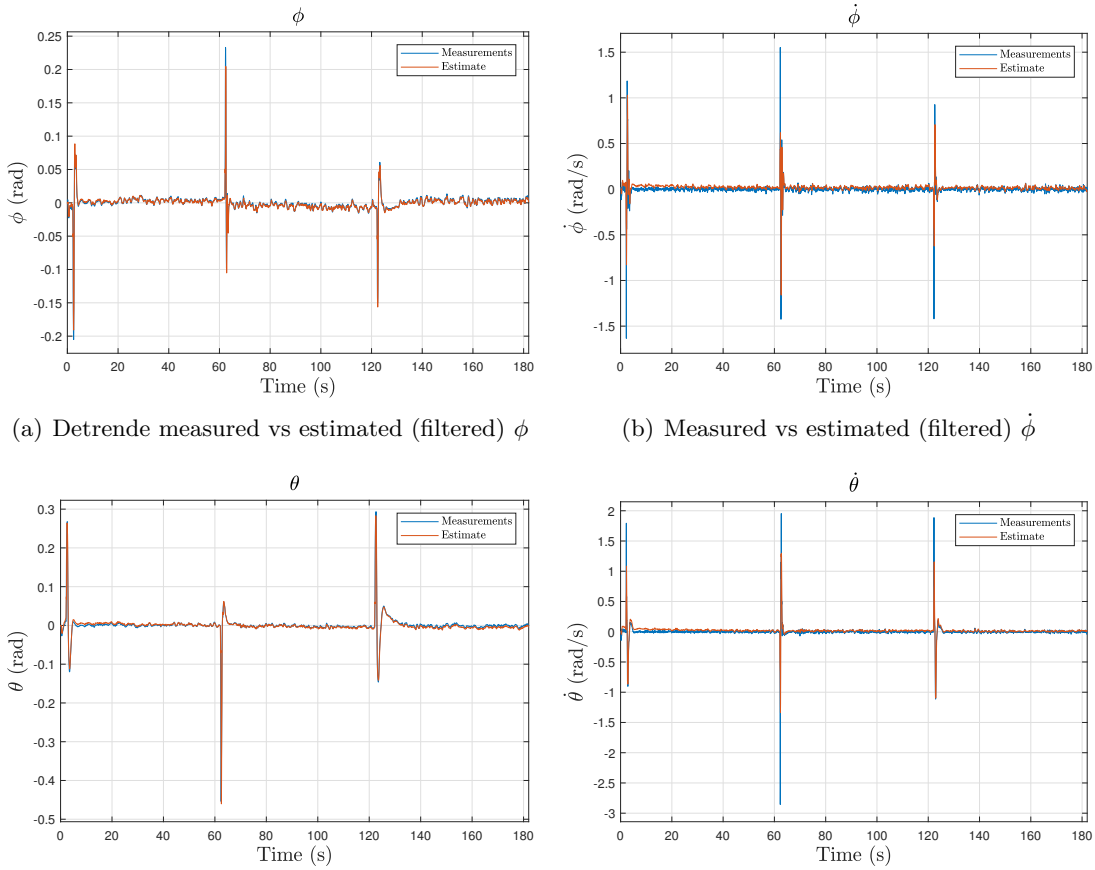
Figure 5-9: Experimental control actions.



(a) Measured vs estimated (filtered) ψ (b) Measured vs estimated (filtered) $\dot{\psi}$

Figure 5-10: $\hat{\mathbf{r}}$ *observer* experimental results.

From Eq. (5-24) we can corroborate that the improvement is larger in the second output. Figure 5-11 presents the first part of the experimental results of the $\hat{\mathbf{q}}$ *observer*. Likewise the



(a) Detrende measured vs estimated (filtered) ϕ

(b) Measured vs estimated (filtered) $\dot{\phi}$

(c) Detrended measured vs estimated (filtered) θ

(d) Measured vs estimated (filtered) $\dot{\theta}$

Figure 5-11: $\hat{\mathbf{q}}$ *observer* experimental results: Part 1.

$\hat{\mathbf{r}}$ *observer*, the filtering is more noticeable in the angular rates. We employ the variance once again to confirm this statement:

$$\begin{aligned}
&E[\mathbf{y_q}(k)\mathbf{y_q}(k)^T] \\
&= [2.2347 \times 10^{-4},\ 7.8641 \times 10^{-3},\ 8.7529 \times 10^{-4},\ 1.5772 \times 10^{-2},\ 0.1635, \\
&9.9671 \times 10^{-3},\ 1.3310,\ 7.9755 \times 10^{-2}], \\
&E[\hat{\mathbf{y}}_\mathbf{q}(k)\hat{\mathbf{y}}_\mathbf{q}(k)^T] \\
&= [1.9628 \times 10^{-4},\ 4.8404 \times 10^{-3},\ 8.1283 \times 10^{-4},\ 1.0144 \times 10^{-2},\ 0.1634, \\
&8.1653 \times 10^{-3},\ 1.3369,\ 7.8724 \times 10^{-2}].
\end{aligned} \tag{5-25}$$

At this point we are interested in the first 4 entries of each vector in Eq. (5-25), where we can see improvement in all of them, but more importantly in the second and forth values, which correspond to the variance of the angular rates.

With the aim of elaborating on pitch angle's performance, θ, we plot the error $\theta - \hat{\theta}$ in Figure 5-12, where a small linear trend can be observed. As mentioned in Section 4-3-3 on page 27, the trend of θ and ϕ varies between experiments, thus, we can expect the identified

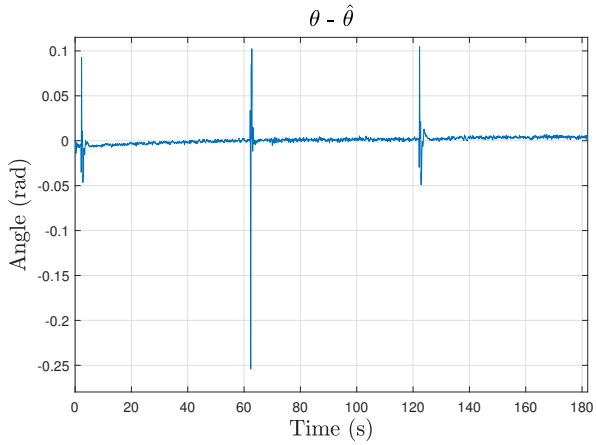
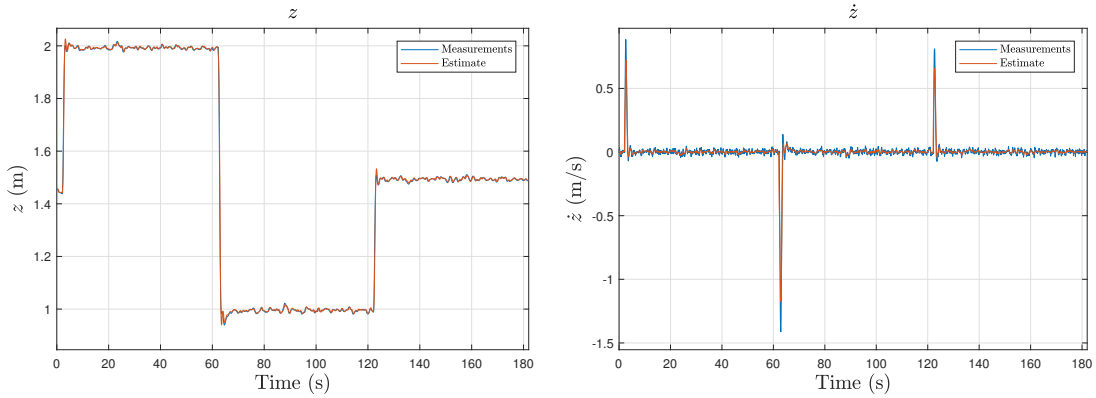


Figure 5-12: Observer error: $\theta - \hat{\theta}$.

one to be slightly off from the real trend. However, the discrepancy is small enough to be able to use the estimated angles. Moreover, the reader should consider that the measured angles were detrended offline in MATLAB using the entire dataset, whereas in the real-time system the trend is set at the commencement of the experiment.

With respect to remaining measurable states, i.e. z, $\dot{z}$, x, and y, the corresponding plots are displayed in Figure 5-13, where it can be seen that the effect of the filter is more prominent in $\dot{z}$ and that the estimated and measured x and y do not overlap completely. In order to clearly observe the discrepancy, we present the error signals of these two outputs in Figure 5-14. There are two possible explanations for this outcome: i) the corresponding penalties in the matrix Q_q, see Eq. (5-7) on page 37, are not tunned well enough, consequently they can be revised awaiting a lower error; and ii) the model mismatch in x and y is not fully counter by the observer. Independently of the imperfectness of the observer, there is a substantial improvement compared to the results obtained in the system identification phase described in Section 4-4 and, more importantly, we have shown that the designed and implemented control topology can cope for this dissimilarity.



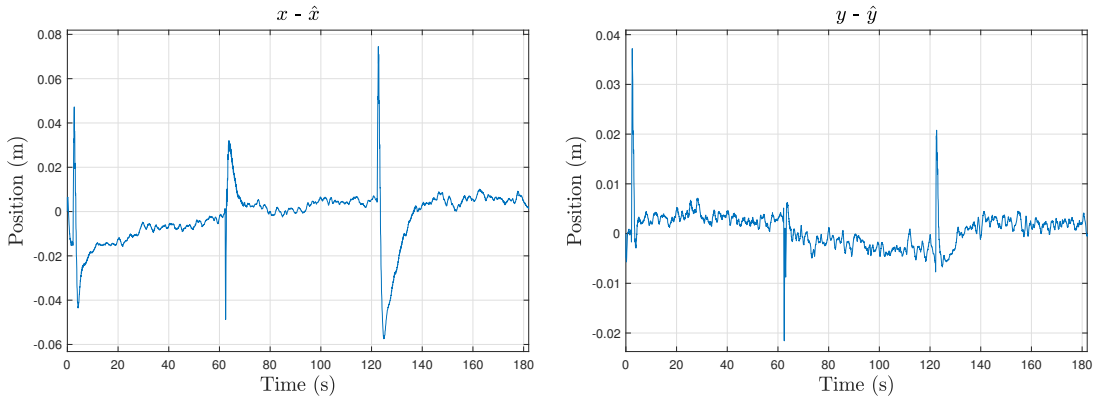
(a) Measured vs estimated (filtered) z

(b) Measured vs estimated (filtered) $\dot{z}$

(c) Measured vs estimated (filtered) x

(d) Measured vs estimated (filtered) y

Figure 5-13: $\hat{\mathbf{q}}$ *observer* experimental results: Part 2.



(a) Observer error: $x - \hat{x}$

(b) Observer error: $y - \hat{y}$

Figure 5-14: Observer error in x and y.

Chapter 6

Conclusions, Recommendations, and Future Work

This final chapter presents the conclusions from this document, gives a few recommendations on how to improve the current status of the attained development platform, and lastly, enunciates and discusses potential applications where the platform can be used.

6-1 Conclusions

To begin with, in Chapter 2 the dynamic model of the Matrice 100 was derived considering the already available low-level controllers –implemented in the drone's autopilot. We opted for separating the dynamics of the yaw angle, ψ, from the rest of the model and, assuming a constant $\psi = \psi_0$, we linearized the equations that describe the horizontal displacement of the UAV.

Chapter 3 showed and explicated the setup and equipment employed in the implementation of the development platform designed in this thesis. The block diagram of the complete setup comprises: the user, the ground station (RC, emergency system, onground PC, router, Motive, and OptiTrack), and the aerial platform (Matrice 100, Guidance, Manifold and the HC-11 attached to it). Moreover, the available sensors were enunciated along with the measurements we gather from them: ϕ, θ, $\dot{\phi}$, $\dot{\theta}$, and $\dot{\psi}$ from the drone's IMU, $\dot{z}$ from the Guidance, and x, y, z, and ψ from OptiTtrack.

In Chapter 4 the system identification routine was explained and used in multiple identification experiments from which the parameters of the derived dynamical models were estimated. Moreover, the outcome of another set of experiments was utilized to validate the identified systems. A trend in the measured roll and pitch angles (ϕ and θ, respectively) was noticed and determined by averaging the trends from all the experiments.

The identified x and y dynamics were found to be reliable for 4 s. From that moment on, the simulation began to diverge from the real measurements but the shape was preserved. The

underlying reasoning for this discrepancy was the accumulation of error when going from the acceleration ($\ddot{x}$ and $\ddot{y}$) to the position. Despite this discrepancy, the trustworthy lapse was long enough to cover the entire MPC's prediction horizon.

With the aim of controlling the flight of the Matrice 100, the attained models were first discretized using a zero-order-hold approximation with sampling time, h, equal to 0.04 s (40 ms), and then used to design and implement the control topology presented in Chapter 5, which included two Kalman observers, one LQR, and one MPC. The purpose of the LQR was to regulate ψ to 0 rad to maintain the validness of the linear model that describes the horizontal displacement of the drone; whilst the MPC controlled the position of the drone according to a given reference.

The MPC achieved its objective by solving every time instant an open-loop optimal control problem whose cost function penalized the difference between the current position and the reference, the control inputs, and the states of the system at the end of the prediction horizon. In addition, the constraints of the problem consisted of the corresponding (10-state and 3-input) dynamic model, the limits of the control actions, and the bounds of the workspace.

An API in ROS C++ was developed to implement the control topology. This software comprises four classes, namely: *LinearMPC*, *Observer*, *LQ*, and *SubAndPub*, and two libraries: *Utilities* and *CVXGEN solver*. All of them but *CVXGEN solver* were coded to be as general as possible, so that they can be utilized not only in the particular control case treated in this thesis but anywhere else they may be required.

When implementing the MPC, it was found that one has diverse options: i) to manually transform the optimization problem into a standard form (if possible) and use an available solver, ii) to develop a fully custom solution that may include the code of the solver, iii) to employ an automatic code generator that produces a solver from a high-level description of the problem. In this thesis, the third choice was selected as it is more suitable for application-oriented projects where the designer changes the formulation of the optimization problem multiple times before arriving to the definite one. The code generator for embedded convex optimization CVXGEN was chosen for this purpose.

During the design phase of the MPC, it was experienced that although it is capable of handling constraints and its cost function can be customized, it is limited by the complexity of the resulting optimization problem. In other words, the more variables or computations are introduced, the more complex the problem is and irrespective of the solver, the amount of variables they can handle is bounded. It was seen that the prediction horizon has the largest effect on this regard as every extra step adds a complete set of states to the optimization vector. In the specific problem treated in this thesis, using a 10-state prediction model and CVXGEN permitted a maximum prediction horizon of 10 samples.

In contrast, the above mentioned limitations were not faced when designing the LQR. However, such an optimal controller does not handle constraints different form the state-space model of the to-be-controlled system.

The performance of the practical implementation of the control topology did not correspond exactly to that of the simulations. More precisely, the steady-state error in x and y was larger in the real-life experiments, but their values are acceptable compared to the dimensions of the Matrice 100. In addition, there was a discrepancy between the estimated x and y and

the corresponding measurements due to model mismatch and/or not accurate tuning of the observer.

In the two observers implemented in this project, the largest noise reduction occurred in the angular rates and the vertical velocity, which may indicate that their weights are better tuned than the others, opening the room for improvement.

As a final conclusion, the main aim of this project was achieved: a reliable development platform for indoor flight control of a UAV (Matrice 100) was designed and successfully tested using an LQR to regulate the yaw angle and an MPC to control the position of the quadrotor. The upcoming users of this platform will be able to readily utilize the API to implement their own flight control algorithms in the Matrice 100 utilized in this project – or in a different quadrotor from DJI that supports DJI's ROS-SDK, by executing first the presented identification routines to estimate its parameters and, then, tuning the controllers and observers.

6-2 Recommendations

As explained in Chapter 3, when the user pushes the emergency button, the Arduino detects a rising edge at its interrupt pin, triggering the emergency action. Such a method has been proven to work, however, it is suggested to change it to continuously send a command while the pin's voltage is HIGH and stop sending it when it is LOW. The voltage can be controlled by the button as follows: when released the voltage is HIGH and LOW when it is pushed. The mentioned command would function as a heartbeat signal, so when the Manifold and/or the on-ground PC does not receive it, it/they would activate the emergency action. By doing so, the system would be more robust to (wired or wireless) connection failures or power shutdowns, for instance.

It was shown that the attained controlled system is reliable, but there is steady-state error in the tracking outputs, one method to eliminate it is to integrate the tracking error and add it to the MPC's output, similarly to an LQI, which can be achieved by tunning the integrator gains manually in simulation and implementing them by means of the LQ class from the API, see the class documentation for instructions on how to use the integral action.

The other concern on the experimental performance is the discrepancy between the estimated and measured x and y, in order to improve it, the corresponding weights in the observer can be fine-tuned. Given that R_q was set according to the variance of the measurements while the drone was hovering, it is more likely that Q_q is the weighting matrix that requires fine tuning. The other option is to revise the identified model, which can be enhanced via closed-loop identification as the current MPC would allow the user to command rich (position) reference signals while ensuring the UAV will not collide against the windows or the net.

In Chapter 5 it was mentioned that the user needs to take off manually before starting the control experiment. Nonetheless, two autonomous methods were tested: i) calling the takeoff command from DJI's ROS-SDK and ii) arming the rotors and using the explained control topology to reach a given altitude. With the first method, the UAV did take off, but the SDK gave a constant thrust command to the Matrice 100 for about 10 seconds during which other inputs were ignored by the autopilot, hence, the drone drifted while gaining height, which is not desirable as it may collide with the windows or the net around the workspace. The second

approach also lifted up the drone, the inconvenient was that it depended on the autopilot's vertical velocity controller which, at the same time, depends on the vertical velocity estimate from the Guidance, which is not useful before the drone reaches an altitude of about 50 cm. Thus, the drone also drifted while taking off, even more than with the SDK's command. On top of it, it was seen that when the rotors are armed from a custom algorithm, a subsequent landing instruction shuts them down, i.e. the drone falls rather than landing, hence, this procedure was discarded for safety reasons. The suggestion is to debug DJI's SDK to try to find the cause of the latter problem, if it is successfully fixed, the low-level thrust controller may be identified (alike to the other low-level controllers) and used for takeoff, instead of utilizing the vertical velocity controller.

One more bug in DJI's SDK was experienced: when landing is commanded through the Onboard-SDK, it works every two calls. In other words, when we instruct the drone to land from the Onboard-SDK for the first time, it does it, the next one it does not, the third it does, and so on. This bug was reported by the author of this thesis on DJI's GitHub forum, but no answer has been given yet (as of September 17, 2018). Thus, it is recommended to track the landing call as far as possible to try to determine the source of the bug and correct it. However, the problem may be in the autopilot to which only DJI has access.

There are a few deprecated functions in the API, which can be removed. More importantly, a better object-oriented structure can be designed to rearrange the existing classes. For example, there can be a class called *StateSpace* from which LinearMPC and LQ inherit, so that the A, B, and C matrices would not have to be redefined in each class. Such an API would be easier to maintain and use.

Thus far, the control-related calculations have been executed in the on-ground PC. However, the API has been developed in ROS C++ with the aim of making it suitable for embedded systems, hence, for onboard implementation. It is suggested to test the performance of the controlled system when the main algorithm is run in the onboard computer. Even though porting the code to the Manifold is a matter of copying the existing files into its ROS workspace and compiling them, the emergency action check in the main algorithm would have to be revised because, onboard, the emergency command is received through the HC-11, whose USB port is read by the edited ROS-SDK, as explained in Chapter 4. Thus, an inter-process communication strategy should be used to forward the emergency command from the ROS-SDK to the main algorithm, two possibilities to achieve it are: User Datagram Protocol (UDP) [43] or pipes [44].

6-3 Future Work

Before exploring potential applications where the current development platform can be employed, designing a non-linear controller that includes a time varying yaw angle is appealing. Such a controller would permit the user to execute more complex tasks such as tracking a given position reference while altering the heading of the drone. A possible approach is the so-called Non-linear Model Predictive Control (NMPC) which can be implemented using ACADO [45], a C-code generator *similar* to CVXGEN, but whose main difference is that the MPC problem (cost function and constraints) can only be entered in continuous time.

In Chapter 1, a few works in flight formation found in the literature were briefly mentioned,

there the researchers tested their algorithms in simulations, but not in real aircraft. The attained development platform can be used for real-life experiments. In fact, in flight formation more than one aircraft is present, thus, at least one more Matrice 100 (or any DJI drone compatible with DJI's ROS-SDK) should be identified following the identification routines explained in this document and, then, the corresponding controllers and observers should be tuned. Once the new drones are ready, one may tune the flight formation algorithms in simulation and, afterwards, test them with the actual drones. Clearly, the API should be extended for this purpose and, more importantly, the user should be aware of the size of the available workspace before executing the experiments as it limits the plausible maneuvers, formations, and number of aircraft that can fly at the same time.

Obstacle avoidance ([15, 46, 47]) utilizing OptiTrack as sensor to detect where the objects are can be integrated to the current platform. Similarly, it can be extended with localization and mapping ([48, 49]) by adding extra sensors, e.g. onboard cameras or LiDAR, and their measurements may be fused with those from OptiTrack to explore sensor fusion algorithms. Regardless of the final integration, obstacle avoidance, alike to flight formation, consists in developing a motion planer that generates the reference signal for the MPC, thus, it can be easily tested with the current platform.

The last potential future work we would like to discuss is emulation of *spacecraft rendezvous and docking*, where rendezvous refers to taking an actuated spacecraft (namely the *chaser*) to the proximity of a satellite (known as *target*), while docking is pairing both of them. This spacecraft maneuver is essential in missions such as: transport of a payload from the Earth to the International Space Station, capture and recovery of satellites, spacecraft formation, capture of samples (as in the Mars Sample Return (MSR) mission), etc. [50]; therefore, researchers are investigating control methods that allow the complete rendezvous and docking to be executed autonomously –nowadays it is semi-autonomous as the chaser is piloted at specific moments either by the crew or from a ground station. Recently, they have used MPCs (in simulations) for this purpose, because of MPC's capability of handling constraints within its formulation and the well-known relative dynamics that describe the movement of the chaser w.r.t. the target [50, 51, 52, 53, 54, 55].

Now that most of the research in the above topic is limited to simulations, it would be useful to have a platform where to test the proposed control strategies. However, one should understand that using actual spacecraft is laborious, expensive and dangerous. Thus, we propose to identify relevant features that may be scaled down to be tested with quadrotors, such as the Matrice 100 employed in this thesis. By doing so, implementation challenges (at least software-wise) could be detected and addressed with the aim of, eventually, being able to scale the results up for real implementation on spacecraft. With such an objective in mind, we present a brief literature review next and determine a few aspects that could be emulated from it.

Hartley et al. [51] divided the rendezvous and docking mission into multiple phases and controlled them with different MPCs, whose formulation depended on the most suitable prediction model or the relevant constraints; Cairano et al. [50] designed an MPC for spacecraft formation that included collision avoidance, which can be added to the rendezvous and docking maneuver; and particularly when docking, satisfying the line of sight (LOS) constraints of the target's docking sensor, e.g. LiDAR, is essential to complete the mission, examples on how to model these constraints can be found in [51, 53, 56]. From these previous works,

we could emulate aspects such as: relative movement of a chaser w.r.t. a target including collision avoidance between them (or also external objects), having a mission with multiple phases where diverse MPCs need to be switched, or maintaining a chaser aircraft within the LOS of a target.

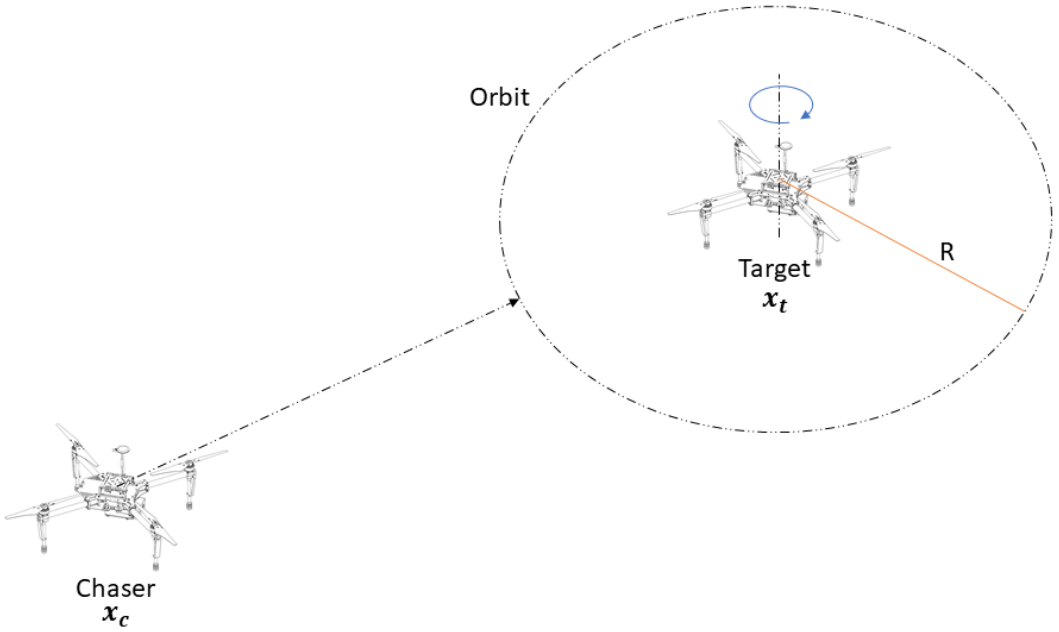


Figure 6-1: Two-Quadrotor emulation of spacecraft rendezvous and docking, where $\mathbf{x_c}$ denotes the position of the chaser and $\mathbf{x_t}$ the target's.

An initial proposal is presented in Figure 6-1, where $\mathbf{x_c}$ is the chaser's position, $\mathbf{x_t}$ is the target's, and R is the radius of a virtual orbit centered on the target. The mission is as follows: the two aircraft (target and chaser) take off from different points (not necessarily at the same time); once they reach a pre-defined altitude, the target starts spinning at a fixed position, i.e. its yaw angle ψ is constantly changed while maintaining its position; meanwhile, the chaser moves towards the target until reaching a distance R from it, resembling the rendezvous maneuver. At that point, we propose two options: i) the chaser hovers until the target's nose faces it and then begins to orbit around the target while staying in front of its nose, or ii) the chaser starts orbiting around the target at low or high speed until it is faced by the target and synchronizes its speed with that of the target to keep facing it. Any of these two approaches would require a different prediction model from the one used during the first part, as it would have to include the relative position/velocity of the chaser with respect to the target's heading and yaw rate. Then, the second phase of the mission would somewhat represent the docking procedure.

Bibliography

[1] DJI, "DJI MATRICE 100 User Manual." Available at https://dl.djicdn.com/downloads/m100/M100_User_Manual_EN.pdf (2017/11/07).

[2] DJI, "Manifold User Manual." Available at https://dl.djicdn.com/downloads/manifold/20170918/Manifold_User_Manual_v1.2_EN.pdf (2018/05/30).

[3] B&H Photo Video Pro Audio, "DJI Guidance CP.VL.00103 B&H Photo Video." Available at https://www.bhphotovideo.com/c/product/1185290-REG/dji_cp_vl_000003_guidance.html (2018/05/30).

[4] S. N. Ghazbi, Y. Aghli, M. Alimohammadi, and A. A. A. and, "Quadrotors unmanned aerial vehicles: A review," *International Journal on Smart Sensing and Intelligent Systems*, vol. 9, no. 1178-5608, pp. 309–333, 2016.

[5] R. E. Mahony, V. Kumar, and P. Corke, "Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor.," *IEEE Robot. Automat. Mag.*, vol. 19, no. 3, pp. 20–32, 2012.

[6] J. Navia, I. Mondragon, D. Patino, and J. Colorado, "Multispectral mapping in agriculture: Terrain mosaic using an autonomous quadcopter uav," in *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 1351–1358, June 2016.

[7] G. Hoffmann, S. Waslander, and C. Tomlin, ch. Distributed Cooperative Search Using Information - Theoretic Costs for Particle Filters, with Quadrotor Applications. Guidance, Navigation, and Control and Co-located Conferences, American Institute of Aeronautics and Astronautics, Aug 2006.

[8] S. Lubell, "The master of drones turns flying machines into performers." Available at https://www.wired.com/2016/03/master-drones-turns-flying-machines-performers/ (2017/10/25).

[9] S. Lubell, "Drones rising as valuable tool in commercial film industry." Available at https://www.bostonglobe.com/business/2017/

06/23/drones-rising-valuable-tool-commercial-film-industry/mbvWUH4Ydc5rkdHrMqxgjN/story.html (2017/10/25).

[10] D. Baocang, *Modern Predictive Control*. Taylor & Francis, 2009.

[11] Í. B. Viana, D. A. dos Santos, and L. C. S. Góes, "Formation control of multirotor aerial vehicles using decentralized mpc," *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, vol. 40, p. 306, May 2018.

[12] X. Wang, V. Yadav, and S. N. Balakrishnan, "Cooperative uav formation flying with obstacle/collision avoidance," *IEEE Transactions on Control Systems Technology*, vol. 15, pp. 672–679, July 2007.

[13] A. T. Hafez, A. J. Marasco, S. N. Givigi, M. Iskandarani, S. Yousefi, and C. A. Rabbath, "Solving multi-uav dynamic encirclement via model predictive control," *IEEE Transactions on Control Systems Technology*, vol. 23, pp. 2251–2265, Nov 2015.

[14] T. Nägeli, J. Alonso-Mora, A. Domahidi, D. Rus, and O. Hilliges, "Real-time motion planning for aerial videography with dynamic obstacle avoidance and viewpoint optimization," *IEEE Robotics and Automation Letters*, vol. 2, pp. 1696–1703, July 2017.

[15] N. Potdar, "Online trajectory planning and control of a mav payload system in dynamic environments: A non-linear model predictive control approach," Master's thesis, Delft, The Netherlands, 2018.

[16] I. Sa, M. Kamel, R. Khanna, M. Popovic, and R. Nieto, J. Siegwart, "Dynamic System Identification, and Control for a cost effective open-source VTOL MAV," Jan. 2017.

[17] M. P. Tully Foote, "REP 103 – Standard Units of Measure and Coordinate Conventions (ROS.org)." Available at http://www.ros.org/reps/rep-0103.html (2018/05/24).

[18] B. Hu, "dji_sdk - ROS Wiki." Available at http://wiki.ros.org/dji_sdk (2018/05/24).

[19] F. Dai, K. Wang, and P. Lin, "A stereo camera-equipped quadrotor platform for vision based nonlinear control," in *2016 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 1864–1869, Dec 2016.

[20] D. Bohdanov, "Quadrotor uav control for vision-based moving target tracking task," Master's thesis, Toronto, Canada, 2012.

[21] P. Bouffard, "On-board Model Predictive Control of a Quadrotor Helicopter: Design, Implementation, and Experiments," tech. rep., EECS Dept., UC Berkeley, 12 2012.

[22] H. Bolandi, M. Rezaei, R. Mohsenipour, H. Nemati, and S. M. Smailzadeh, "Attitude control of a quadrotor with optimized pid controller," *Intelligent Control and Automation*, vol. 04, no. 03, pp. 335–342, 2013.

[23] T. Komura, "Geometric transformations." Available at http://www.inf.ed.ac.uk/teaching/courses/cg/2017/pdf/cg17-l5.pdf, 2017.

[24] K. Klausen, T. I. Fossen, and T. A. Johansen, "Nonlinear control with swing damping of a multirotor uav with suspended load," *Journal of Intelligent & Robotic Systems*, vol. 88, pp. 379–394, Dec 2017.

[25] DJI, "Buy Matrice 100 TB47D Battery - DJI Store." Available at https://store.dji.com/product/matrice-100-tb47d-battery (2018/05/30).

[26] DJI, "Matrice 100: The quadcopter for developers - DJI." Available at https://www.dji.com/matrice100 (2017/11/07).

[27] J. Rehor and V. Havlena, "Grey-box model identification – control relevant approach," *IFAC Proceedings Volumes*, vol. 43, no. 10, pp. 117 – 122, 2010. 10th IFAC Workshop on the Adaptation and Learning in Control and Signal Processing.

[28] MathWorks Benelux, "Estimate Parameters from Measured Data." Available at https://nl.mathworks.com/help/sldo/gs/estimate-parameters-from-measured-data-using-the-gui.html (2018-06-10).

[29] MathWorks Benelux, "Solve Non-linear Curve Fitting (Data-fitting) Problems in Least-squares Sense." Available at https://nl.mathworks.com/help/sldo/gs/estimate-parameters-from-measured-data-using-the-gui.html (2018-06-10).

[30] M. Verhaegen and V. Verdult, *Filtering and System Identification: A Least Squares Approach*. New York, NY, USA: Cambridge University Press, 1st ed., 2007.

[31] MathWorks Benelux, "Smooth response data - MATLAB smooth - MathWorks Benelux." Available at https://nl.mathworks.com/help/curvefit/smooth.html (2018-06-18).

[32] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. San Diego, CA, USA: California Technical Publishing, 1997.

[33] K. J. Åström and B. Wittenmark, *Computer-controlled Systems (3rd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[34] MathWorks Benelux, "Linear-Quadratic-Integrator control - MATLAB lqi - MathWorks Benelux." Available at https://nl.mathworks.com/help/control/ref/lqi.html (2018-08-26).

[35] MathWorks Benelux, "Solve discrete-time algebraic Riccati equations (DAREs) - MATLAB dare - MathWorks Benelux." Available at https://nl.mathworks.com/help/control/ref/dare.html (2018-08-23).

[36] MathWorks Benelux, "Linear-Quadratic Regulator (LQR) Design - MATLAB lqr - MathWorks Benelux." Available at https://nl.mathworks.com/help/control/ref/lqr.html (2018-08-23).

[37] J. H. Lee, "Model predictive control and dynamic programming," in *2011 11th International Conference on Control, Automation and Systems*, pp. 1807–1809, Oct 2011.

[38] J. Mattingley, Y. Wang, and S. Boyd, "Receding horizon control," *IEEE Control Systems Magazine*, vol. 31, pp. 52–65, June 2011.

[39] M. Juelsgaard, "Implementing MPC using CVX." Available at http://kom.aau.dk/~mju/downloads/otherDocuments/MPCusingCVX.pdf (2018-08-23).

[40] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004.

[41] J. Mattingley and S. Boyd, "Cvxgen: a code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, pp. 1–27, Mar 2012.

[42] M. Grant, S. Boyd, and Y. Ye, *Disciplined Convex Programming*, pp. 155–210. Boston, MA: Springer US, 2006.

[43] Sascha Nitsch Unternehmensberatung UG, "Linux Howtos: C/C++ -> Sockets Tutorials." Available at http://www.linuxhowtos.org/C_C++/socket.htm (2018-09-07).

[44] die.net, "pipe(2): create pipe - Linux man page." Available at https://linux.die.net/man/2/pipe (2018-09-07).

[45] B. Houska, H. J. Ferreau, and M. Diehl, "Acado toolkit—an open-source framework for automatic control and dynamic optimization," *Optimal Control Applications and Methods*, vol. 32, no. 3, pp. 298–312.

[46] K. Chang, Y. Xia, K. Huang, and D. Ma, "Obstacle avoidance and active disturbance rejection control for a quadrotor," *Neurocomputing*, vol. 190, pp. 60 – 69, 2016.

[47] K. Motonaka, K. Watanabe, and S. Maeyama, "Obstacle avoidance for autonomous locomotion of a quadrotor using an hpf," vol. 2, no. 2, pp. 110–116, 2016. 110.

[48] Y. Wei and Z. Lin, "Vision-based tracking by a quadrotor on ros**this work was supported in part by the u.s. army research office under grant w911nf1510275.," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 11447 – 11452, 2017. 20th IFAC World Congress.

[49] M. Faessler, F. Fontana, C. Forster, E. Mueggler, M. Pizzoli, and D. Scaramuzza, "Autonomous, vision-based flight and live dense 3d mapping with a quadrotor micro aerial vehicle," *Journal of Field Robotics*, vol. 33, no. 4, pp. 431–450.

[50] S. di Cairano, H. Park, and I. Kolmanovsky, "Model predictive control approach for guidance of spacecraft rendezvous and proximity maneuvering," *International Journal of Robust and Nonlinear Control*, vol. 22, no. 12, pp. 1398–1427, 2012.

[51] E. N. Hartley, P. A. Trodden, A. G. Richards, and J. M. Maciejowski, "Model predictive control system design and implementation for spacecraft rendezvous," *Control Engineering Practice*, vol. 20, no. 7, pp. 695 – 713, 2012.

[52] L. Breger and J. How, ch. J2-Modified GVE-Based MPC for Formation Flying Spacecraft. Guidance, Navigation, and Control and Co-located Conferences, American Institute of Aeronautics and Astronautics, Aug 2005.

[53] F. Gavilan, R. Vazquez, and E. F. Camacho, "Chance-constrained model predictive control for spacecraft rendezvous with disturbance estimation," *Control Engineering Practice*, vol. 20, no. 2, pp. 111 – 122, 2012.

[54] D. Morgan, S.-J. Chung, and F. Y. Hadaegh, "Model predictive control of swarms of spacecraft using sequential convex programming," *Journal of Guidance, Control, and Dynamics*, vol. 37, pp. 1725–1740, Apr 2014.

[55] U. Kalabić, R. Gupta, S. D. Cairano, A. Bloch, and I. Kolmanovsky, "Constrained spacecraft attitude control on so(3) using reference governors and nonlinear model predictive control," in *2014 American Control Conference*, pp. 5586–5593, June 2014.

[56] L. S. Breger and J. P. How, "Safe trajectories for autonomous rendezvous of spacecraft," *Journal of Guidance, Control, and Dynamics*, vol. 31, pp. 1478–1489, Sep 2008.

Glossary

List of Acronyms

MPC	Model Predictive Control
ENU	East-North-Up
FLU	forward-left-up
MSR	Mars Sample Return
LP	Linear Programming
QP	Quadratic Programming
LOS	line of sight
DARE	discrete algebraic Riccati equation
SDK	Software Development Kit
I/O	input-output
RF	radio frequency
IMU	inertial measurement unit
RC	remote control
VAF	Variance accounted for
LQR	Linear-Quadratic Regulator
DARE	discrete-time algebraic Riccati equation
RHC	Receding Horizon Control
KKT	Karush–Kuhn–Tucker
API	application programming interface

LQI	Linear-Quadratic-Integral control
NMPC	Non-linear Model Predictive Control
VTOL	Vertical Takeoff and Landing
UDP	User Datagram Protocol

List of Symbols

ϕ	Roll
ψ	Yaw
θ	Pitch