Model-based Design of a Line-tracking Algorithm for a Low-cost Mini Drone through Vision-based Control

BY

PAOLO CEPPI
B.S. in Mechanical engineering, Politecnico di Torino, Turin, Italy, 2018

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer engineering in the
Graduate College of the
University of Illinois at Chicago, 2020

Chicago, Illinois

Defense Committee:

Igor Paprotny, Chair and Advisor
Milos Zefran
Alessandro Rizzo, Politecnico di Torino

To my Uncle Damiano,

Happiness is to love an uncle like a big brother.

ACKNOWLEDGMENTS

I dedicate this space to the people who, with their support, have helped me in this engaging research work through which I applied the knowledge acquired during my academic studies.

Firstly and foremost, I would like to thank my UIC advisor, Dr. Igor Paprotny, who, in these five months of work, has been able to continually guide me with tools, improvements, useful tips during the research, and drafting of the paper. Thanks to him, I have improved my analysis and problem-solving skills, which will be useful in my future career.

I am grateful to my advisor in Politecnico di Torino, Dr. Alessandro Rizzo, who allowed me to undertake this research work at UIC and who has always been available to help me with practical suggestions. Thanks to him, I acquired a working method that I will surely replicate in the future.

I sincerely thank my parents for their constant support, their patience, and their motivation during this challenging period and for allowing me to complete academic studies with serenity, focusing on my goals.

I want to express my gratitude to my close relatives, and my best friends, who always made me feel optimistic and encouraged me not to give up.

A special thanks to my mates of Micromechatronic Systems Lab for welcoming me in their united and tight-knit group and for supporting me during these months.

Finally, I would like to dedicate this milestone made of sacrifices to myself. I hope it could be the beginning of a long and brilliant professional career, made of determination, enthusiasm, and desire for learning.

PC

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF ABBREVIATIONS

UAV	Unmanned Aerial Vehicle
RPA	Remote Piloted Aircraft
DOF	Degrees of Freedom
RPY	Roll, Pitch, and Yaw
MIL	Model in the loop
HIL	Hardware in the loop
MMA	Motor Mixing Algorithm
PWM	Pulse Width Modulation
SONAR	Sound Navigation and Ranging
IMU	Inertial Measurement Unit
DMP	Digital Motion Processor
RNDIS	Remote Network Driver Interface Specification
PAN	Personal Area network
FCS	Flight Control System
PID	Proportional-Integral-Derivative
RGB	Red Green and Blue (Color space)
SHT	Standard Hough Transform
UI	User Interface

SUMMARY

This Thesis research project aims to design a Line-tracking algorithm for a low-cost mini drone through Vision-based control with image processing techniques. The design process is the application of the principles of Model-based software design, which is a modern technique to design control systems, based on the development of a model of the plant and the controller with enough detail to have a realistic representation of its behavior to accomplish the specifications. The designed model is tested in a simulation environment (Model-in-the-loop phase). Then, if it satisfies the requirements, it is tested in real-time, deploying the algorithm on the Hardware to evaluate if its performances are still acceptable or if it requires to be updated.

A significant advantage that characterizes this technique is the auto-code generation, which allows us to automatically translate the blocks of the model built through Simulink into a C-code executable by the hardware, instead of writing it manually.

This research project is adapted from a competition organized by Mathworks, which aims to make a drone follow a line of a specific color and land at the end of it on a circle. The task should be accomplished in as little time as possible but at the same time remaining stable and following the path as precisely as possible (within the low-cost limits of the mini drone used). The environment used to design and develop the control system is MATLAB, with Simulink and their add-on toolboxes like Aerospace blockset, image processing, computer vision, and Hardware support package for Parrot mini drone, which is the specific company that made the drone model of this project.

Firstly, the preliminary goal is the accomplishment of the stabilization of flight maneuvers through a suitable control system architecture and PID controllers tuning.

Then, the Flight Control System design proceeds with Image processing and Path planning subsystems design. The line-tracking algorithm implementations developed are two. The first one is based on the analysis of the pixels of the image acquired from the downward-facing camera and elaboration through image

processing techniques like color thresholding and edge detection. The path planning logic was implemented through Stateflow, which is an add-on tool of Simulink, useful for State machines design. This first designed control system also has another simplified version, useful because computationally lighter on the hardware compared to the first standard version. The second algorithm, instead, is realized by using user-defined functions, like thresholding operation for noise removal in the binary image, or like the function that searches and detects the path and the line angles, and by some other already existing functions provided by the computer vision toolbox.

Finally, their performances were both tested on the hardware and then analyzed and compared. The validation phase was discussed, commenting on their limits, and highlighting other issues encountered, not previously noticed within the simulation 3D environment during the Model-in-the-loop test.

CHAPTER 1

INTRODUCTION

1.1. Brief History of UAV and Advantages

The Leonardo da Vinci's "aerial screw" project in the Renaissance period (Figure 1.1) would have been only a progenitor of what has been a continuous technological evolution until now dictated by the greatest of man's desires: flying.

Today, not only can this action be carried out standardly, but it is also possible to control any aircraft while sitting comfortably.

Figure 1.1: "aerial screw" prototype as designed by Leonardo da Vinci [1]

The "main character" that embodies this new concept of flight is the drone, which is a remotely piloted aircraft, or more generally a flying device, which is not piloted on board.

As often happens in aviation history, the technique and quality of the aircraft have developed first in the war field during the two world wars, and the same goes for drones. The progress of technology pushed more increasingly during the Cold War, and it allowed us to reduce their sizes progressively up to what we know.

Because of the growing commercialization and due to the more excellent usability of the vehicle, the FAA (Federal Aviation Administration) has decided to regulate multi-copters based on their features and to overall performance: weight, range, and service ceiling are just some of the parameters taken into consideration [2].

Without a driver, the drone can perform certain types of missions between two points through an on-board computer or a remote-control system, and for this reason, it is also called RPA (Remote Piloted Aircraft). They are also known with other acronyms, many of which are of Anglo-Saxon derivation: in addition to RPA (Remotely piloted aircraft) can also be referred to as RPV (Remotely piloted vehicle), UVS (Unmanned vehicle system), ROA (Remotely operated aircraft), UAV (Unmanned aerial vehicle), etc.

Their use is now consolidated in the military field. It is increasing for civil applications, for example, in fire counteraction and critical operations, for reconnaissance, for remote sensing, research, and, in general, in all cases where these devices can permit the accomplishment of standardized, time-consuming, or dangerous missions. They are often carried out with lower economic costs than traditional air vehicles and do not encounter so many ethical issues as it could be expected a few years ago because of their reliability. Depending on the range, they can be equipped for long distances with a video camera that allows them to follow in real-time the movements in first-person: this class is called FPV (First-person view) [3].

The drone, thus, has become the everyday aircraft used for playful purposes as well as for professional purposes, see aerial footage and surveys, or even short-range air raid. This variety of purposes immediately defines its characteristics and construction solutions. In response to these purposes, there are numerous advantages of a rotary-wing aircraft over a fixed-wing model, firstly the economic cost.

The considerable ease of piloting appears fundamental due to technological developments in microelectronics and now reliable control systems, that make possible lots of varieties of missions, previously carried out by helicopters with human pilots.

Furthermore, the possibility of flying indoors is not negligible, it has been made much safer now by the modern control system, and it would have been unthinkable in the past. In recent years, lots of research has been carried out all over the world regarding the simulation and the control of these devices. They are different according to the requested task, for example, the drone's ability to be a carrier of a load. By realizing systems of this form and implementing control systems, it can be possible to carry out missions in which previously

manned helicopters were employed.

Examples are the meausurement of the magnetic field of the Earth using a drone with a suspended magnetometer, or the study of the seabed carried out by a drone with payload immersed in water.

Unlike traditional aircraft, RPA can be used in situations characterized by a great danger to human life and in areas difficult to reach, also flying at low altitude. For this reason, they can be used during the monitoring phases of areas affected by natural or artificial disasters (earthquakes, floods, road accidents, etc.).

1.2 Applications

In recent years, technologies related to the development of RPAs systems have undergone a rapid surge. In particular, technological development in the field of sensors made it possible to equip RPAs with many different loads, in the visible spectrum (compact digital cameras or professional), infrared (thermal cameras), multispectral cameras, up to sensors more advanced such as sensors for monitoring air quality. Here are some civil applications for RPAs:

- The monitoring of Archaeological sites, against the looting and illegal trade. A clear example is the monitoring of the ancient necropolis of Fifa in Jordan with drones. It is a logical solution, also considering the difficulty of hiring staff in militarily active areas, such as the Middle East.

Figure 1.2: An archaeologist monitoring a historical site [4]

- Monitoring of Thermal power plants and industrial plants. RPAs can also be used to monitor over time the electricity production plants, or, more generally, industrial systems, using sensors (thermal imaging

cameras, multispectral cameras, etc.).

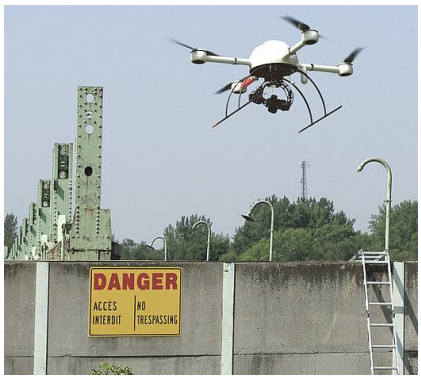


Figure 1.3: A drone monitoring an industrial power plant [5]

- Remote sensing. It is the technical-scientific discipline or applied science with diagnostic-investigative purposes that allows us to derive qualitative and quantitative information on the environment and on objects placed at a distance from a sensor through electromagnetic radiation measurements (emitted, reflected or transmitted) that interacts with the physical surfaces of interest. Thanks also to the possibility of flying at very low altitudes and of having small but suitable sensors available, RPAs classified as "mini-drones" can be used for applications linked to remote sensing, such as the creation of agricultural crop vigor maps and monitoring the health of vegetation. Another remote sensing useful application is in creating coverage maps and land-use maps, for the analysis and the support in the phases that follow a natural disaster immediately or for the monitoring and mapping of the thermal losses of private and public buildings (houses, warehouses, industrial plants). This is very important in this period when there is much talk about sustainable development and loss of land to be allocated to areas greens.

Figure 1.4: Airborne Hyperspectral sensor [6]

- Aerial photogrammetry and architectural survey: Photogrammetry is a survey technique that allows acquiring metric data of an object (shape and position) through a pair of stereometric frames. With the advent of small digital cameras (compact or reflex) which can guarantee a high-quality standard concerning the image produced, photogrammetry can be approached to the RPA and to their use to create digital soil models, for orthophoto production and for the architectural survey of infrastructures and buildings to create 3D models.

Figure 1.5: A drone monitoring the construction of a building [7]

- Environmental monitoring and natural disasters: The RPAs have been actively used in monitoring areas severely affected by earthquakes and floods. Some examples are the US Global Hawk RPAs that have flown over the nuclear power plant in Fukushima Dai-ichi, Japan, entering the forbidden zone to monitor the reactors after the explosions caused by the Tōhoku earthquake in 2011, also taking photos with infrared sensors. The high radioactivity made the presence of humans impossible.

Figure 1.6: the US Global Hawk model that flew over the nuclear power plant in Fukushima after the earthquake, monitoring the reactors [8]

- Biodiversity and fauna monitoring: RPAs can be used for monitoring wild animals. There are periodic numerical controls of those species which are either endangered or, on the contrary, which have a high rate of reproduction that could be a problem, both for the biodiversity of the environment in which they live and for the economic damage caused to agricultural production and livestock in the area.

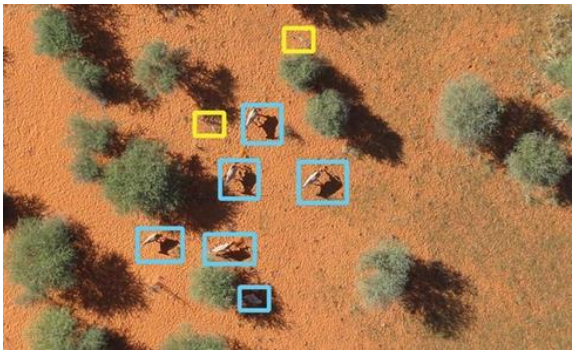


Figure 1.7: Use of Swiss AI and drones to count African wildlife [9]

- Search and Rescue Operations: RPAs can play an essential role in search and rescue operations for quick reconnaissance and detection, after the occurrence of emergencies.

Figure 1.8: the DJI - M200 Series during Search and Rescue in Extreme Environments [10]

- Video footage and photographs for general purpose: RPAs in combination with the latest and lightest digital video cameras, including general-purpose ones and not only professional, are making themselves more and more competitive for all those needs of "aerial" shooting, that substituted other complicated and expensive tools such as the dolly shot or at higher altitudes the helicopter.

CHAPTER 2

MODEL-BASED DESIGN AND SPECIFICATIONS

2.1 Model-based software design

We can imagine that we should design a controller for an industrial robot, a drone, a wind turbine, an autonomous vehicle, an excavator, or a servo motor. If the code is written manually and the search for requirements data is document-based, the only way to respond to the previous questions will be through "trial and error" procedure and tests on a physical prototype, which can also be expensive.

If a single requirement changes, the whole system will have to be redesigned, delaying the delivery of days or even weeks.

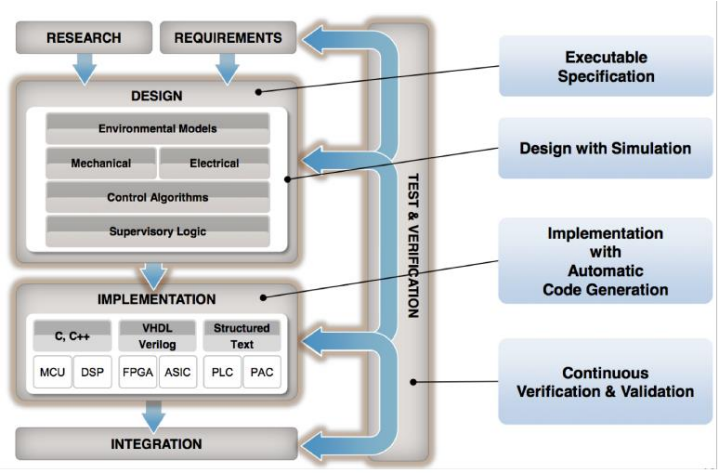


Figure 2.1: Workflow of Model-based design [11]

Following the **Model-Based design** (**MBD**) procedure (Figure 2.1), instead of developing a hand-written code and use hand-written documents, a model of the system can be developed, subdividing it into different subsystems of variable complexity. For a control system, they are generally the plant, actuators, sensors, and the controller.

Model-Based Design consists of a technique empowering quick and cheap design of dynamic systems, control systems, communication systems, and signal processing. In this method, the model of the system is at the focal point of the development procedure, beginning from requirements, then implementation, and finally testing. It can be continuously refined throughout the development process, and it can be simulated at any time to get an instant view of the system behavior. Multiple scenarios can be tested without any risk, without any delay and without using any expensive machinery directly. Various techniques can be applied to create the mathematical model to be inserted into the Simulink project, and the physical system can be represented as much detailed as the task that must be accomplished requires.

In this project, the ready-made general model of a quadcopter is provided by *MathWorks* together to a three-dimensional graphic simulator that represents the system to be controlled (Simulink 3d environment).

In the Figure below (Figure 2.2), the approach that describes Model-based software design is shown, which is also known as "V-shaped" development flow.

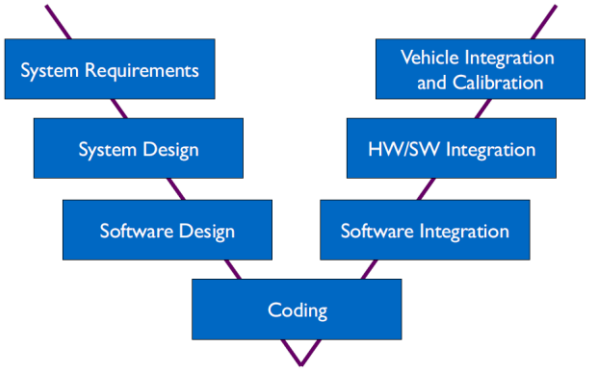


Figure 2.2: V-shaped Workflow of Model-based design [11]

2.1.1 Model-in-the-loop test

In this first stage, both the **Plant** (systems to be controlled) and the **Controller** (algorithm controlling the Plant) are modeled. The Simulation helps in refining both models and evaluate design alternatives.

The model exists entirely in native simulation tools (Simulink and Stateflow), and this phase is useful for control algorithm development.

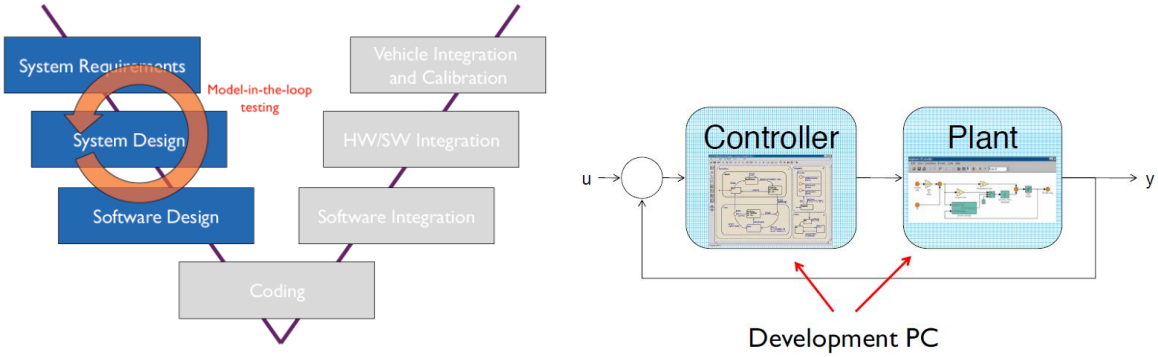


Figure 2.3: Model-in-the-loop test [11]

2.1.2 Optimization, code generation, and Software-in-the-loop test

In this stage, different transformation rules produce hardware and software. The implementation is co-simulated with the plant model to test its correctness, and it is still executed on a PC. A portion of the model is implemented in the native simulation tool (e.g., Simulink / Stateflow), and another portion as executable C-code. This phase is suitable to test the controller implementation written in C-code.

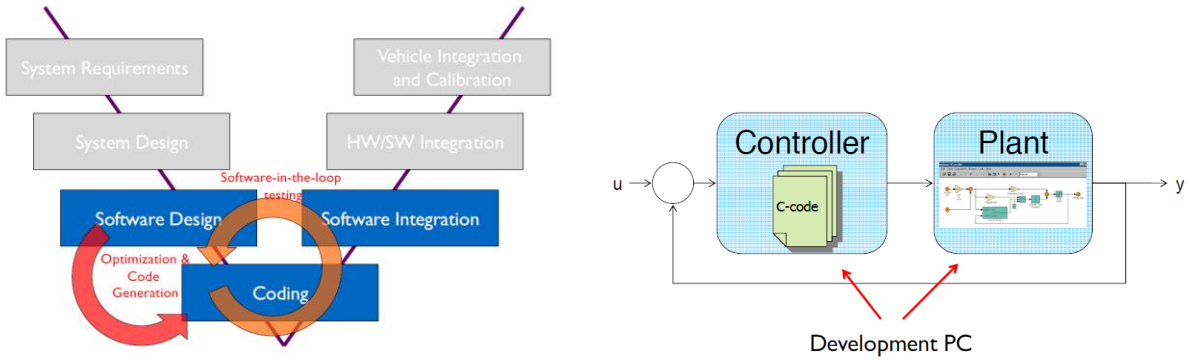


Figure 2.4: Optimization, Code generation, and Software-in-the-loop test [11]

2.1.3 Processor-in-the-loop test

In this stage, the implementation is deployed and runs on the target hardware (e.g., EVB or ECU). The implementation is co-simulated with the plant model to test its correctness. However, the system is not running in real-time yet.

A portion of the model is implemented in the native simulation tool (e.g., Simulink / Stateflow), and another portion as executable C-code, running on target hardware or rapid prototyping hardware.

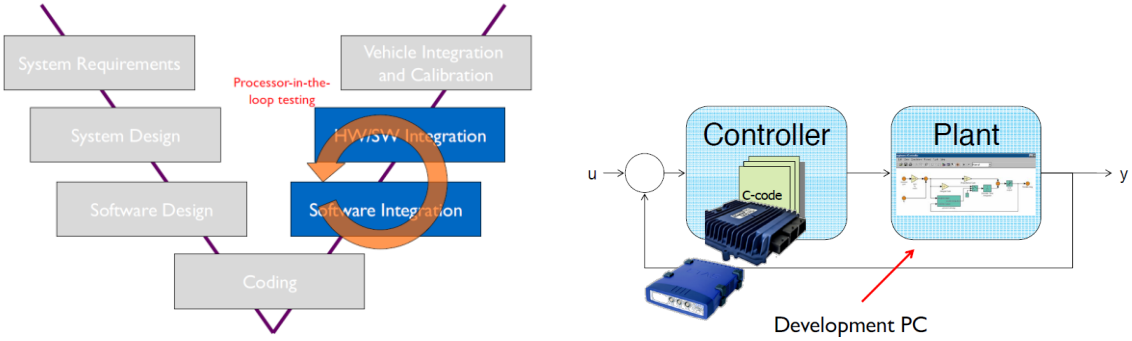


Figure 2.5: Processor-in-the-loop test [11]

2.1.4 Hardware-in-the-loop test

In this stage, the controller implementation is co-simulated with the plant model to test its correctness. The controller runs on the target hardware, the plant model on rapid prototyping hardware, and the system is running in real-time. A portion of the model runs in a real-time simulator, and another portion could be implemented as physical hardware (ECU). This phase is suitable to test the interactions with hardware and real-time-performance. The controller algorithm is sometimes distributed among two devices: the ECU and rapid prototyping hardware.

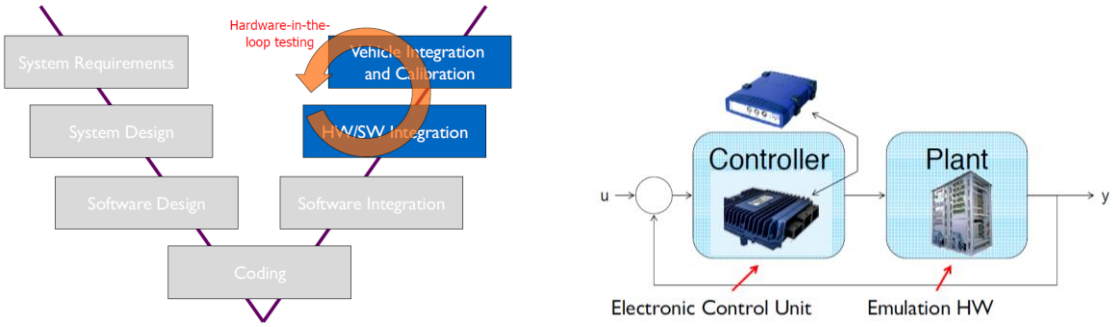


Figure 2.6: Hardware-in-the-loop test [11]

2.2 Project goals and specifications

This Thesis research project aims to study and to design a model-based algorithm on Simulink and Matlab through simulation, and to test it by deploying on the Hardware of a Parrot Mambo Fly mini-drone through the interfaces provided with its Hardware support package in Simulink.

The first goal to be accomplished is to maintain the drone stable during the flight maneuvers adopted to

follow the path. In addition to the previous objective, the drone should acquire images through its downward-facing camera, then elaborate them with several image processing techniques and move the drone along the track according to a specific path planning control algorithm that receives the vision-based data and elaborate them, obtaining in this way an autonomous vision-based control system.

The line specifications are adapted from the rules of the competition organized by MathWorks: the multinational private company specialized in the production of software for mathematical calculations. Its main products are MATLAB and Simulink. In addition to government and industrial purposes, the software products made by the company are also used in teaching and research in many universities throughout the world.

The rules of this competition are based precisely on the same purpose of this Thesis, which is to make the drone follow autonomously a path of a particular color (specification useful for the image processing algorithm that should have a filter with a threshold). The project in Simulink is the one that Mathworks provided to the participants who had to modify the general model of the Quadcopter according to the aim of the competition.

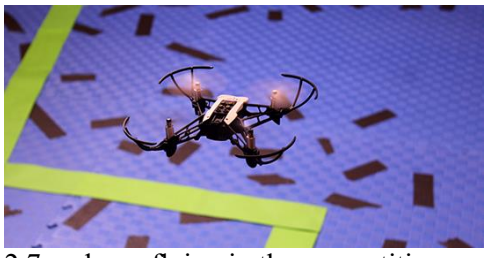


Figure 2.7: a drone flying in the competition arena [13]

Some of the rules describing the track are the following:

- The line track is about 4 inches in width.
- The circular landing figure has 8 inches in diameter.
- The path is made of connected line segments only and does not have any smooth curves at the connections.
- The end of the path is about 10 inches far from the landing circle.
- For the Thesis project, it was chosen either a random color like red or, more conveniently, for the image

processing algorithm, a white line on a black background to enhance the contrast useful in the image processing algorithm.

- The background is not made by a single color (black), but it has texture (different color papers were used in the tests). This texture helps the optical flow estimation algorithm, through which it is possible to estimate the speed and the displacement of the drone. This is not necessary for the Simulation 3d environment, which does not influence the sensors as the physical environment does in real-time.

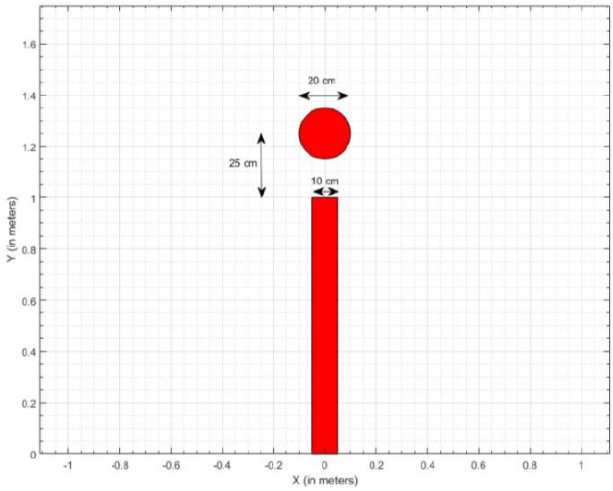


Figure 2.8: rules on the path measurements in the competition arena [14]

The circle is used to land the drone as part of the competition, but for this Thesis project, this specification was an optional goal, not necessary to the accomplishment of the main task, which is the line tracking, but it is considered only to improve the algorithm.

CHAPTER 3

MODELING AND CONTROL OF QUADCOPTERS

3.1. Main Components and Working Principles

In this chapter, the working principle of a Quadrotor will be explained, and its mathematical model will be analyzed, deriving its kinematics and its dynamical equations (useful for the implementation on Simulink). The approach followed is the one used by Orsag et al. [16] and Totu [15].

The main components of quadrotors are:

- The frame, which gives physical support for the other components, consists of a center and four arms.
- Electronics and a battery in the center of the frame.
- A motor and a propeller on each arm.

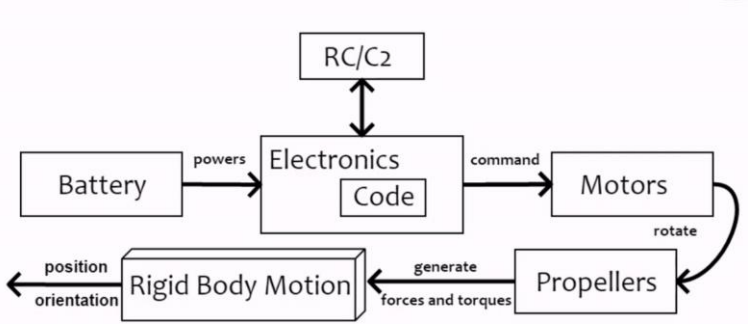


Figure 3.1: Schematic of the model of a quadrotor [15]

A spinning propeller creates thrust, a force that is perpendicular to the propeller's rotation plane. Besides the thrust force, a spinning propeller produces a turning effect (or torque) on the quadrotor frame. It is in the opposite direction to the propeller's rotation.

There are two types of propellers:

- Type 1, or right-handed propeller, produced thrust in the upward direction when rotating

counterclockwise.

- Type 2, or left-handed propeller, produces thrust in the upward direction when rotating clockwise.

A quadrotor has two Type 1 and two Type 2 propellers (figure 3.2).

Figure 3.2: type 1 and type 2 propellers [15]

The quadrotor is an underactuated system. It has four propellers, but there are six degrees of spatial freedom.

- Translational degrees of freedom: up/down, forward/backward, left/right.
- Rotational degrees of freedom: heading, pitch, and roll.

Since the actuators (4) are less than the D.O.F. (6), then some directions are not controllable at any given time. For example, the quadrotor is not capable of moving right without rotating in that direction. The same goes for forward and backward motions as well.

It is possible to overcome this underactuation problem by designing a control system that couples rotations and thrusts to achieve the overall tasks.

The two possible quadrotors configurations are the “Plus” configuration and the “Cross” configuration.

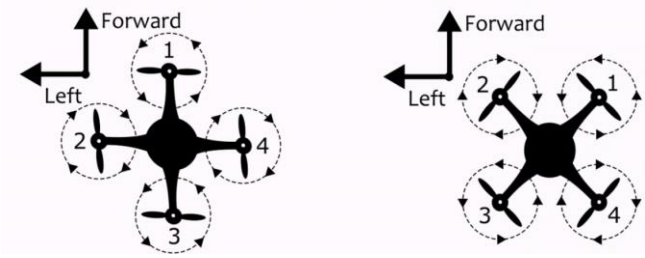


Figure 3.3: “Plus” and “cross” configurations [15]

Let us assume that the frame of the quadrotor is perfectly level with the ground.

If the same commands are given to the motors, the overall thrust is in the vertical direction, and it can

compensate for the gravity to generate a movement up (Figure 3.4).

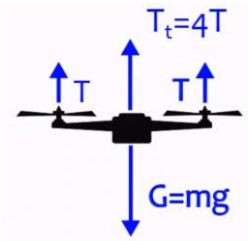


Figure 3.4: Forces balance between thrusts and weight while Hovering [15]

If the overall thrust is less than the force of gravity, then the quadrotor will move down.

To pitch, i.e., rotate around the left-right axis, we must create an unbalance in the forward-side and backward-side forces. Pitching forwards is done by decreasing the force in the forward side and/or increasing in the backward side.

Pitching backward is done by decreasing the force on the backward side and/or increasing on the forward side.

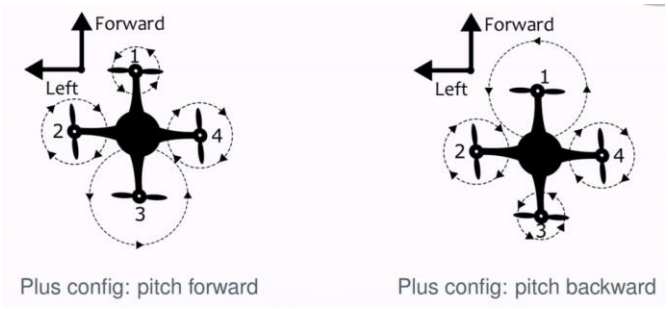


Figure 3.5: Pitch in “Plus” configuration [15]

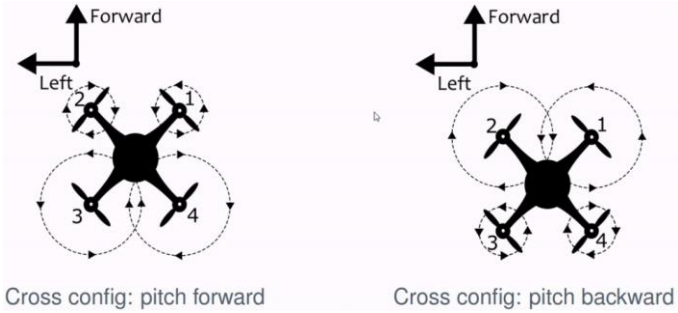


Figure 3.6: Pitch in “cross” configuration [15]

The pitching rotation is coupled with translation on the forward/backward direction. When the quadrotor is

pitching forward/backward, it will also move forward/backward.

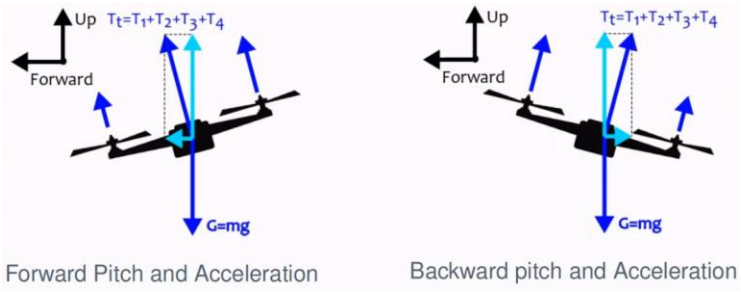


Figure 3.7: forces balance during Pitch maneuver [15]

To roll, i.e., rotate around the forward-backward axis, we must create an unbalance in the left-side and right-side forces. Rolling right is done by decreasing the force on the right side and/or increasing on the left side. Rolling left is done by decreasing the force in the left part and/or increasing in the right part.

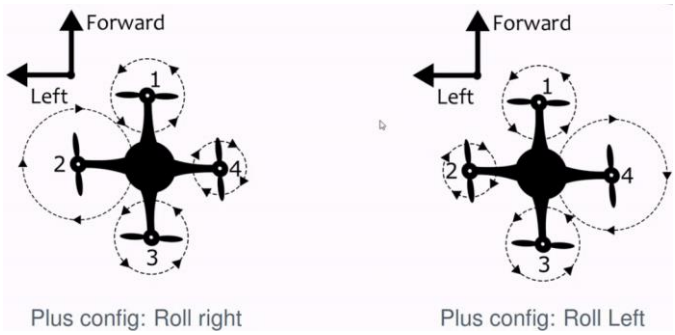


Figure 3.8: Roll in “plus” configuration [15]

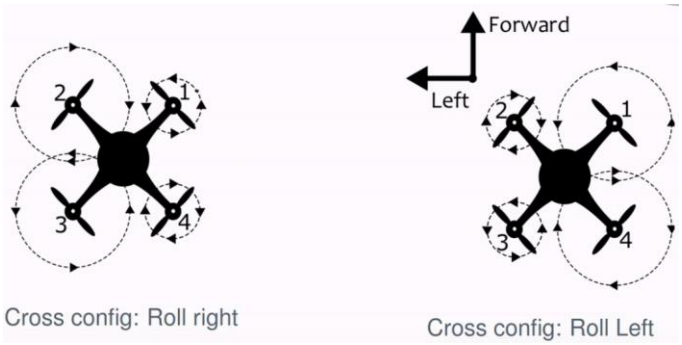


Figure 3.9: Roll in “cross” configuration [15]

The rolling rotation is coupled with translation on the left/right direction. When the quadrotor is rolling left/right, it will also move left/right.

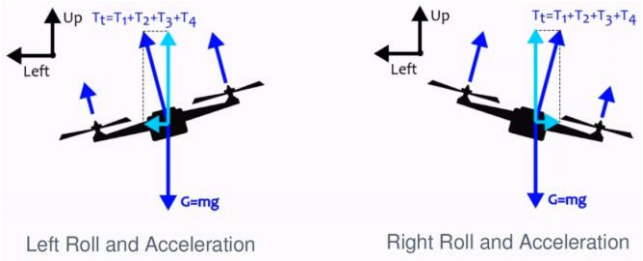


Figure 3.10: forces balance during Roll maneuver [15]

Yawing is, instead, the rotation around the up/down direction. It is important to remember that rotating propellers are causing an opposite torque/turning effect on the frame. If all propellers were rotating in the same direction, the frame would rotate around its up/down axis in the opposite direction, and it would be spinning in place. The quadrotor has two types of propellers, such that they rotate in the opposite direction in pairs, and the reaction effect is canceled.

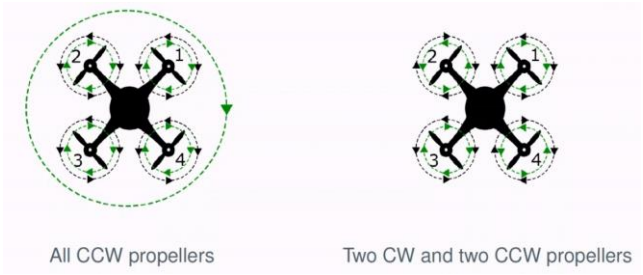


Figure 3.11: Torques balance [15]

The input is increased on the pair of clockwise propellers and decreased on the counterclockwise pair to create a controlled counterclockwise yawing rotation, whereas the input is increased on the pair of counterclockwise propellers and decreased on the clockwise pair to create a controlled clockwise yawing rotation.

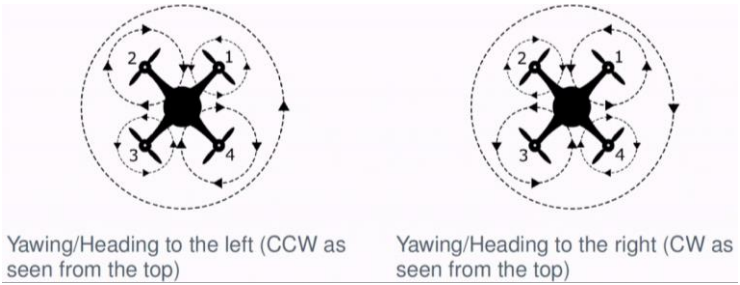


Figure 3.12: Yawing maneuver and torques balance [15]

Summarizing it is possible to say that Pitching and forward/backward motion are coupled, Rolling and left/right motion are coupled, Up and down motion is independent, and Yawing (changing of heading) is independent.

It could be useful to implement a motor mixing algorithm that can convert the roll, pitch, yaw, and thrust commands into the motor speeds, and which can be useful in the model representation on Simulink.

The principles to generate Thrust, Roll, Pitch, Yaw maneuvers described before are applied in the same way to the MMA (Motor mixing algorithm) according to the convention followed by Douglas [19].

$$\text{motor}_{\text{front right}} = \text{thrust}_{\text{cmd}} + \text{yaw}_{\text{cmd}} + \text{pitch}_{\text{cmd}} + \text{roll}_{\text{cmd}} \quad (3.1)$$

$$\text{motor}_{\text{front left}} = \text{thrust}_{\text{cmd}} - \text{yaw}_{\text{cmd}} + \text{pitch}_{\text{cmd}} - \text{roll}_{\text{cmd}} \quad (3.2)$$

$$\text{motor}_{\text{back right}} = \text{thrust}_{\text{cmd}} - \text{yaw}_{\text{cmd}} - \text{pitch}_{\text{cmd}} + \text{roll}_{\text{cmd}} \quad (3.3)$$

$$\text{motor}_{\text{back left}} = \text{thrust}_{\text{cmd}} + \text{yaw}_{\text{cmd}} - \text{pitch}_{\text{cmd}} - \text{roll}_{\text{cmd}} \quad (3.4)$$

3.2. Rigid body motion

A rigid body object can be seen as a system consisting of a vast (in the limit infinite) number of small (in the limit infinitesimal) point-mass particles, with the property that the relative positions of the particles relative to each other are constant (rigidly).

As the following picture (3.13) shows, the Rigid body motion model of the drone is needed to understand how from generalized forces in input, it is possible to derive the position and the attitude of the drone in the space.

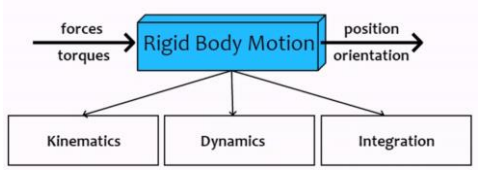


Figure 3.13: Rigid body motion model schematic [15]

3.2.1. Kinematics

In the Kinematics of the rigid body, there are different concepts introduced and explained. Some of them are

the translational and rotational motion, position, linear velocity, and linear acceleration vectors, the rotation matrix that defines the body frame attitude relative to a fixed inertial frame, angular velocity, and angular acceleration, motion composition between frames, etc.

Firstly, the cartesian coordinate system must be defined to describe the geometrical vectors like the position vector **p**.

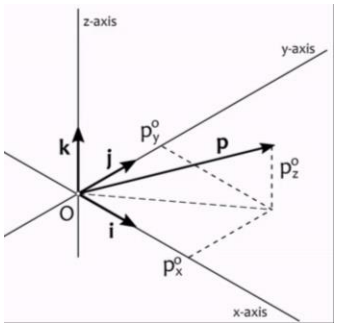


Figure 3.14: Vector p in the Cartesian reference frame (inertial) [15]

The vector **p** can be described either through a geometrical description, like the following:

$$\mathbf{p} = p_x^0\mathbf{i} + p_y^0\mathbf{j} + p_z^0\mathbf{k} \tag{3.5}$$

or through an algebraic description:

$$\mathbf{p}^0 = \begin{bmatrix} p_x^0 \\ p_y^0 \\ p_z^0 \end{bmatrix},\ \mathbf{i}^0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix},\ \mathbf{j}^0 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},\ \mathbf{k}^0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.6}$$

It could be possible to define an inertial cartesian reference frame centered in O, that can be called e-frame (for “external”) and to define the points A and C respectively in the propeller center and in the center of mass of the drone, which should coincide approximately with the center of the drone frame.

Three positions vector could be introduced:

- Position of C relative to O, **p**
- Position of A relative to O, **r**
- Position of A relative to C, **s**

Furthermore, it is possible to put in evidence the following relationship between these vectors:

$$\mathbf{r} = \mathbf{p} + \mathbf{s} \tag{3.7}$$

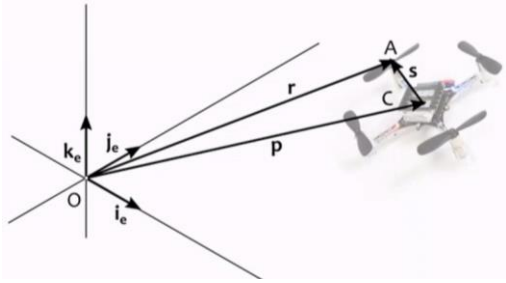


Figure 3.15: quadrotor's position vectors in the inertial frame [15]

Under a translational motion, there will be a variation of the position vectors **r** and **p**, but not of vector **s** that will be constant.

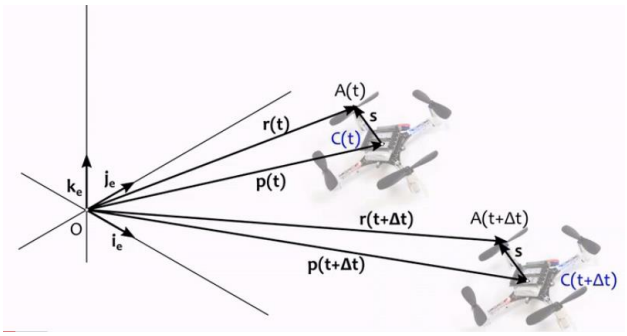


Figure 3.16: translation of the quadrotor [15]

The velocity and acceleration of the quadrotor are the time derivative of vector **p**:

$$\mathbf{v}_P(t) \triangleq \dot{\mathbf{p}}(t) = \frac{d\mathbf{p}(t)}{dt} \tag{3.8}$$

$$\mathbf{a}_P(t) \triangleq \dot{\mathbf{v}}_P(t) = \ddot{\mathbf{p}}(t) = \frac{d\mathbf{v}(t)}{dt} = \frac{d^2\mathbf{p}(t)}{dt} \tag{3.9}$$

Now it will be interesting to see how the vector **r** change in time:

$$\mathbf{v}_r = \dot{\mathbf{r}} = \dot{\mathbf{p}} + \dot{\mathbf{s}} = \dot{\mathbf{p}} = \mathbf{v}_p \quad , \qquad \mathbf{v}_r = \mathbf{v}_P \tag{3.10}$$

$$\mathbf{a}_r = \ddot{\mathbf{r}} = \dot{\mathbf{v}}_r = \dot{\mathbf{v}}_p = \mathbf{a}_p \quad , \qquad \mathbf{a}_r = \mathbf{a}_p \tag{3.11}$$

Under a translation movement, it can be noticed that all points of the moving object; in this case, the drone has the same velocity and acceleration.

Under a rotational motion, instead, is possible to observe that **p(t) = 0**, and a body reference frame, "b-

frame”, that rotates together with the quadrotor, around an axis passing through its center of mass, and the position vector **r** defined by OA, which points to a fixed point A on the body frame that could be the propeller center.

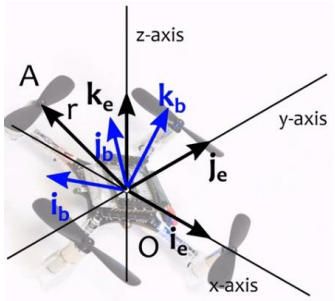


Figure 3.17: quadrotor’s body reference frame and inertial frame [15]

The goal is to represent the speed and acceleration of vector **r**.

The b-frame can be denoted with respect to the e-frame basis vectors through the following notation:

$$\mathbf{i}_b = i^e_{b,x}\mathbf{i}_e + i^e_{b,y}\mathbf{j}_e + i^e_{b,z}\mathbf{k}_e \tag{3.12}$$

$$\mathbf{j}_b = j^e_{b,x}\mathbf{i}_e + j^e_{b,y}\mathbf{j}_e + j^e_{b,z}\mathbf{k}_e \tag{3.13}$$

$$\mathbf{k}_b = k^e_{b,x}\mathbf{i}_e + k^e_{b,y}\mathbf{j}_e + k^e_{b,z}\mathbf{k}_e \tag{3.14}$$

It is possible to express vector **r** in the b-frame

$$\mathbf{r} = r^b_x\mathbf{i}_b + r^b_y\mathbf{j}_b + r^b_z\mathbf{k}_b \tag{3.15}$$

and to refer this equation in the e-frame:

$$\mathbf{r}^e = r^b_x\mathbf{i}^e_b + r^b_y\mathbf{j}^e_b + r^b_z\mathbf{k}^e_b \tag{3.16}$$

The vector **r** can be represented either in b-frame or in e-frame, and the passage between them is done by multiply a **rotation matrix R**:

$$\mathbf{r}^e = \mathbf{R}^e_b\mathbf{r}^b \tag{3.17}$$

Where:

$$\mathbf{r}^{e} = \begin{bmatrix} r_x^e \\ r_y^e \\ r_z^e \end{bmatrix}, \qquad \mathbf{r}^{b} = \begin{bmatrix} r_x^b \\ r_y^b \\ r_z^b \end{bmatrix}, \qquad \mathbf{R}_b^e = \begin{bmatrix} i_{b,x}^e & j_{b,x}^e & k_{b,x}^e \\ i_{b,y}^e & j_{b,y}^e & k_{b,y}^e \\ i_{b,z}^e & j_{b,z}^e & k_{b,z}^e \end{bmatrix} \tag{3.18}$$

The rotation matrix consists of 9 elements, but only three are required to specify a rotation/orientation in 3D space. Some alternative ways of representing the orientation in a more compact way instead of the Rotation matrix are the **Unit Quaternions** (also known as "Euler parameters") and the **Euler angles**.

Unit Quaternions are a set of 4 numbers (4D vectors). They are equivalent to rotation matrices, and they are the most efficient and numerically stable tool to express orientations and rotations. For this reason, as it will be shown in chapter 6, the Simulink quadcopter simulation model also has the unit quaternions representation, provided by the Aerospace blockset add-on tool.

Quaternions are 4-dimensional vectors, and they are the combination of a 3-dimensional vector with a scalar.

$$\mathbf{q} = \begin{bmatrix} s \\ \mathbf{v} \end{bmatrix} = [s \quad v_1 \quad v_2 \quad v_3]^T \tag{3.19}$$

It is possible to express a rotation matrix as a function of a quaternion is the following way:

$$\mathbf{r}^{e} = [-\mathbf{v} \quad s\mathbf{I}_3 + [\mathbf{v}]_x] \begin{bmatrix} -\mathbf{v}^T \\ s\mathbf{I}_3 + [\mathbf{v}]_x \end{bmatrix} \mathbf{r}^{b} = \mathbf{R}_b^e(\mathbf{q})\mathbf{r}^{b} \tag{3.20}$$

$$\mathbf{R}_b^e(\mathbf{q})\mathbf{r}^{b} = \begin{bmatrix} s^2 + v_1^2 - v_2^2 - v_3^2 & 2v_1v_2 - 2sv_3 & 2v_1v_3 + 2sv_2 \\ 2v_1v_2 + 2v_3s & s^2 - v_1^2 + v_2^2 - v_3^2 & -2sv_1 + 2v_2v_3 \\ 2v_1v_3 - 2sv_2 & 2sv_1 + 2v_2v_3 & s^2 - v_1^2 - v_2^2 + v_3^2 \end{bmatrix} = \tag{3.21}$$

$$= 2\begin{bmatrix} s^2 + v_1^2 - 0.5 & v_1v_2 - sv_3 & v_1v_3 + sv_2 \\ v_1v_2 + v_3s & s^2 + v_2^2 - 0.5 & -sv_1 + v_2v_3 \\ v_1v_3 - sv_2 & 2sv_1 + 2v_2v_3 & s^2 + v_3^2 - 0.5 \end{bmatrix} \tag{3.22}$$

The derivative of the quaternion is the following:

$$\dot{\mathbf{r}} = \frac{1}{2}\begin{bmatrix} -\mathbf{v}^T \\ s\mathbf{I}_3 + [\mathbf{v}]_x \end{bmatrix} \omega^{b} = \frac{1}{2}\begin{bmatrix} -v_1\omega_x^b - v_2\omega_y^b - v_3\omega_z^b \\ s\omega_x^b - v_3\omega_y^b + v_2\omega_z^b \\ v_3\omega_x^b + s\omega_y^b - v_1\omega_z^b \\ -v_2\omega_x^b + v_1\omega_y^b + s\omega_z^b \end{bmatrix} \tag{3.23}$$

The other alternative way to represent orientation is through **Euler Angles**. They are a set of 3 numbers that are equivalent to a rotation matrix, but they have an intrinsic disadvantage because they have singularities in operation (divisions by zero in expressions), and these need to be treated as special cases.

The rotation of the frame can be split up in a series of three individual rotations around the coordinate system axes, that is a product of three 2D rotations

$$\mathbf{R}_{\mathrm{b}}^{\mathrm{e}} = \mathbf{R}_{\mathrm{z}}(\varphi)\mathbf{R}_{\mathrm{y}}(\theta)\mathbf{R}_{\mathrm{x}}(\psi) \tag{3.24}$$

where:

$$\mathbf{R}_{\mathrm{z}}(\varphi) = \begin{bmatrix} \mathrm{c}\varphi & -\mathrm{s}\varphi & 0 \\ \mathrm{s}\varphi & \mathrm{c}\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{R}_{\mathrm{y}}(\theta) = \begin{bmatrix} \mathrm{c}\theta & 0 & \mathrm{s}\theta \\ 0 & 1 & 0 \\ -\mathrm{s}\theta & 0 & \mathrm{c}\theta \end{bmatrix}, \quad \mathbf{R}_{\mathrm{x}}(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathrm{c}\psi & -\mathrm{s}\psi \\ 0 & \mathrm{s}\psi & \mathrm{c}\psi \end{bmatrix} \tag{3.25}$$

The relation between the rotation matrix and the Euler angles can be expressed in the following way:

$$\mathbf{R}_{\mathrm{b}}^{\mathrm{e}} = \begin{bmatrix} \mathrm{c}\theta\mathrm{c}\varphi & \mathrm{s}\psi\mathrm{s}\theta\mathrm{c}\varphi - \mathrm{c}\psi\mathrm{s}\varphi & \mathrm{c}\psi\mathrm{s}\theta\mathrm{c}\varphi + \mathrm{s}\psi\mathrm{s}\varphi \\ \mathrm{c}\theta\mathrm{s}\varphi & \mathrm{s}\psi\mathrm{s}\theta\mathrm{s}\varphi - \mathrm{c}\psi\mathrm{c}\varphi & \mathrm{c}\psi\mathrm{s}\theta\mathrm{s}\varphi - \mathrm{s}\psi\mathrm{c}\varphi \\ -\mathrm{s}\theta & \mathrm{s}\psi\mathrm{c}\theta & \mathrm{c}\psi\mathrm{c}\theta \end{bmatrix} \tag{3.26}$$

ψ is the roll angle, φ is the yaw angle, and θ is the pitch angle, and. This intrinsic sequence is also known as Cardan angles.

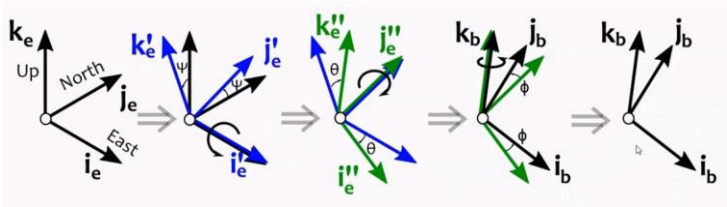


Figure 3.18: Euler angles and rotations [15]

In total, there are 12 extrinsic combinations and 12 intrinsic combinations.

As far as the derivation of velocity and acceleration for rotational motion is concerned, it can be observed that $\mathbf{r}^{\mathbf{b}}$ is constant (because the b-frame rotates with the drone) and $\mathbf{r}^{\mathbf{e}}$ is changing with time. If the derivative of a rotation matrix is used, the **linear velocity** in the e-frame is written as:

$$\mathbf{v}^{\mathrm{e}}(\mathrm{t}) = \dot{\mathbf{R}}_{\mathrm{b}}^{\mathrm{e}}(\mathrm{t})\mathbf{r}^{\mathrm{b}}(\mathrm{t}) \tag{3.27}$$

Instead, the derivative of the rotation matrix can be denoted as:

$$\dot{\mathbf{R}}_b^e(t) = \Omega^e \mathbf{R}_b^e(t) = \mathbf{R}_b^e(t)\Omega^b \tag{3.28}$$

Matrices **Ω** are skew-symmetric, and they have the following form:

$$\boldsymbol{\omega} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \qquad \boldsymbol{\Omega} = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} = [\boldsymbol{\omega}]_x \tag{3.29}$$

The multiplication by a skew-symmetric matrix is another way to represent the vector cross-product. The vector **ω** is the **angular velocity**, which is a physical vector.

The derivative of the rotation matrix is expressed as:

$$\dot{\mathbf{R}}_b^e(t) = [\boldsymbol{\omega}^e]_x \mathbf{R}_b^e(t) = \mathbf{R}_b^e(t)\left[\boldsymbol{\omega}^b\right]_x \qquad \text{with} \qquad \boldsymbol{\omega}^e = \mathbf{R}_b^e \boldsymbol{\omega}^b \tag{3.30}$$

The angular rate in the rotating body frame $\boldsymbol{\omega}^b$ can be obtained from the measurements of a MEMS gyroscope sensor placed on the b-frame.

The linear velocity vector is written with the following notation:

$$\mathbf{v}^e = [\boldsymbol{\omega}^e]_x \mathbf{r}^e \tag{3.31}$$

The **angular acceleration α** is the derivative of the angular velocity vector ω, and its unit is [rad/s].

The linear acceleration in the e-frame $\mathbf{a}^e$ can be written as:

$$\mathbf{a}^e = [\boldsymbol{\alpha}^e]_x \mathbf{r}^e + [\boldsymbol{\omega}^e]_x [\boldsymbol{\omega}^e]_x \mathbf{r}^e \tag{3.32}$$

Instead, $\mathbf{a}^b$ is the output of the quadrotor's accelerometer sensor:

$$\mathbf{a}^b = \mathbf{R}_e^b \mathbf{a}^e \tag{3.33}$$

Under a roto-translational motion, we have the two motions combined, which is the generalized case, and the most important in the drone application.

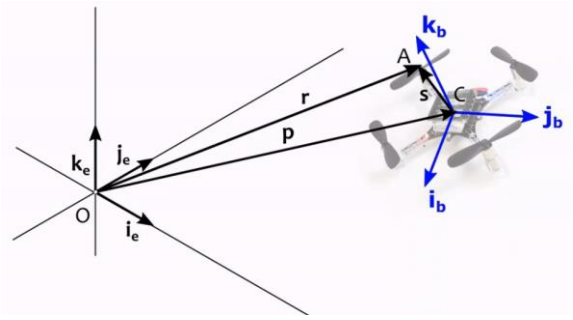


Figure 3.19: quadrotor's body reference frame and inertial frame [15]

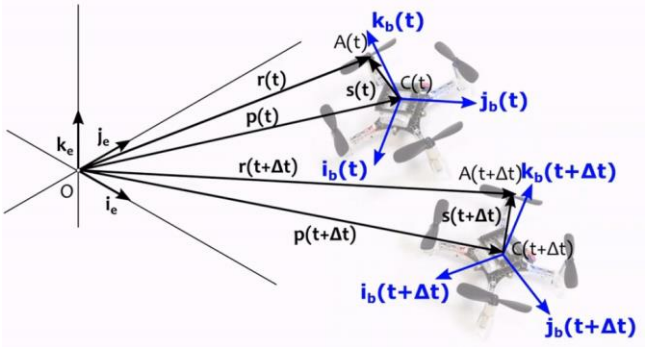


Figure 3.20: roto translation of the quadrotor [15]

The following position vector is obtained, combining translation and rotation and translation motion:

$$\mathbf{r}^e = \mathbf{p}^e + \mathbf{s}^e = \mathbf{p}^e + \mathbf{R}_b^e \mathbf{s}^b \tag{3.34}$$

Moreover, the following linear velocity of point **r** relative to e-frame is expressed as

$$\mathbf{V}_r^e = \mathbf{V}_p^e + \mathbf{R}_b^e [\omega^b]_x \mathbf{s}^b = \mathbf{V}_p^e + [\boldsymbol{\omega}^e]_x \mathbf{s}^e \tag{3.35}$$

The linear acceleration of point r is obtained deriving the linear velocity:

$$\mathbf{a}_r^e = \mathbf{a}_p^e + [\boldsymbol{\alpha}^b]_x \mathbf{s}^e + [\boldsymbol{\omega}^e]_x [\boldsymbol{\omega}^e]_x \mathbf{s}^e \tag{3.36}$$

3.2.2 Dynamics

The following laws are considered for the derivation of the equations describing the dynamics of the quadrotors.

- Newton's laws of motion for point masses/particles
- Euler's laws of motion for the rigid body (translational and rotational equation of motion)

Among the first ones the Second law tells that the sum of forces on a particle object is equal to the mass of

the object times the acceleration of the object in an inertial reference frame (in this project the hypothesis is that the earth is like as an inertial frame)

$$f_{total} = \sum_k f_k^e = m\mathbf{a}^e = m\ddot{\mathbf{p}}^e \Leftrightarrow \sum_k \begin{bmatrix} f_{k,x}^e \\ f_{k,y}^e \\ f_{k,z}^e \end{bmatrix} = m \begin{bmatrix} a_x \\ a_y^e \\ a_z^e \end{bmatrix} = \begin{bmatrix} \ddot{p}_x^e \\ \ddot{p}_y^e \\ \ddot{p}_z^e \end{bmatrix} \tag{3.37}$$

According to the Third principle of Dynamics, instead, when a particle applies a force on a second particle upon some form of interaction, contact, or at-a-distance), the second particle simultaneously applies a force equal in magnitude, but opposite, onto the first particle.

The direction of the two forces is along the straight line joining the point masses. If **i** and **j** are two particles, $\mathbf{f_{ij}}$ with which particle i acts upon particle j, and $\mathbf{r_i}$ and $\mathbf{r_j}$ are position vectors then:

$$\mathbf{f}_{ij} = -\mathbf{f}_{ji} \tag{3.38}$$

$$\mathbf{f}_{ij} = \pm\|\mathbf{f}_{ij}\|(\mathbf{r}_i - \mathbf{r}_j) \tag{3.39}$$

If a system of particles is considered, with fixed distance one each other, that is a rigid body, the Euler's laws of motion tell the dynamics behavior of this system. The resultant of the internal forces applied to these particles is, therefore, null and the only nonzero forces are the external ones, that can be thought to act on the center of gravity of the object. The resultant of external forces equals the center of gravity acceleration multiplied by the total mass of the system.

$$\mathbf{p} = \frac{1}{m}\sum_i m_i\mathbf{r_i} \tag{3.40}$$

$$\mathbf{f}_{ext,total}^e = m\ddot{\mathbf{p}}^e = m\mathbf{a}_p^e \tag{3.41}$$

Here **r** is the center of the gravity position vector, and **m** is the body mass. Forces not only push or pull a rigid body (translation) but also tend to rotate it, and the torque expresses this effect.

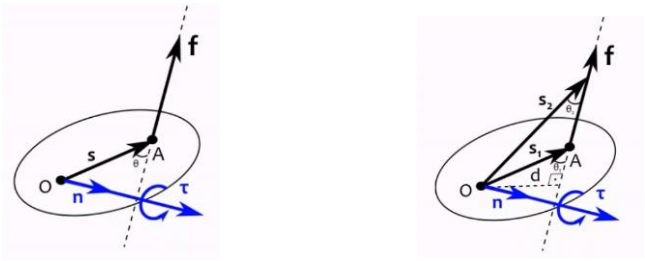


Figure 3.21: torque description [15]

$$\boldsymbol{\tau}_O = [\mathbf{s}]_x\mathbf{f} = \|s\|\|f\|\sin(\theta)\mathbf{n} = \|f\| \cdot d \cdot \mathbf{n} \tag{3.42}$$

If a quadrotor is considered like in the following picture, the external torque about O can be expressed in the following way:

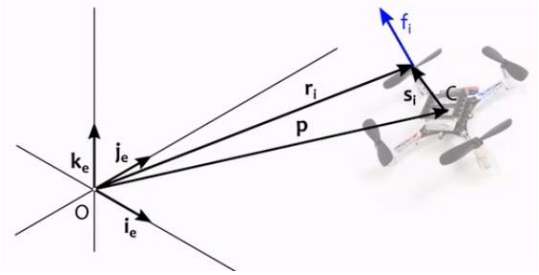


Figure 3.22: external force on the quadrotor [15]

$$\boldsymbol{\tau}_{O,ext} = [\mathbf{p}]_x\mathbf{f}_{total,ext} + \tau_{C,ext} \tag{3.43}$$

Furthermore, as observed for the resultant of the forces on a rigid body, the torque expression has also only the external component, because the sum of internal torque is zero.

Combining the last equation with the (3.41), the following equation is obtained:

$$\boldsymbol{\tau}_0^e = \sum_i m_i[\mathbf{r}_i^e]_x\mathbf{a}_i^e \tag{3.44}$$

$$\boldsymbol{\tau}_O^e - [\mathbf{p}^e]\mathbf{f}_{total,ext}^e = \mathbf{J}^e\alpha^e + [\boldsymbol{\omega}^e]_x\mathbf{J}^e\boldsymbol{\omega}^e \tag{3.45}$$

And combining equation (3.43) with (3.45), we can notice that the resultant of Torques about C (center of mass of the quadrotor) is a sum of two components as the following expression shows:

$$\boldsymbol{\tau}_C^e = \mathbf{J}^e\alpha^e + [\boldsymbol{\omega}^e]_x\mathbf{J}^e\boldsymbol{\omega}^e \tag{3.46}$$

Where:

$$\mathbf{J}^{e} = -\sum_{i} m_i [\mathbf{s}_i^e]_x [\mathbf{s}_i^e]_x \tag{3.47}$$

that is the global inertia matrix.

The rotational equation of motion can also be represented in body-frame coordinates:

$$\boldsymbol{\tau}_c^b = \mathbf{J}^b \boldsymbol{\alpha}^b + \left[\boldsymbol{\omega}^b\right]_x \mathbf{J}^b \boldsymbol{\omega}^b \tag{3.48}$$

And the global inertia matrix is represented in relation to the body inertia matrix as:

$$\mathbf{J}^b = \mathbf{R}_e^b \mathbf{J}^e \mathbf{R}_b^e \tag{3.49}$$

$$\mathbf{J}^b = -\sum_i m_i \left[\mathbf{s}_i^b\right]_x \left[\mathbf{s}_i^b\right]_x = -\sum_i m_i \begin{bmatrix} 0 & -z_i & y_i \\ z_i & 0 & -x_i \\ -y_i & x_i & 0 \end{bmatrix} \begin{bmatrix} 0 & -z_i & y_i \\ z_i & 0 & -x_i \\ -y_i & x_i & 0 \end{bmatrix} \quad \text{with} \quad \boldsymbol{s}_i^b = \begin{bmatrix} x_i \\ yi \\ z_i \end{bmatrix} \tag{3.50}$$

$$\mathbf{J}^b = \sum_i m_i \begin{bmatrix} y_i^2 + z_i^2 & -x_i y_i & -x_i z_i \\ -x_i y_i & x_i^2 + z_i^2 & -y_i z_i \\ -x_i z_i & -y_i z_i & x_i^2 + y_i^2 \end{bmatrix} \tag{3.51}$$

$$\mathbf{J}^b = \begin{bmatrix} \int_V (y_i^2 + z_i^2)\, dm & \int_V -x_i y_i dm & \int_V -x_i z_i dm \\ \int_V -x_i y_i\, dm & \int_V (x_i^2 + z_i^2) & \int_V -y_i z_i\, dm \\ \int_V -x_i z_i dm & \int_V -y_i z_i dm & \int_V (x_i^2 + y_i^2) \end{bmatrix} \tag{3.52}$$

Furthermore, it can be noticed that while $\boldsymbol{s}_e^b$ is variable in time, vector $\boldsymbol{s}_i^b$ is constant, meaning that $\mathbf{J}_e$ is time-dependent, while $\mathbf{J}_b$ in the body frame is time constant.

The differential equations of motion can be summarized and put together to form the dynamical model of the quadrotor, and they can be written either with the rotation matrix notation:

$$\dot{\mathbf{p}}^e = \mathbf{v}^e \tag{3.53}$$

$$\dot{\mathbf{v}}^e = \frac{1}{m} \mathbf{f}_{total,ext}^e = \frac{1}{m} \mathbf{R}_b^e \mathbf{f}_{total,ext}^b \tag{3.54}$$

$$\dot{\mathbf{R}}_b^e = \mathbf{R}_b^e \left[\boldsymbol{\omega}^b\right]_x \tag{3.55a}$$

$$\dot{\boldsymbol{\omega}}^b = \left(\mathbf{J}^b\right)^{-1} \left(-\left[\omega^b\right]_x \mathbf{J}^b \boldsymbol{\omega}^b + \tau_c^b\right) \tag{3.56}$$

Alternatively, with the Unit Quaternion notation, having the equation (3.55a) been replaced by the following one:

$$\dot{\boldsymbol{q}} = \begin{bmatrix} \dot{s} \\ \dot{\boldsymbol{v}} \end{bmatrix} = \frac{1}{2}\begin{bmatrix} -\mathbf{v}^{\mathrm{T}} \\ s\mathbf{I}_3 + [\mathbf{v}]_{\times} \end{bmatrix}\omega^{b} = \frac{1}{2}\begin{bmatrix} -v_1\omega_x^b - v_2\omega_y^b - v_3\omega_z^b \\ s\omega_x^b - v_3\omega_y^b + v_2\omega_z^b \\ v_3\omega_x^b + s\omega_y^b - v_1\omega_z^b \\ -v_2\omega_x^b + v_1\omega_y^b + s\omega_z^b \end{bmatrix} \quad (3.55b)$$

3.2.3 Torques and forces generation

The following schematic (figure 3.23) shows the part of the system which involves motors and propellers, and, in this paragraph, the mechanism of the force generation on the propeller from the motor's torque will be analyzed.

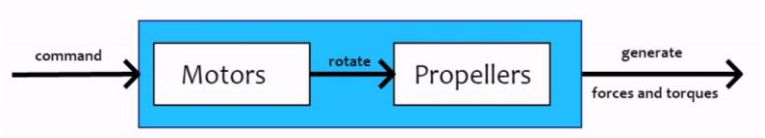


Figure 3.23: Actuators' model [15]

Firstly, the code on electronics (firmware/embedded code) sends a digital command to the motors, and then it is transformed into an analog PWM (Pulse Width Modulation) signal that commands the coreless motors of the quadrotor. This digital command consists of four 16-bit integer numbers (from 0 to 65535), one for each motor.

The physical principles that regulate the movement of the quadrotor (the same principles are valid for every aircraft) are the following ones:

- Bernoulli's theorem: Daniel Bernoulli experimented that pressure inside a fluid, liquid, or gas, decreases as much as the speed of the fluid increases, in other words: "in a moving fluid, the sum of pressure and speed in any point remains constant. "
- Venturi effect: Giovanni Battista Venturi proved experimentally that a fluid particle, passing through a narrowing, increases its speed.
- 3rd Newton's Law: there is always an equal and opposite reaction to each corresponding action.

The concept of lift can summarize the previous principles applied to an apparatus with an airfoil, for

example, a wing or a propeller. If it moves in the air (which has its atmospheric pressure and speed), at a certain speed with a specification upward inclination (called the angle of attack), it will pass through an airflow splitting this into two flows. The one running along the top of the profile will do it with a higher speed than the other, which flows through the lower part (Venturi effect). Moreover, higher speed implies lower pressure (Bernoulli's theorem). Therefore, the surface of the upper wing is subjected to a lower pressure than that inferior.

The airfoil is typically obtained from a body that has a specially designed shape to obtain most of the force generated by the speed and pressure variation of the fluid when it is flowing in an air stream.

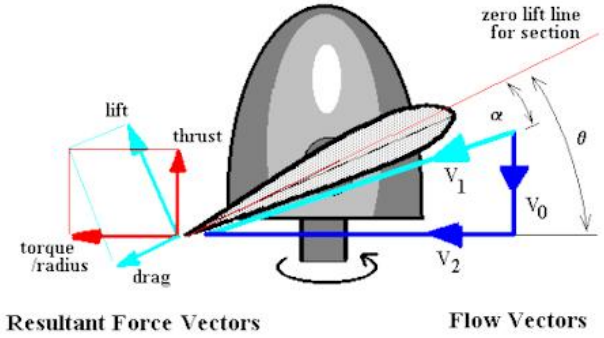


Figure 3.24: propeller model and air propulsion [17]

CHAPTER 4

HARDWARE DESCRIPTION AND FIRMWARE SETUP

4.1 Hardware and instrumentation

The mini drone that was used in this project is the "*Mambo*" model, produced by the French company *Parrot.* This model can be considered a low-cost drone, considering its cost, from 40 to 80 dollars, especially if compared to other models from the same company that can also reach up to 500 dollars.

Figure 4.1: PARROT Mambo Fly minidrone [20] and [22]

The other instruments are the following.

- Safety goggles ensure safety during the flight.
- micro-USB cable used both for charging the mini-drone and for uploading the firmware released from Mathworks to the drone.
- USB dongle compatible with Bluetooth Low Energy used to deploy the model from Simulink to the hardware and make it run in real-time.
- USB dongle drivers usually provided on a CD included with the dongle.
- Additional batteries to be optionally purchased besides the one included with the mini-drone, due to their short duration (about 8-10 minutes of flight)
- Charger.

  It is possible to run the model alternatively also on a PARROT Rolling Spider mini-drone model because it is also compatible with the Simulink Hardware support package.

Figure 4.2: Safety goggles (left) [24], Bluetooth dongle adapter (center) [25], and USB cable (right) [26]

The environment is made by background and a path that must be followed by the mini drone. For the background, a non-glossy backdrop, like the ones of a photo studio, can be put on the ground. White tape was used to create the track such that there is more contrast with the background, and in addition to this, some pieces of paper or other casual objects were randomly put on the background to improve the Optical flow estimation.

Figure 4.3: black background (left and center) [42] and white tape for the track (right) [43]

4.1.1 PARROT mini drones

PARROT mini drones are tiny and cheap quadrotors (4 motors), which can be commanded through a smartphone or a tablet. They are considered among the most stable quadcopters available to buy, because of autopilot on their original firmware. However, this will not be used in the project because the new firmware supports a customized controller in the Simulink model. They are also very safe, and they can also fly indoor thanks to the “cut-out” system, which activates in case of impact and shuts down suddenly the motor. They are provided with a 3-axis gyroscope and a 3-axis accelerometer, a down-facing camera useful for optical flow estimation and image processing, and a SONAR sensor and a pressure sensor for altitude measurements.

4.1.2 Technical characteristics of MAMBO model and main components

Some of the main technical characteristics [20] will be shown below:

- **Dimensions**: 7.1 x 7.1 in.
- **Weight**: 2.22 oz
- **Energy**:

- 660 mAh LiPo battery
- 8 minutes of autonomy with accessories
- 30 minutes of charging with a 2.1 Ampere charger
- 10 minutes of autonomy without accessories

- **IMU sensors**:

- to measure speed and accelerations. The creation of a sensory system capable of perceiving the movement of an object in the surrounding environment is a relevant problem in mobile robotics. Control algorithms are designed from these sensory data, and their availability and reliability are essential requirements. An inertial navigation system (*INS*) is a system capable of acquiring the object navigation information (linear and angular positions and speeds) in its body frame and of transforming it into the fixed (inertial) reference frame.

- **Vertical Stabilization**:

- **Ultrasound sensor**: SONAR (sound navigation & ranging) systems are used in mobile robotics as a low-cost solution for measuring the distance between the robot and the surrounding environment (in the case of this project to measure the distance between the drone and the ground). Their working principle is based on the calculation of the flight time of an ultrasound wave to travel the distance back and forth from the sensor: they are both signal generators and transducers. The quadcopter uses it to measure altitude. It emits a high-frequency sound wave, and it measures the time that the wave takes to reflect on the ground and to be received back by the sensor. From the measured time, the distance between the floor and the drone can be calculated. The maximum altitude that can be estimated is about 13 ft.

- **Pressure sensor:** it is an aid to the ultrasonic sensor for calculating the altitude of the drone. As the drone increases the altitude, the air pressure decreases. The pressure variation can be used to estimate the change in altitude.

- **Horizontal Stabilization**:

- Camera sensor

- **Speed Measurement**:

- 60 FPS (Frame per second) vertical camera

- **SDK (OS of the Firmware)**:

- OS Linux

Figure 4.4: top (right) and bottom (left) view of PARROT Mambo Fly model [20]

On the front of the mini drone, there are the two indicator lights that change color depending on the state of the drone:

- Steady orange: The *Parrot Mambo* is starting up.
- Steady green: *Parrot Mambo is* ready to fly.
- Steady red: *Parrot Mambo* has detected a problem.
- Blinking red: The *Parrot Mambo* battery is running out.

In the upper part of the drone, there is a connector for any external accessories, such as the FPVfirst-person view camera and the clamps.

It is also possible to protect the propellers by installing the hulls on the appropriate supports.

In the rear part, there is the place that houses the battery, the *micro-USB* port, together with an LED indicator that indicates the charging status (Red: charging; Green: 100% battery). With the same USB port, it is possible to communicate the PCB to the computer.

In addition to the on and off button, below the drone, there are several sensors:

- Pressure sensor (black protrusion at the tip of the drone).
- Ultrasonic sensor (Sonar).
- 60 FPS camera: It uses an image processing technique called optical flow to determine the change in shape or translation of objects from a frame and the next one. From these modifications in the position of objects and shapes, the drone can recognize horizontal motion and compute speed.

In figure 4.5, the drone motherboard is shown. It is equipped with the latest SiP6 chipset Parrot with an 800 MHz ARM A9 (in the gray box of figure 4.5 on the right), which has a size of about 12 × 12 mm. This motherboard runs on Linux and is equipped with a pressure sensor, accelerometer, gyroscope, and ultrasonic sensor. It also includes a 60 frame-per-second vertical camera.

Figure 4.5: motherboard of Mambo fly model [27]

The 6-axis motion detection device (Figure 4.6) is an **MPU-6050** from the US manufacturer *InvenSense*. Inside this, there is a *Digital Motion Processor* (DMP) interacting with a three-axis accelerometer and a three-axis gyroscope. The device dimensions are about 0.16 × 0.16 × 0.035 inches. This device is mounted just above the processor (in the blue box of Figure 4.6 on the left).

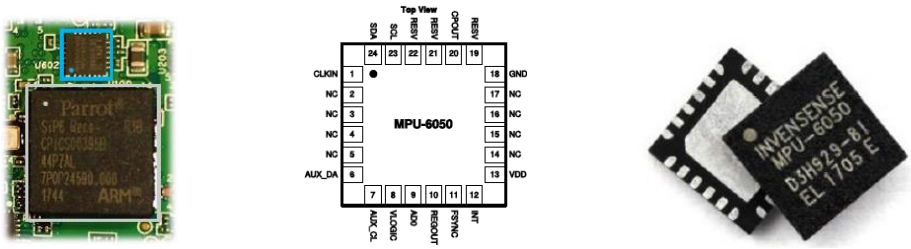


Figure 4.6: IMU with gyroscope and accelerometer [28] and [29]

The axes of the motion sensor (Figure 4.7) are not concordant with the axes of movement of the drone (figure 5.32 in the next chapter).

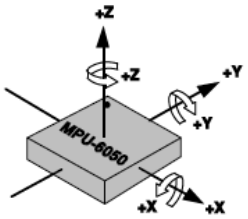


Figure 4.7: IMU reference frame [29]

To conclude this brief overview of the components of the *Parrot Mambo,* the motors and their propellers are shown (Figure 4.8). An important feature to classify these motors is their spin direction.

It can also be noticed that the drone propellers are of two different colors: the white ones are the ones mounted in the front while the black ones are in the back.

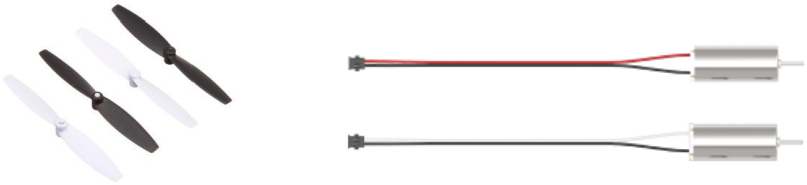


Figure 4.8: propellers (left) and coreless motors (right) [23]

The drone has two different types of motors (Figure 4.8 on the right):

- Motor A (Counterclockwise) is identified with a black and white wire together with a circular sign black printed on the upper part (where the rotation axis comes out), and it is located on the front right and rear left parts.

- Motor C (Clockwise) is identified with a red and black wire, and it is located on the front left and right rear parts.

Both motors are *Coreless* type and dimensions of 8.5 × 20 mm. Similarly, there are two types of propellers, identifiable by the same word "C" (clockwise) and "A" (counterclockwise):

- Front left and rear right: "C".
- Front right and rear left: "A".

These plastic propellers have been designed to minimize the electrical consumption of the engines and, at the same time, ensure maximum thrust to the engines.

To better understand what a *Coreless* motor is, it is useful to briefly describe a typical DC motor with a collector.

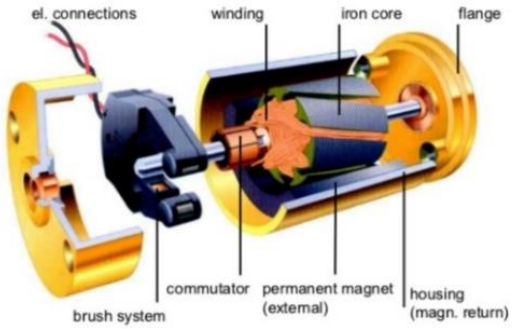


Figure 4.9: DC motor structure and components [30]

It consists of several main parts:

- Stator: typically composed of a permanent magnet.
- Rotor: composed of a set of ferromagnetic plates, in the shape of a circular crown where the various copper windings are wrapped.
- Collector: it is a set of blades fixed on a cylindrical drum to the coil terminals.
- Brushes: made of graphite. They remain in contact with the collector even when it is in rotation, and they give the main power to the motor.

In *Coreless* motors (Figure 4.10 on the left), there are all the components of the previous motor, except for the core of laminations in the rotor. The approach used to describe them is adapted from Karnavas et al. [22].

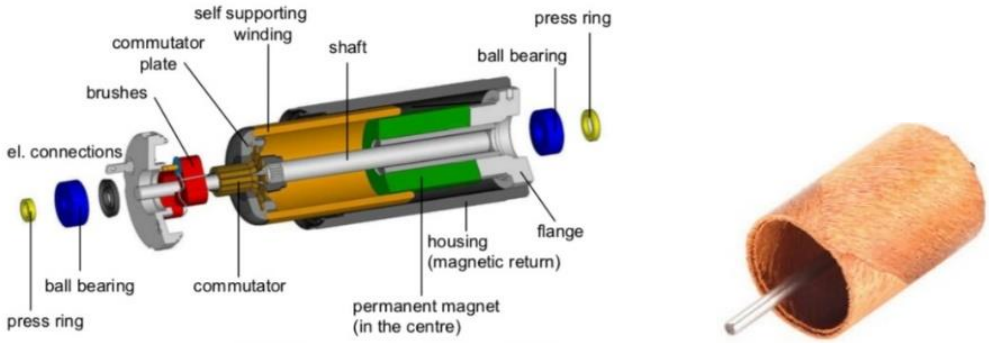


Figure 4.10: Coreless DC motor structure [30] and rotor winding [29]

The rotor winding (Figure 4.10 on the right) is wound obliquely, or honeycomb, to form a self-supporting hollow cylinder. Since there is no rigid structure that supports it, the windings are covered with epoxy resin.

The stator which generates a magnetic field is located inside the rotor and is composed of rare earth magnets, such as Neodymium, or Al-Ni-Co (Aluminum, nickel, cobalt), or Sm-Co (Samarium, cobalt).

The *Coreless* motor usually has tiny dimensions: between 6 millimeters to less than 75 millimeters in diameter. Their power limit, in general, is around 250 watts or even less.

The following properties make Coreless motors valid solutions for different applications:

- Coreless rotor technology ensures smooth operation (regular dynamics)
- The rotor has lower inertia than iron-core DC motors, and this leads to more significant accelerations.
- The absence of iron losses implies higher efficiency (about 85% compared to 50% of the iron-core ones), longer life, and less overheating.
- Lighter than iron-core DC motors.
- At low speeds, the starting voltage is lower (0.3 V).
- Faster dynamical response (low mechanical time constant, about 10 ms.
- Longer life of the brushes and the collector because of a reduced inductance.

4.2 SIMULINK Hardware support package and firmware setup

In this paragraph, the procedure to install the firmware on the drone from the Hardware support package of Matlab will be explained. The firmware released by Mathworks makes it possible to deploy the model from the Matlab project to the drone board and run it through Bluetooth connection.

The material needed for this operation is the following:

- Parrot Mambo drone (with the charged battery inserted).
- USB- micro-USB cable.
- Bluetooth USB Dongle.
- Bluetooth USB CSR driver already installed.
- Add-ons: "Simulink Support Package for Parrot Minidrones" installed.

The following method explains how to make the drone communicate with the PC via Bluetooth in a Windows 10 operating system. From the main MATLAB window, open the *Add-Ons Manager* (Manage Add-Ons) and start the procedure by clicking on the gear icon. Connect the drone with the USB cable (Figure 4.11) and wait a few moments for it to be recognized by the PC. Click *Next* and wait for the recognition of the drone. Press *Next* again, the firmware file will be loaded into the drone's memory.

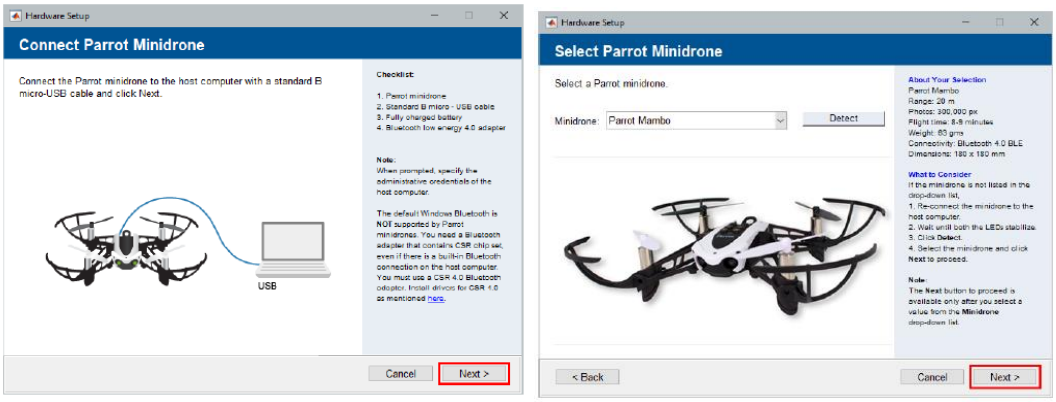


Figure 4.11: recognition of the drone through the USB cable [12]

It is necessary to disconnect the USB cable from the drone to start the setup procedure (figure 4.12). During

installation, the front LEDs will flash orange. Once completed, the LEDs will flash green (make sure that for at least 10 seconds, this last state occurs to proceed with the Setup).

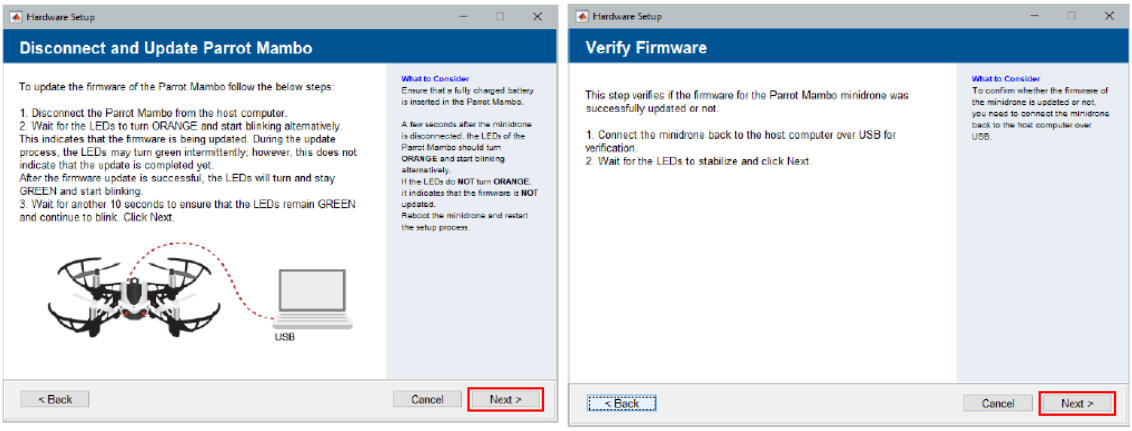


Figure 4.12: Firmware setup [12]

Reconnect the USB cable to the drone (wait a few moments for it to be recognized) and proceed by clicking *Next* (Figure 4.13).

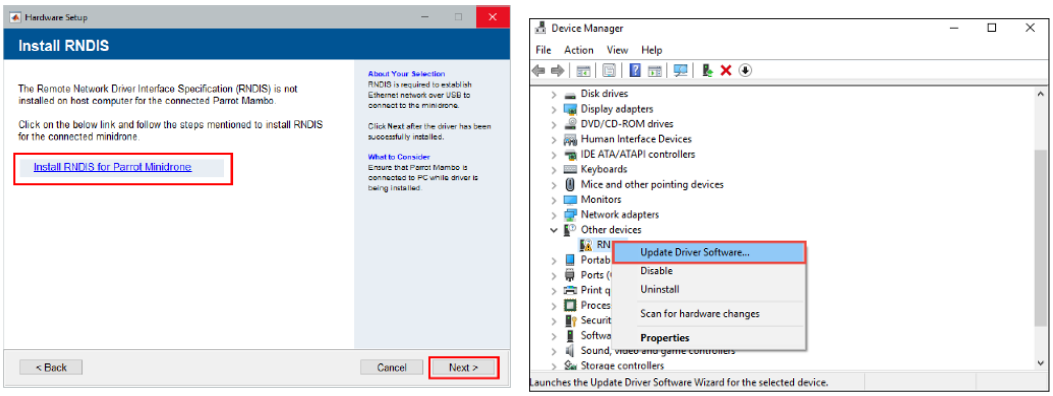


Figure 4.13: RNDIS driver setup (1) [21]

Before doing *Next,* the following procedure must be completed to install the RNDIS driver (figure 4.14).

Open "*Device Manager"* and click on "*Other devices"*. From the list, select "*RNDIS"* and "*Update Driver Software..."*.

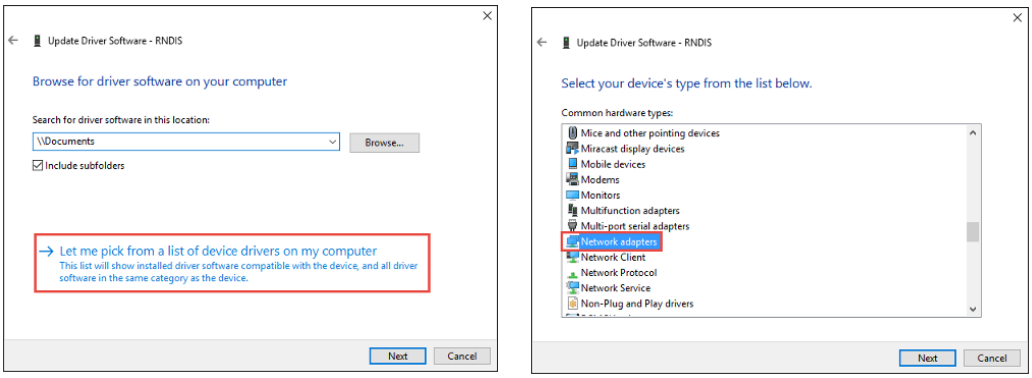


Figure 4.14: RNDIS driver setup (2) [21]

In the new screen, the driver software must be picked from the list of devices. Among the "*Common hardware types"*, select "*Network adapters*".

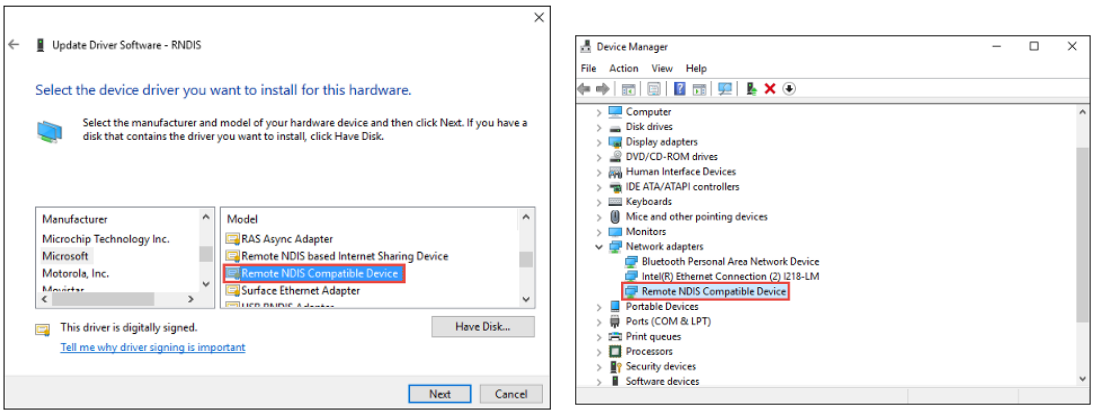


Figure 4.15: RNDIS driver setup (3) [21]

Look for "*Microsoft*" among the *Manufacturers*, "*Remote NDIS Compatible Device*" among the *Models* (figure 4.16). In the "*Update Driver Warning"* dialog box, select "*Yes"*. Once installed, the RNDIS device can be found among the devices inside "*Network adapters"*. Now, it is possible to continue with the Setup procedure.

By clicking on *Next* (Figure 4.16), a new window shows up, where there are instructions to connect the drone via Bluetooth. First, you must disconnect the USB cable from the drone again.

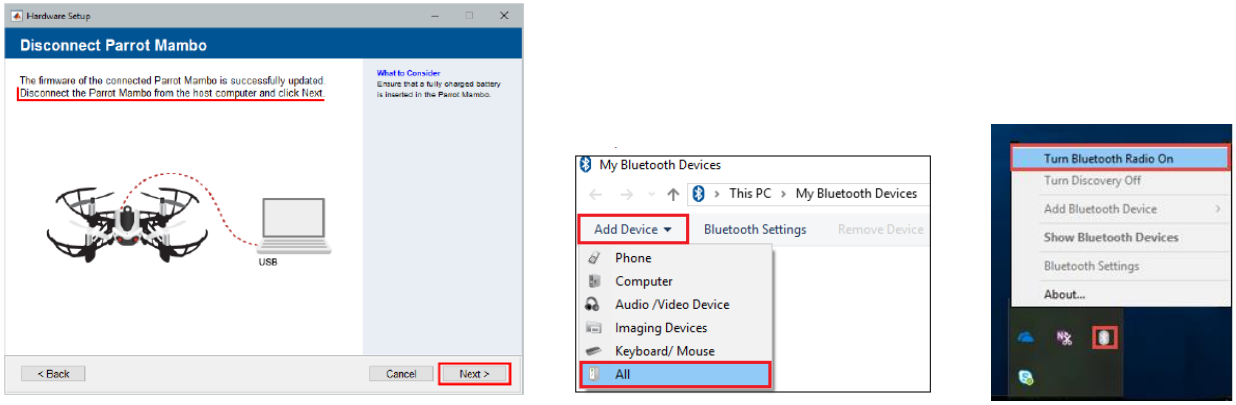


Figure 4.16: firmware checking (left) and Bluetooth radio device setup (center and right) [20] and [44]

Now, from the *file explorer* window, click on "This PC" and then double click on "My Bluetooth devices". Then, open the "Add device" dropdown and select "All".

The search for Bluetooth devices starts. The name of the device to be connected is called "Mambo" followed by several numbers (relative to the hardware serial number). It is necessary to choose the one that has the icon of a Joystick (figure 4.17); otherwise, the communication between MATLAB and the drone will not be possible.

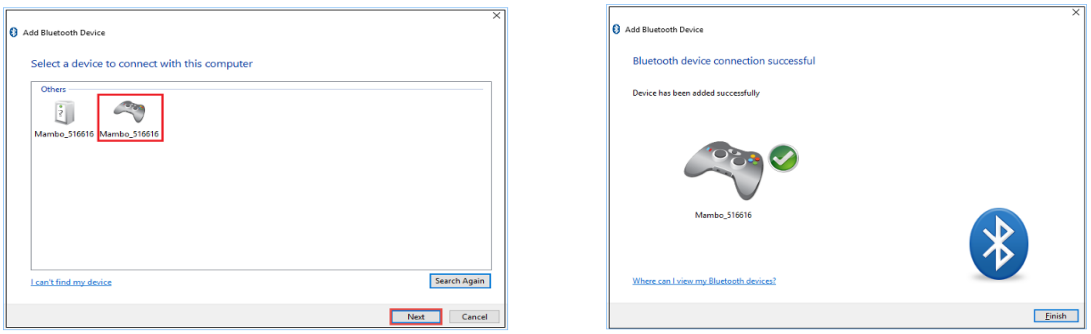


Figure 4.17: Bluetooth pairing [44]

Once *pairing is complete*, double click on the device just added (Figure 4.19), and after pressing with the right text above the icon accompanied by name *Personal Area Networking (NAP)*, click *Connect*.

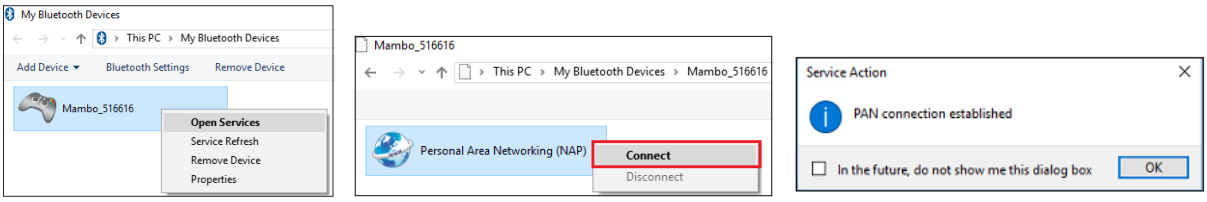


Figure 4.18: PAN connection setup [44]

Now press *Next* (Figure 4.19), and in the next screen, click on "Test connection" to ping to verify that the communication occurs.

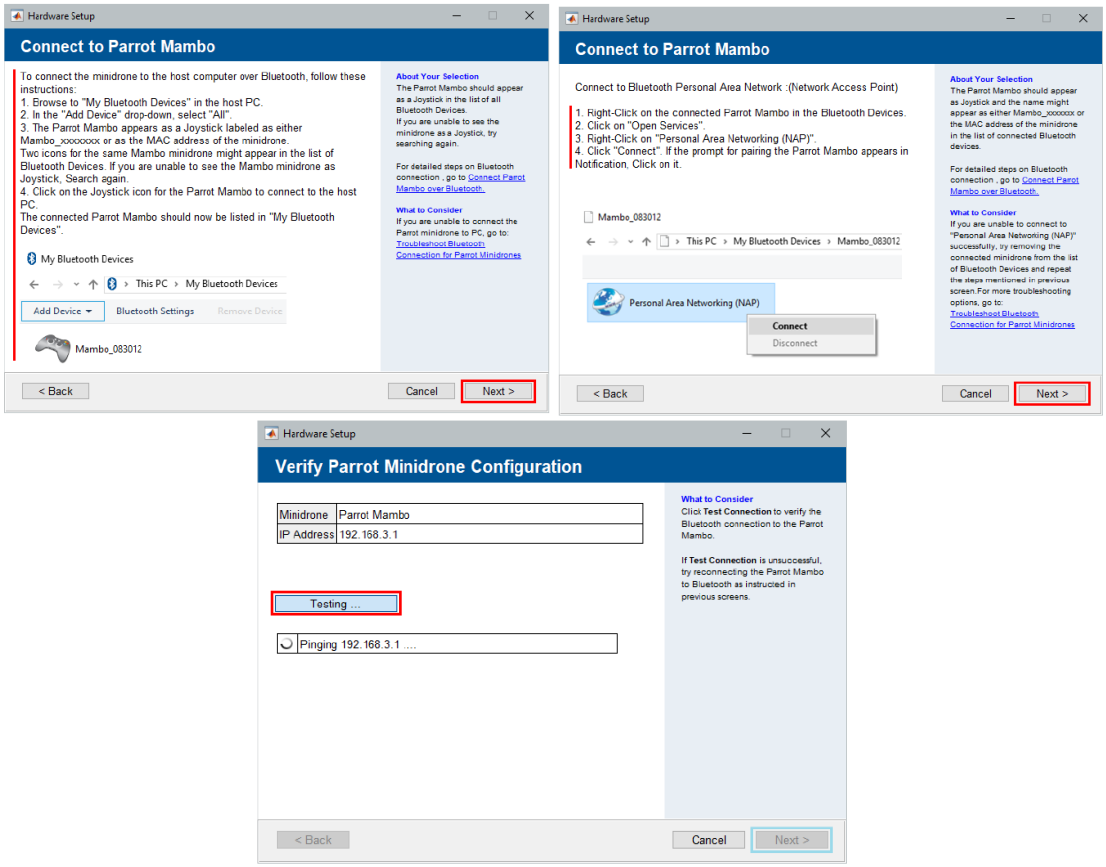


Figure 4.19: Bluetooth connection test [12]

Once completed and successful, click *Next,* and then *Finish* to end the Setup. This procedure only needs to be done once. If we want to connect the drone other times, it must be done via Bluetooth with PAN (Personal Area Network) in the services of the device, keeping the Bluetooth dongle plugged in the PC and turned on.

CHAPTER 5

SOFTWARE TOOLS DESCRIPTION AND MODEL CONFIGURATION

5.1 Description of Software Tools used in the MATLAB environment

For this Thesis project, the leading software and tools used are MATLAB 2019b, Simulink, Stateflow, and some of their add-on tools like Aerospace Blockset, Simulink 3D Animation, which lets us visualize simulations in 3D, the Simulink Hardware Support Package for PARROT Minidrones, and Simulink Coder. Simulink is an extension of Matlab that allows us to build control systems models with a graphical representation through blocks, to simulate them, and to adopt Model-Based Design technique. It provides automatic code generation, and hardware test and validation. It also offers solvers for dynamical systems and customizable block libraries.

MathWorks developed the Simulink support package for PARROT mini drones for various purposes. One is to make people interested and aware of the influence of Model-Based Design in modern engineering applications. Another reason is to assist developers in the industry in acquiring Model-Based Design techniques through an innovative application. Finally, to support instructors and professors in training students on this design method through drones as an admired hardware platform [18].

The add-on tools and toolboxes used in Matlab 2019b environment are the following:

- Simulink Hardware Support Package for Parrot Minidrones: used for firmware setup, for the connection of the drone via Bluetooth, and algorithm deployment through MATLAB. It allows us to employ different onboard sensors for creation, simulation, and test of flight control algorithms. Through this package, we can have a low-cost and ultra-compact drone to conduct experiments with its control system.
- Aerospace Blockset: for modeling, simulation, and analysis of aerospace vehicles. It provides blocks for

modeling and simulation of crewless airborne vehicles, propulsion systems, aircraft, and spacecraft, that are subjected to environmental and atmospheric conditions.

- Optimization Toolbox: a toolbox for solving linear optimization problems, quadratic, integer, and nonlinear.
- Signal Processing Toolbox: for the processing and analysis of signals.
- Simulink 3D Animation: it allows us to visualize the behavior of dynamic systems in a virtual reality environment.
- Simulink Coder and Matlab Coder: C++ automatic code generation from Simulink models and execute them. We can use the generated source code in non-real-time and real-time applications like rapid prototyping, simulation, and HIL (hardware-in-the-loop) tests. It can also be used to record flight data.
- Embedded Coder: C ++ code generation for embedded processors (Extension of Simulink Coder).
- Simulink Control Design: model linearization and control systems development.
- Image Processing Toolbox: to use the App Color Threshold.
- DSP System Toolbox: Design/Simulation of signal processing systems.

5.2 Software configuration, Simulink project overview, and Simulation Model description

The project should be configured to work on the model design, and for this purpose, the project environment of the Mathworks mini-drone competition organized by Mathworks can be used by entering the command window the word "**ParrotMinidroneCompetitionStart**".

Firstly, the project environment must be configured, and once done it, three windows appear:

- Tree model with all the subsystems and the variables (figures 5.2) of the project named "ParrotMinidroneCompetition".
- Mini Drone simulation model (figure 5.1).
- Simulink 3D Animation: visualization of the drone dynamics during the simulation (figures 5.3)

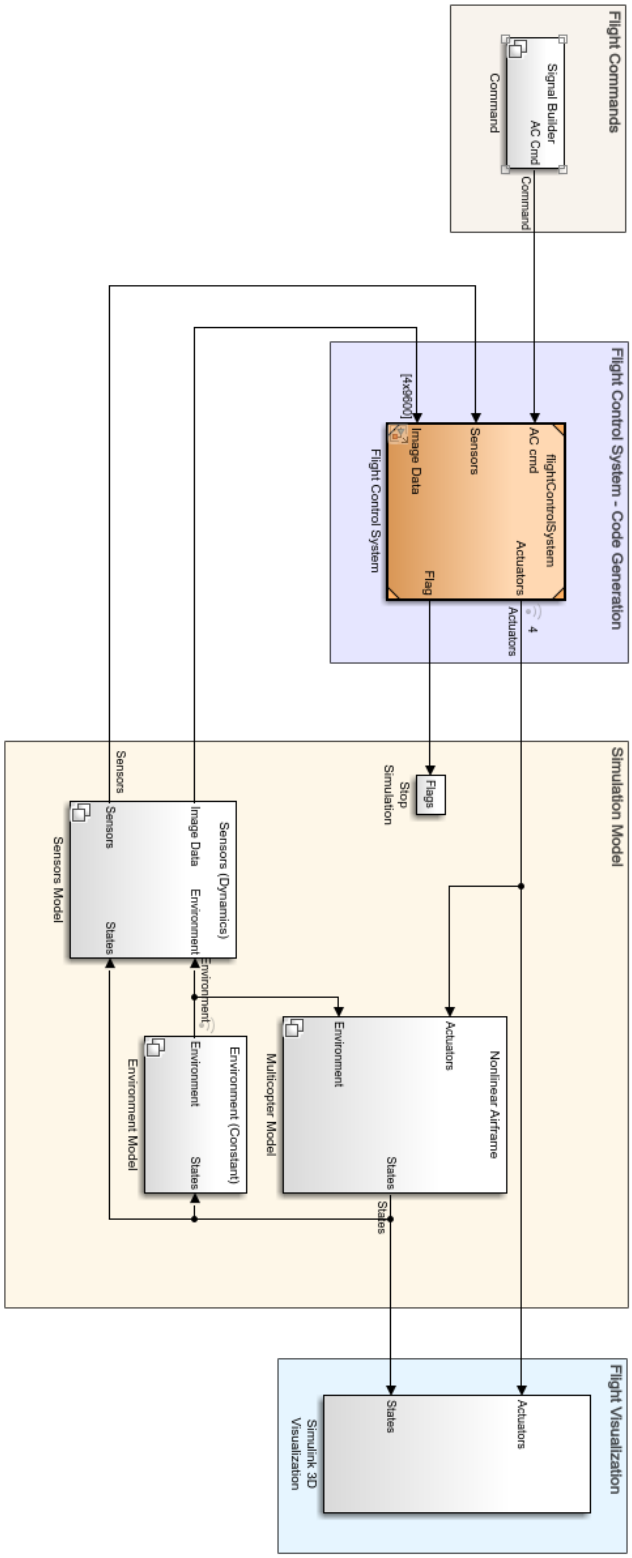


Figure 5.1: The Simulation model

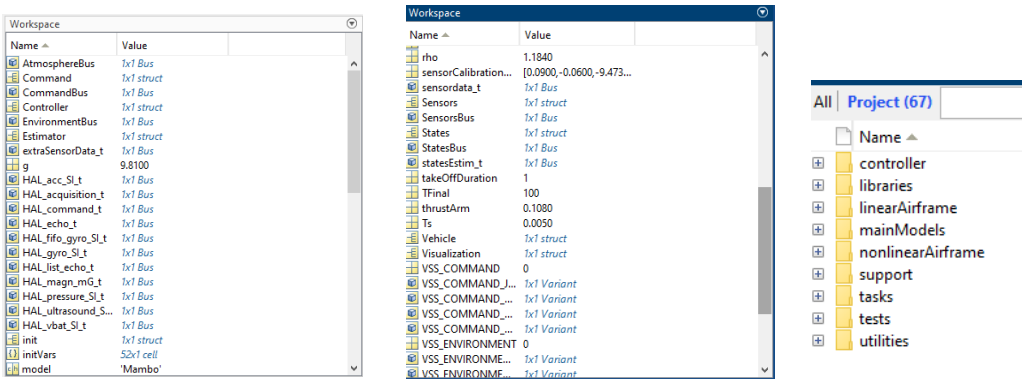


Figure 5.2: workspace with the variables and constants (left and center) and project folders (right)

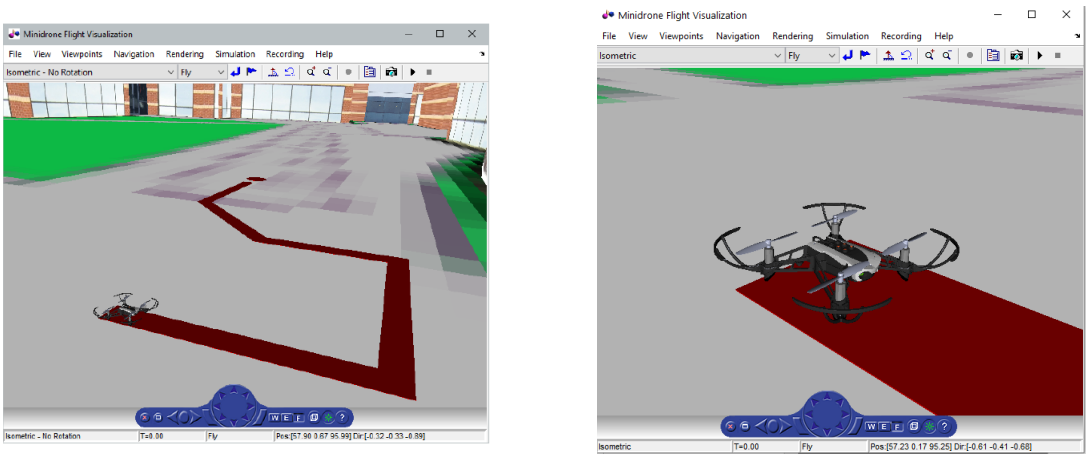


Figure 5.3: Simulink 3D environment used for simulation

Within the project framework, we can organize user-defined tasks related to this competition, files, and settings. The simulation model (figure 5.1) is constituted by six main blocks that embrace the mathematical descriptions of the relative dynamical subsystem: the airframe, the environment, the flight control system (FCS), the sensors, and the subsystems used to show a visualization output or to feed input commands during the simulation.

Four of the six subsystems can be defined as variant subsystems because they allow us to choose between different properties of the subsystem. However, the FCS (Flight Control System) block (figure 5.4) should not be considered as a variant subsystem, but as a modeled subsystem, because its elements are referred to another Simulink model. The FCS subsystem will be explained more deeply in the next chapter.

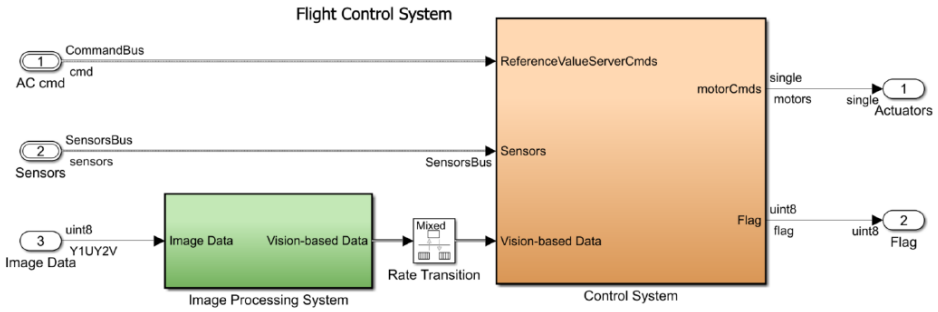
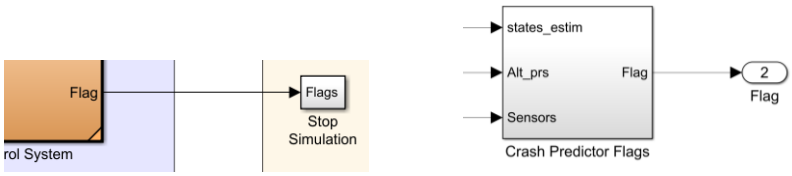


Figure 5.4: Flight Control system

Furthermore, other blocks in the Simulation model are employed to set the simulation pace. Instead, the "Flag" subsystem is used to terminate the simulation if an unwanted flight situation happens (figures 5.5, 5.6, and 5.8).



Figures 5.5: Flag block

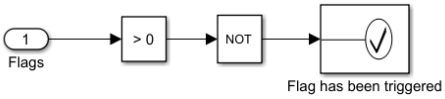


Figure 5.6: Flag subsystem (1)

The "play" icon must be pressed to run the simulation. The system dynamic behavior can be observed for the time defined through "TFinal" variable, and then can be stopped (figure 5.7 on the left). The "set pace" option can be used to modify the speed of the simulation (figure 5.7 on the right).

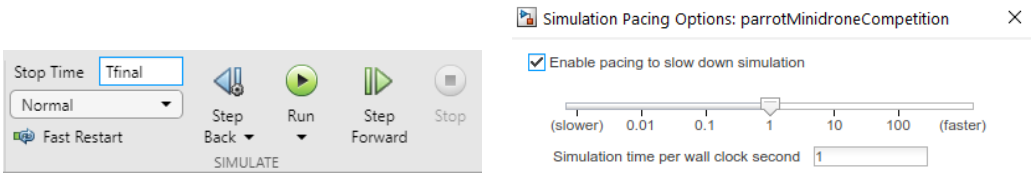


Figure 5.7: Simulation commands (left) and pace settings (right)

For example, the simulation can be run tenth times slower, and the sample time can be increased to 0.1 ms to see the mini drone moving at slow-motion. Once it runs, the mini drone 3D model takes off and hovers.

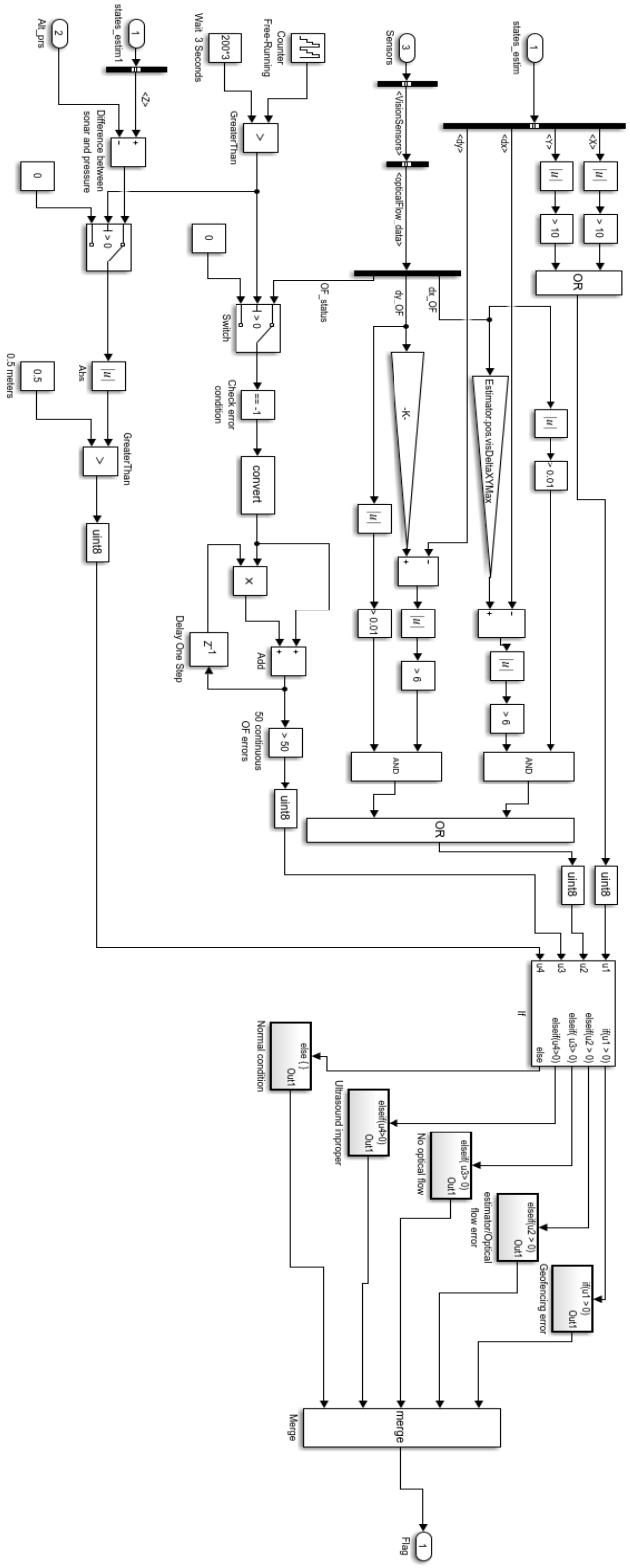
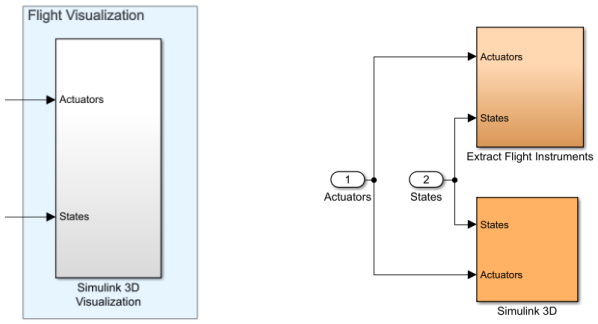


Figure 5.8: Flag subsystem (2)

Other options can be found by double-clicking on the “Flight visualization” subsystem. The signals can be output as shown on a cockpit display, provided by Aerospace Blockset toolbox, with some standard flight instruments like heading, percent RPM, airspeed and climb-rate indicators, and altimeter. The “extract flight instruments” (figure 5.9) subsystem contains the applicable states.



Figures 5.9: Simulink Visualization block

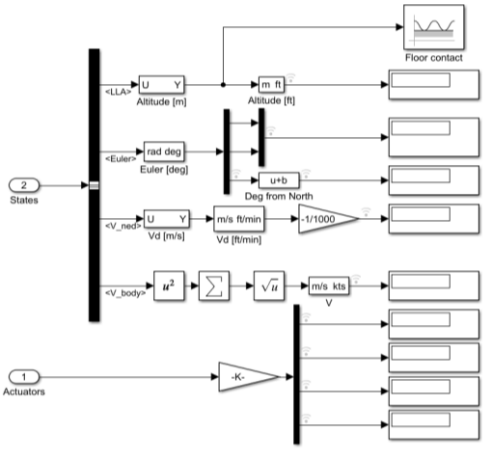


Figure 5.10: Simulink Visualization subsystem

The “Command” subsystem can be used to change the mini drone input signals in the simulation (figure 5.11). Four variant subsystems constitute this one: data input, Signal builder, joystick, and spreadsheet file reading.

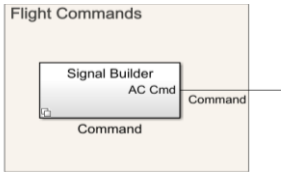


Figure 5.11: Commands subsystem

Signal builder is set as default in the command subsystem, but, in this project, the signals driving the drone come from the path planning subsystem in the Flight controller, which receives the data from the image processing subsystems. The mode selected to drive the drone is according to position coordinates X-Y-Z and yaw, but alternatively, it can be driven by pitch and roll signals coming from the Image processing subsystem. However, in this last case, it would become more complex because the camera orientation has to be ensured to point right to the track, and this would be easily solved by using position coordinates rather than pitch and roll angles. In the Model-Based Design method, the simulation model is the focal point, because it aids to improve the control development before deploying it on the hardware, therefore avoiding damages and crashes.

A part of the FCS subsystem is adapted from the model of Professor Sertac Karaman, and Fabian Riether developed at MIT. Since the project aims to design a flight controller able to track the path while keeping stable itself, the controller subsystem is designed tuning the gains of the PID controllers, and the image processing and path planning subsystems are designed according to our project goal.

Finally, once the FCS is developed and tuned through simulations, the code can be generated and tested on the mini-drone hardware.

5.3 Compiler configuration

To generate the C ++ code to be loaded into the aircraft, the exact compiler must be chosen. What by default it does not allow the download of MAT files from the drone, after each flight simulation.

This procedure can only be performed from the model window opened automatically after generating the code with the "Generate Code" button. Go to the "Modeling" bar (Figure 5.12) and click on the gear icon. It will open a window, shown in Figure 5.13.

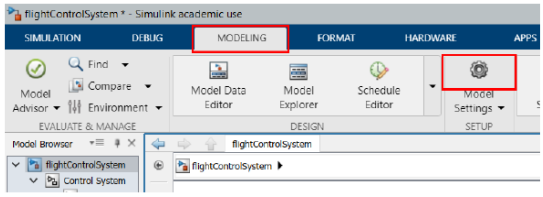


Figure 5.12: Modeling bar [12]

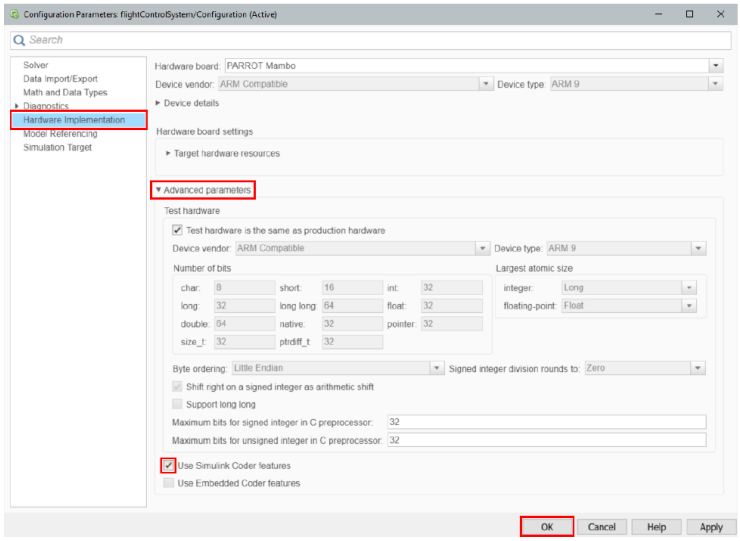


Figure 5.13: Model settings [12]

Click on "Hardware Implementation", then move the mouse over the three dots to bring up the word "Advanced Parameters", click on it to show the menu, then tick the "Use Simulink Coder features" option. Complete the configuration by pressing "OK".

5.4 Preliminary Test of the drone motors

Before performing any flight tests, it is possible to run a test in which the propellers are only moved without generating the minimum thrust to flight (figures 5.14). In this test, the model spins the motors 1 and 3 for 2 seconds and then the motors 2 and 4 for the other 2 seconds. The test model is a template included in the package, and it can be set up by entering the command “parrot_gettingstarted” in Matlab and by selecting “Deploy to Hardware” (figure 5.16).

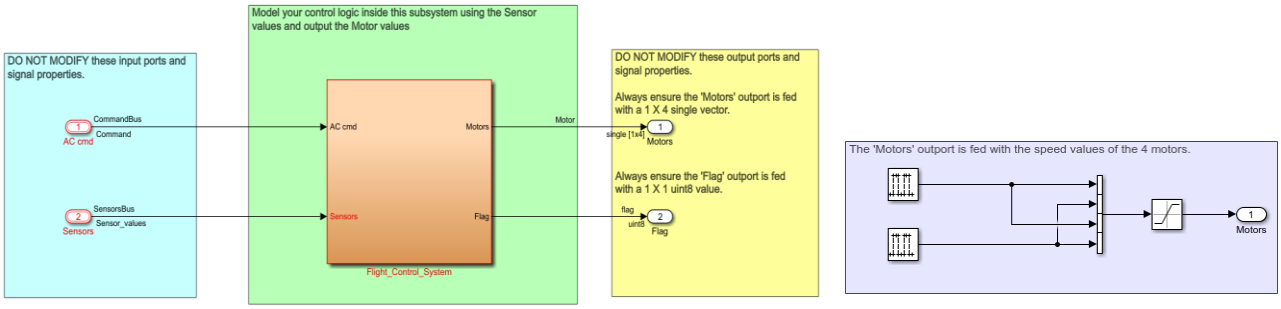


Figure 5.14: FCS subsystem for the actuators test

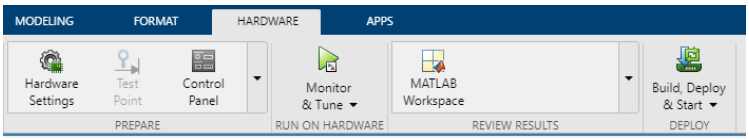


Figure 5.15: Hardware functionalities

The window called “diagnostic viewer” (figure 5.16) allows us to follow and to troubleshoot the compilation, the file transfer, the code generation (figure 5.17), and the execution. There is another way the model can be executed: the external mode (“Monitor and Tune”). Through this execution method, some values can be updated from the block diagrams throughout its execution, besides building and deploying the model on the hardware. It is particularly useful during the test of the image processing algorithm because the real-time visualization of the images acquired from the downward-facing camera allows us to tune the threshold filter for path detection.

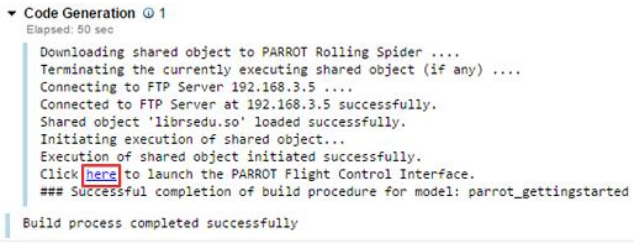


Figure 5.16: Code generation [12]

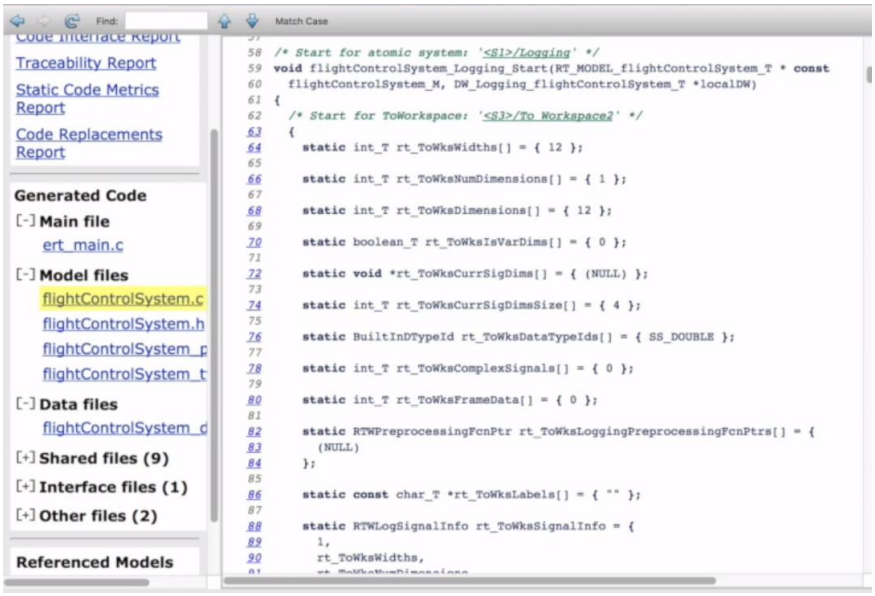


Figure 5.17: C code of the model [12]

After the “diagnostic viewer” confirms that code generation has completed, the model can be run on the hardware by opening the flight control user interface from the diagnostic viewer, and the flight time can also be set as the simulation time (figure 5.18). At this point, to prevent an uncontrolled flight, we should be careful not to increase the power gain of the thrust of the propellers (it is recommended to begin by setting the gain at the 10% of the rated one).

In this test, signals are fed to the motors to ensure that the Bluetooth communication and the toolchain are working correctly.

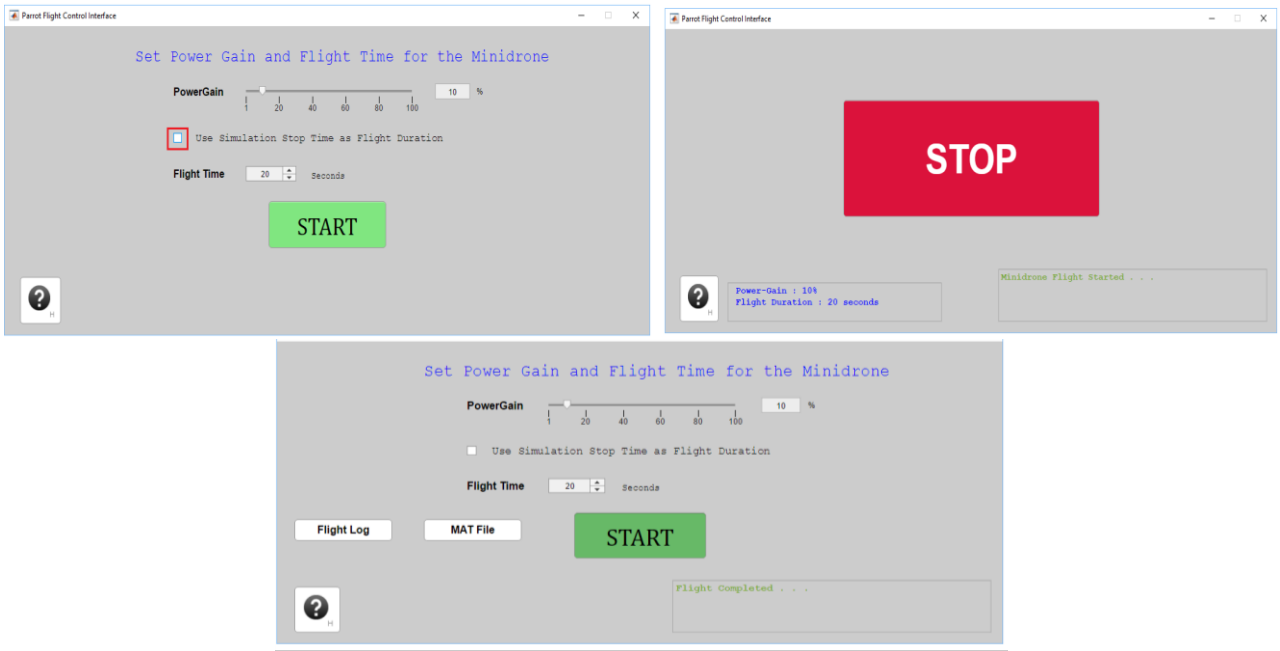


Figure 5.18: Flight control interface with flight time settings, stop and end of the test [12]

If we click on “Stop”, the test can be interrupted, and the flight log and the MAT file with sensor data can be downloaded to the working directory.

It is useful to remember that the hardware target must be set up correctly (in our case “Mambo” model) before deploying the code, and data logging must be enabled to take advantage of Simulink Coder capabilities to capture sensor and controller commands signals for inspection. They can be used to plot trajectory, motor output, sensor signals, altitude, velocities, orientation, position, optical flow velocities, and percentage of

battery charge. The data that can be stored by the mini drone during flight test is limited, and memory allocation can be controlled through the flight time variable "*TFinal"*. The model cannot be executed if there is an excess of memory needed beyond the hardware capacity.

Furthermore, it is possible to list other important guidelines:

- The space where flight tests are conducted must be large at least 20 feet by 20 feet by 10 feet high to avoid damages to the mini drone, the environment, and any observer.
- Small rooms and specific ground materials (like a carpet) could cause flight stability issues because of bouncing or absorption of ultrasound signals.
- Optical flow estimation algorithm, image processing algorithm, and stability during flight can be affected by lighting conditions and shapes on the ground.

5.5 Physical characteristics of the Quadcopter model from Aerospace Blockset

The following picture (5.19) shows the quadcopter model with its body reference frame, taken from Aerospace Blockset.

Figure 5.19: reference frame of Quadcopter model [31]

The origin of the mini drone reference frame coincides with its center of mass. "*VehicleVars*" folder contains the mass and inertias in the Simulink project.

Body Frame:

- The *x*-axis is oriented in the forward direction.
- The *y*-axis is oriented to his right.
- The *z*-axis is oriented down.

Motors directions convention:

- Motor 1 spins positively relative to the *z*-axis, and its position is at -45° from the *x*-axis.
- Motor 2 spins negatively relative to the *z*-axis, and its position is at -135° from the *x*-axis.
- Motor 3 spins as motor 1, and its position is at 135° from the *x*-axis.
- Motor 4 spins as motor 2, and its position is at 45° from the *x*-axis.

The approach used to define the convention is the same as expressed by Prouty [32] and by Ponds et al. [33].

CHAPTER 6

SIMULATION MODEL DESCRIPTION AND CONTROL PROBLEM SETUP

6.1 Nonlinear and Linear models in Model-based design approach

In the design flow, nonlinear and linear models are both needed for designing and testing the flight control software. The following method is the one treated by Douglas [41].

In figure 6.1, the upper block is the flight control software that represents all the control system software. This code must interface with the rest of the mini-drone firmware, and it has two inputs, the raw sensor readings and the reference commands or set points, and two outputs, the motor speed commands, and the stop flag. The reference commands are put inside the flight code. The lower block called 'Nonlinear model' represents everything else, anything that is not the flight control code. This includes the rest of the mini-drone firmware, the hardware, the atmosphere flying in, etc. At an elementary level, the model inputs motors commands, and the stop flag, then it makes a few calculations and outputs sensor measurements. In this way, the model is wrapping around the flight code and provides the feedback loop. If the model were so accurate that it perfectly represents reality, it would be indistinguishable whether the results came from the actual hardware or they came from this perfect model. Therefore, the mini drone performance could be simulated using the model, and we can be very confident that when the flight code is run after on the actual hardware, it would have the same result.

However, a perfect model of reality is impossible to create, and it is not necessary to model everything. The trick is to figure out what to include in the model and what to leave out. Some of that knowledge comes easily by just understanding this system and how it will be operated. For example, the code that turns on and off the front LEDs does not need to be modeled. They will not impact the control system.

Nevertheless, there are a lot of other things that are not as obvious and knowing what to model requires a

little experience and investigation. One example is whether to model the airframe structure as a rigid body or as a flexible body. It is hard to precisely know what to model and what to leave out initially. Usually, what happens is starting with the best guess, and then over time, the fidelity of the model will grow until the match between the experimental results and simulation are satisfying. Therefore, simulation is a way to verify the system in hard situations or time-consuming to physically test, as long as the model adequately reflects reality. The excellent advantage of simulation is that the model can be reset quickly, and the drone can be put in any situation of interest, and if it performs poorly, we make the necessary changes, and we do not damage any of the hardware in the process.

However, a model is also used for control system design, also with the linear analysis tools. Unfortunately, the previously created nonlinear model used for simulation does not lend itself to a suitable design and linear analysis. Thus, we also need a linear model. Essentially, we should remove the nonlinear components in the model or estimate them as linear components. The linear model will not reflect reality as accurately as the nonlinear model, but it should still be accurate enough that it can be used to design the controllers.

Summarizing, two different models are considered: a lower fidelity linear model that is useful for determining the controller structure and gains and a higher fidelity nonlinear model that is useful for simulating the result and verifying the system. The approach used in the method description

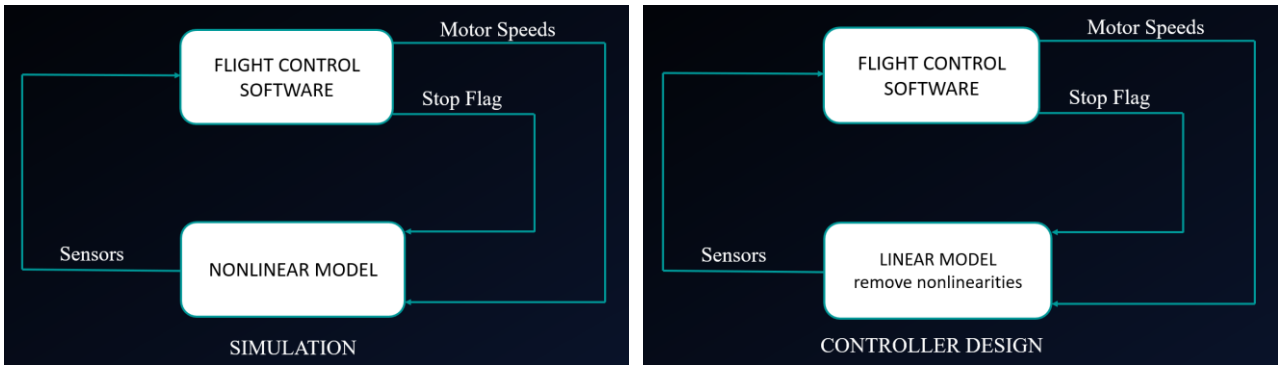


Figure 6.1: Schematics of Feedback Control loop with a nonlinear model for simulation (left) and with a linear model for controller design (right) [41]

To summarize, the steps taken to design the flight control software using Model-Based Design techniques are the following:

1. Create a high-fidelity model of everything the flight control software needs to interact with. This very likely will be a nonlinear model.
2. Verify that the model matches reality with several test cases.
3. Once had a model that reflects reality, create a linear version.
4. The linear model and our linear analysis tools are used to design and analyze our control system.
5. The nonlinear model is used to verify the performance of the control system.
6. The flight control software can be run with enough confidence in the actual hardware for final verification.

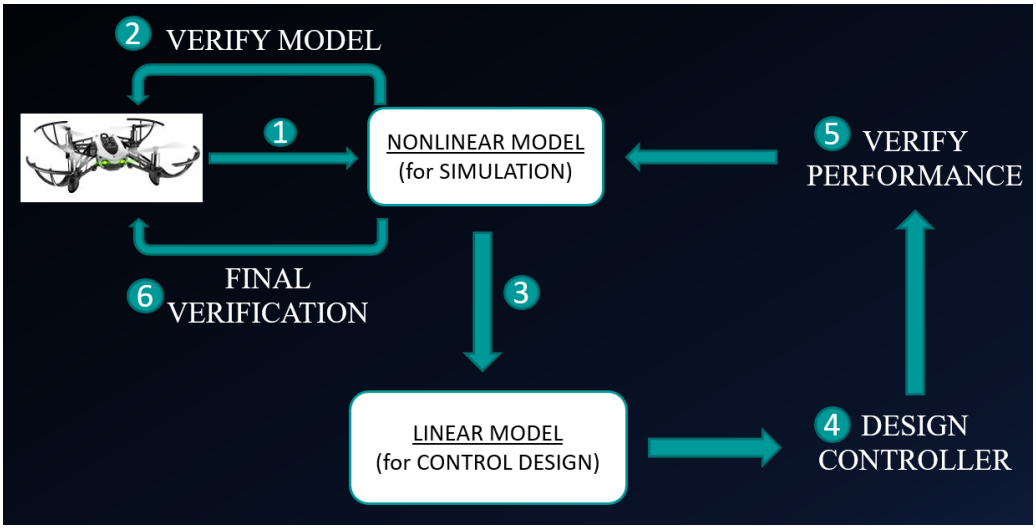


Figure 6.2: Model-based design technique schematic [40]

Instead of thinking about the plant model as a single block or as a monolithic set of calculations, it is generally easier to break it up into several smaller models that represent specific and discrete systems.

For the mini drone, it might be broken up into the airframe structure, the actuators, the environment, and the sensors (figure 6.3). Then, within these models, there are even smaller subsystem models, like the gravity model or the IMU model.

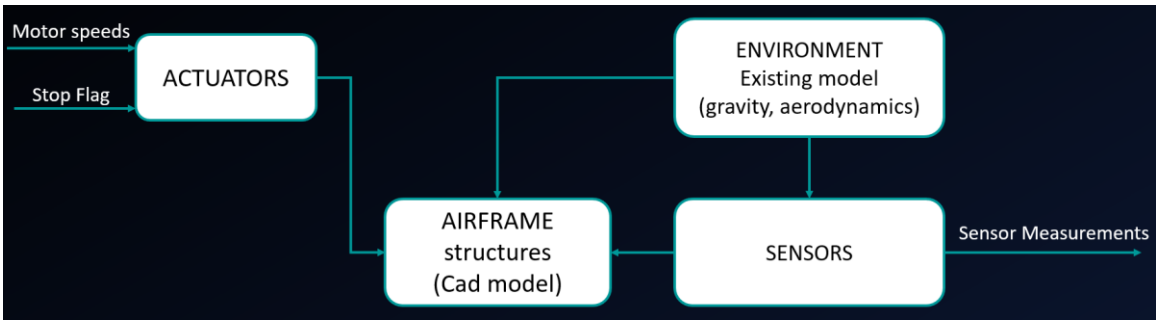


Figure 6.3: Components of the airframe model [40]

There are many reasons to approach modeling in this hierarchical way, rather than lumping all the plant components together into a single model:

- For instance, it allows multiple people and teams to build different parts of the model simultaneously.
- Portions of the model can be upgraded, based on which area needs more fidelity, without impacting the rest.
- Each sub-model can be developed using different modeling techniques. It can be chosen as the modeling technique that makes the most sense or is the easiest for each system. Then when they are put all together, there will be an entire model that can be used for wrapping around the flight control software and simulating the results.

6.2 Control system architecture (Hovering control)

In this chapter, it will be described how to design a control system architecture to control the quadcopter during the flight maneuvers needed to accomplish the task. The first goal is to control hovering. It should be ensured it is stable while hovering, given a three-dimensional reference, X-Y position, and altitude, which in this task are all constant. While tracking the path, the mini-drone will have only a change of reference in the X-Y position, which is generated and continuously updated from the Path planning algorithm. If we assume that during these motions, the mini drone should maintain the camera faced down almost orthogonally to see the track with the right orientation, then there will be low pitch and roll angles, which means that the hovering

control system architecture can be used for the path tracking task as well. The control system is designed according to the approach of Douglas [19].

In this architecture, the mini drone is considered as the plant. It receives as inputs the four motor speeds, spinning the propellers and generating forces and torques as plant outputs. The control goal requires the mini drone to hover at a specified altitude. Therefore, the four motors should be controlled autonomously to obtain this output. The motor mixing algorithm (MMA) can be used to control yaw, pitch, roll, and thrust directly, rather than controlling in terms of the four motor speeds.

Thrust is always oriented along the Z-axis of the body frame. If the drone flies at steep roll or pitch angles, then, if the thrust is commanded, there is a coupling between the altitude variation and the horizontal motion. If a controller for a racing drone must be built, this coupling must be considered because it could fly at steeper roll and pitch angles. However, for this simple hover controller, low roll and pitch angles should be assumed. Therefore, a change in thrust only significantly influences the altitude rate.

Firstly, we should think of adjusting the altitude through a PID controller that commands the thrusts. The drone altitude is the state to measure and to feedback, comparing it to the reference. Then the altitude controller determines how to command the thrust through the obtained error. If, for example, the mini drone hovers at a lower altitude, the error becomes positive, and the thrust command is split evenly among the four motors, and the drone rises. However, the effect of disturbances, like wind gusts, can cause the drone to roll or pitch a little, and so the thrust could also induce to move horizontally, making the drone drift away from the reference position. Therefore, the controller is not suitable, and it can be improved through the roll, pitch and yaw angles control, keeping them constant at zero degrees, so that thrust can only impact altitude, preventing the drone from wandering away. Thrust, roll, pitch, and yaw commands can be independently controlled because they are decoupled. For this reason, three more feedback controllers can be added respectively for yaw, pitch, and roll. Now, these angles are also the plant outputs and the states to estimate and to feedback.

Therefore, in this way, altitude is kept fixed, and the mini drone faces forward. This hover controller is

improved, but there are still some limits. If, for example, multiple wind gusts, as disturbances, flows in the same direction, the controller still allows the drone to meander away slowly. In this control system implementation, the drone is not brought back to its reference position.

The improvements can be made by thinking that pitch and roll angles are not necessarily zero, but they can also assume non-zero values while hovering. For example, the mini drone could hover in a strong constant wind, and in these conditions, it leans at a certain angle into the wind to keep its reference position. Therefore, we should implement a ground position controller that recognizes when the quadcopter wanders off and makes the required changes to take it back to the original point in X-Y coordinates.

Due to the coupling between forwarding, backward, left, and right movements with roll and pitch maneuvers, also position error is coupled with roll and pitch motion. The measured X-Y position should be fed back and compared to the reference signal to obtain the position error. If there is no signal having the sequence of position points to follow generated by the path-planning algorithm, the reference position may be set temporarily to (0, 0). It corresponds to hovering above the take-off point. The position error is taken as input by the position controller, and roll and pitch angles are sent as outputs. These angles are the reference signals for the roll and pitch controllers. In this way, the position controller creates them.

Summarizing, the roll and pitch controllers in the inner loop receive the reference signals from the position controller in an external loop. They are cascaded loops. It should be noticed that also the measured yaw angle should be fed into the position controller. The X-Y position errors are expressed relatively to the environment reference frame, but pitch and roll angles are expressed relatively to the drone frame. This means that a movement in the X and Y world directions can be obtained through pitch and roll only by knowing the yaw angle because it is needed by the position controller for the conversion from the drone X-Y frame to the environment reference frame.

The next picture shows the structure of the architecture described before (figure 6.4).

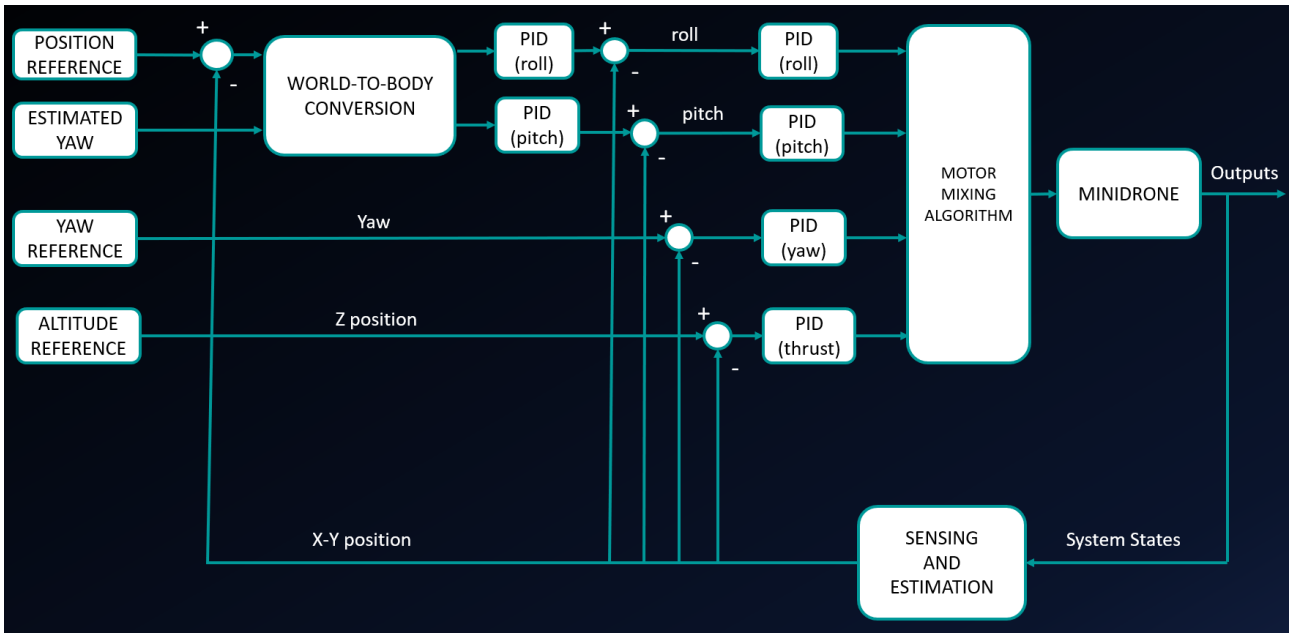


Figure 6.4: Feedback Control loop with a nonlinear model for simulation [34]

In the next paragraph, where the Simulink model will be described, it can be noticed that the controller model has the architecture of the previously explained feedback control system.

6.3 Simulink model structure description

In this paragraph, the Quadrotor model used for simulation purposes will be analyzed, focusing on the part of the model briefly anticipated in chapter 5. Two main parts are described: Simulation model and Flight control system for code generation.

6.3.1 Simulation model

The following picture (6.5) represents the subsystems that constitute the simulation model: the multi-copter model, the sensors model, the environment model, and the flags to stop the simulation, all wrapped around the Flight Control Software.

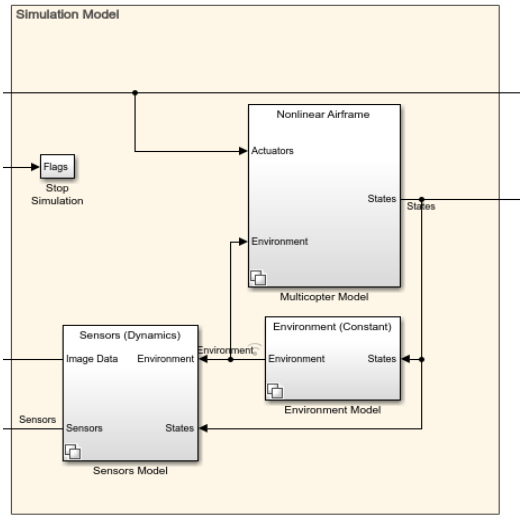


Figure 6.5: Simulation model blocks

Starting from the **multicopter (or airframe) model** (Figure 6.6), it can be noticed that the inputs are the actuators' signals and the data coming from the environment model, while the outputs are the states that must be estimated by state estimator algorithm.

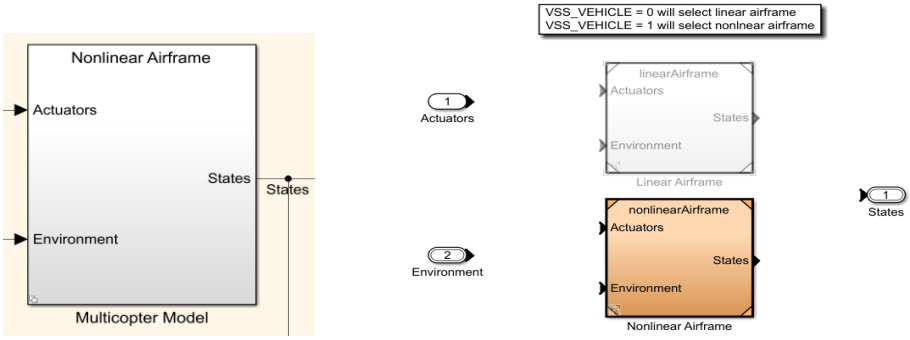


Figure 6.6: Airframe model (nonlinear and linear)

The airframe model is implemented as a variant subsystem, which means that before the model is run, we can select which version of airframe we want to run with: either nonlinear model, that can be used for simulating the flight, or the linear airframe model, that can be used to tune the controllers (Figure 6.7).

The linear model structure can be represented through the typical state-space description and through the blocks corresponding to "*trimLinearizeOpPoint*" functions that, through Simulink Control Design, turn the

mini drone nonlinear model into a linearized one.

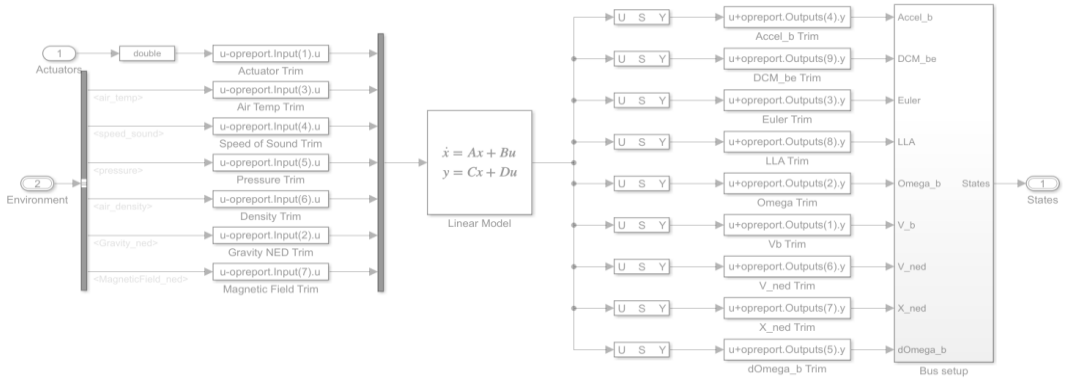


Figure 6.7: Quadcopter Linear model

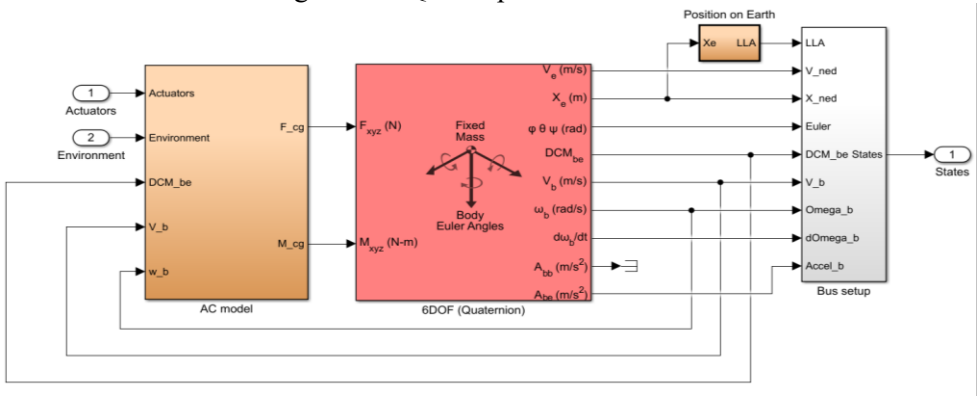


Figure 6.8: Quadcopter Nonlinear model

Looking inside the nonlinear model (figure 6.8), it can be noticed that there are two main blocks. The **AC model** on the left consists of the actuators models and a model of how the environment disturbances impact the system (these subsystems are shown in figure 6.9).

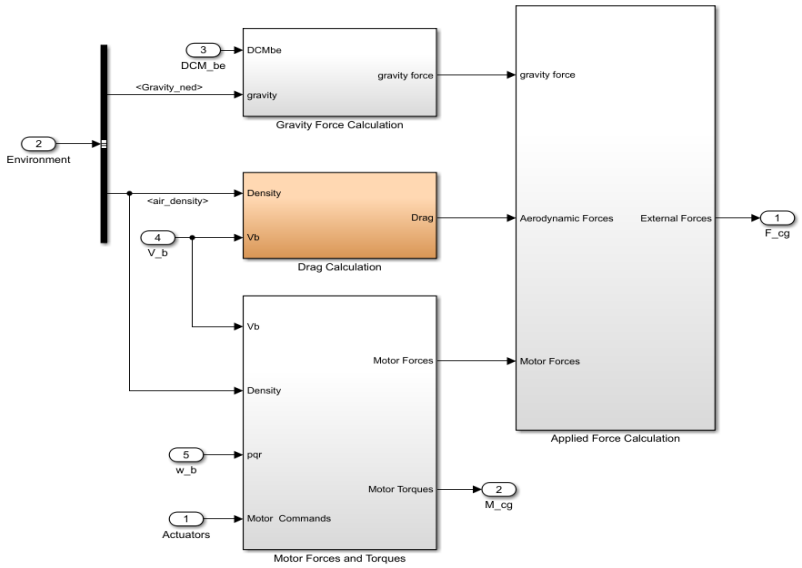


Figure 6.9: AC model (actuators)

Anything that can create a force or torque on the mini drone is calculated in this block. The forces and torques are then fed into the **6DOF model**. This is a rigid body model that comes with the aerospace blockset. It is an example of using an existing model rather than going through the effort of writing out the dynamical equations ourselves.

However, it is still needed to determine the specific parameters for this rigid system like masses and inertias. More than likely, the developer pulled this information from a CAD model of the mini drone, but a physical test could be set up to calculate this information, or it could be done through system identification techniques.

The following pictures show respectively the block diagrams of the "*gravity force calculation*" (figure 6.10 on the left), "*drag calculation*" (figure 6.10 on the right, and 6.11), "*motor forces and torques*" (from figure 6.12 to 6.15) and "*applied force calculation*" (figure 6.16) of the **AC model.**

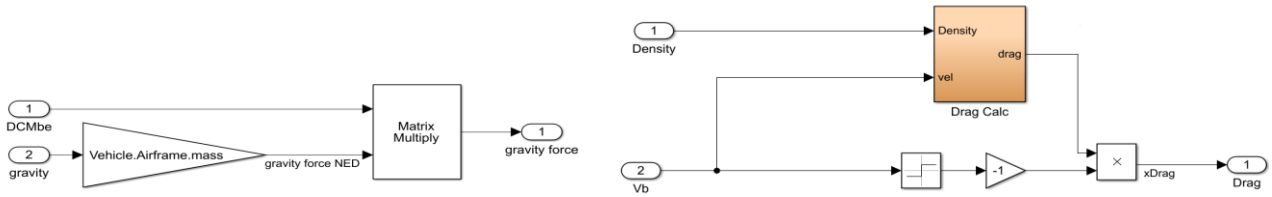


Figure 6.10: Gravity Force calculation (left) and Drag calculation (right)

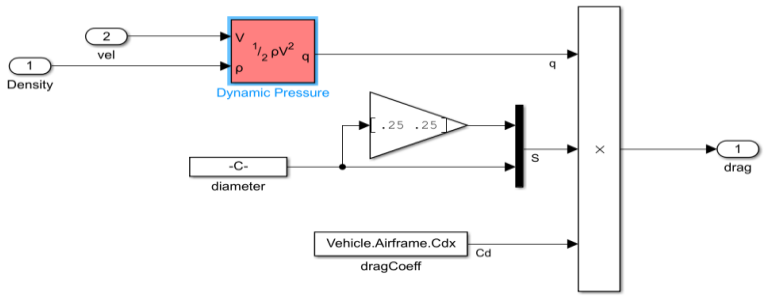


Figure 6.11: Drag calculation subsystem

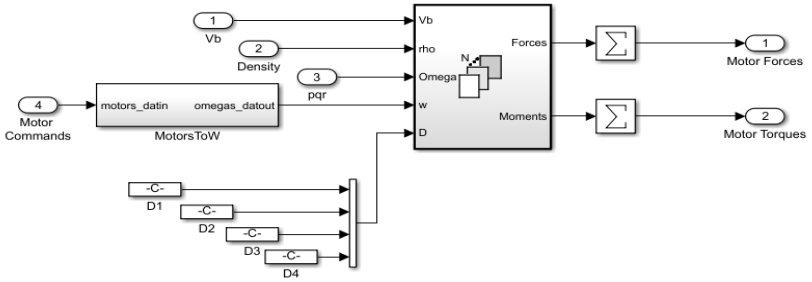


Figure 6.12: Motor forces and torques

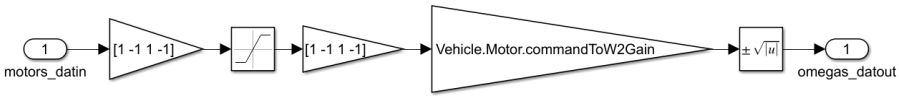


Figure 6.13: Motor forces and torques (MotorsToW subsystem)

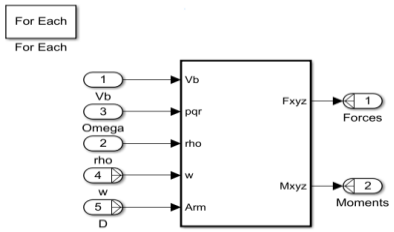


Figure 6.14: Motor forces and torques (Matrix concatenation subsystem)

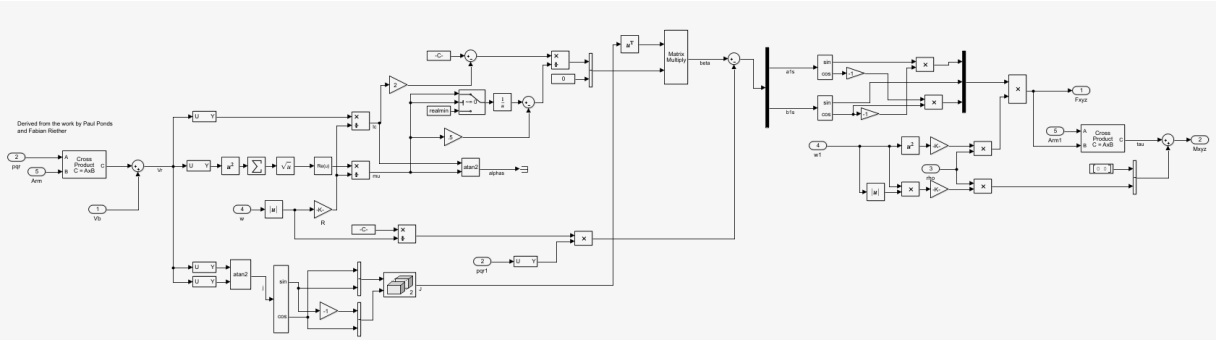


Figure 6.15: Motor forces and torques (Matrix concatenation subsystem structure)

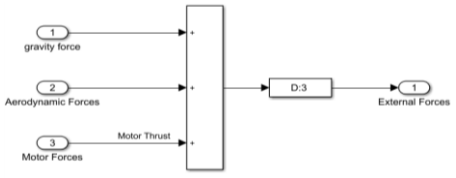


Figure 6.16: Applied Force calculation

Instead**,** the **6DOF model**, which can be described either through Euler angles or Quaternions, is shown in figure 6.17. This block type is included in the Aerospace blockset add-on tool. The block in figure 6.17 on the right models the change of the reference frame from the mini drone's one to the fixed inertial frame.

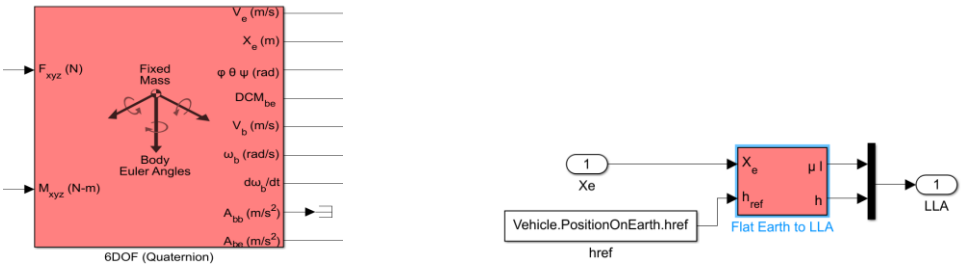


Figure 6.17: 6 DOF model described through Euler angles (left) and change of the reference frame from Body to Earth (right)

If we consider, instead, the **environment block** (figure 6.18) at the top level, it is possible to see that this, again, is a variant subsystem, and we have the option of choosing constant environment variables or variables that change based on position. In this block, different Aerospace Blockset environment models are implemented, for example, the atmosphere and gravity models. The "VSS_ENVIRONMENT" value from the

workspace can be changed to choose between the two previously described models.

However, for this project, the constant variables (figure 6.19) will be selected because things like gravity and air pressure are not going to change between ground level and altitude at less than a meter. Nevertheless, if the objective is to simulate how high the mini drone could fly, then selecting the changing environment (figure 6.20) will lower air pressure and density as it gets higher, which will eventually stall the drone at some maximum altitude. So, choosing one model or another depends on the aim of the test. For our purposes, we will use the Constant Environment model.

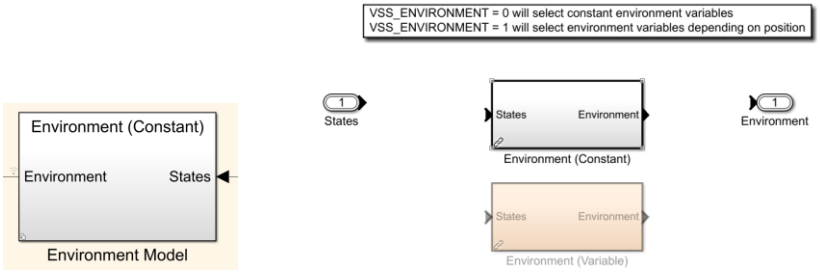


Figure 6.18: Environment model

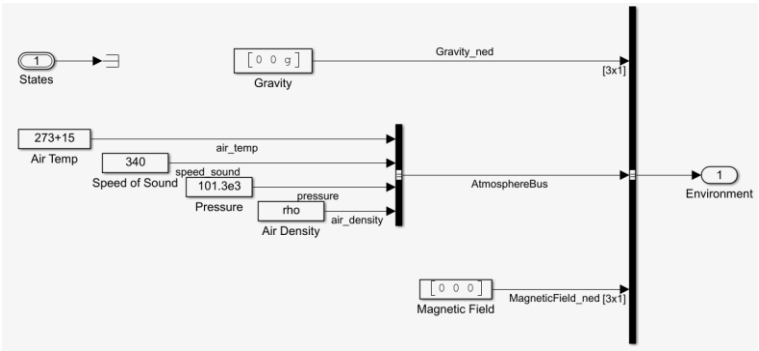


Figure 6.19: Constant environment

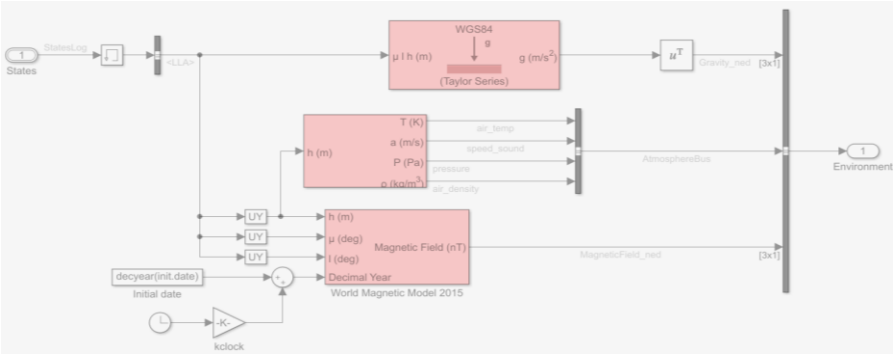


Figure 6.20: Variable environment

Lastly, going into the **sensors block** (figure 6.21), which is also a variant subsystem, the two alternatives to select are either dynamic sensors with noise or feedthrough sensors. The feedthrough option will be select for tuning the controllers, but for simulation purpose, the sensors should be chosen to behave as much like the real things as possible. Inside this subsystem, there are some hardcoded sensor calibration data and a block called sensor system that houses the models for the camera, the IMU, the ultrasound, and the pressure sensor (figure 6.22). As previously explained, the following sensors determine the states of the system.

- An Inertial Measurement Unit (IMU). It measures the translational accelerations and angular rates. (figures 6.23 and 6.25).
- A down-facing camera. It is needed for Image processing and optical flow estimation (figure 6.24)
- An ultrasound sensor. It is needed for altitude measurements (figure 6.25).

The file "*sensorVars"* contains the sensors' characteristics. The "VSS_SENSORS" variable in the workspace can be changed to consider the sensors' dynamics in these measurements.

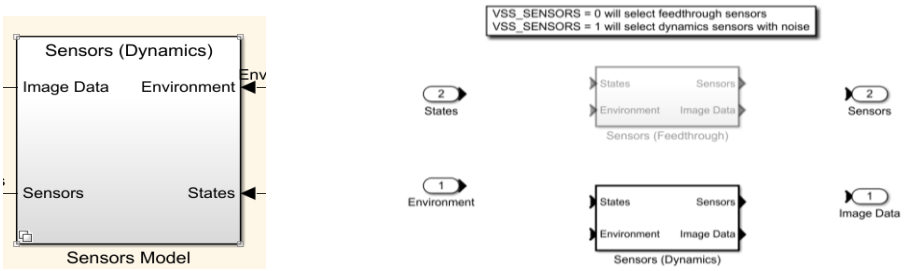


Figure 6.21: Sensors model

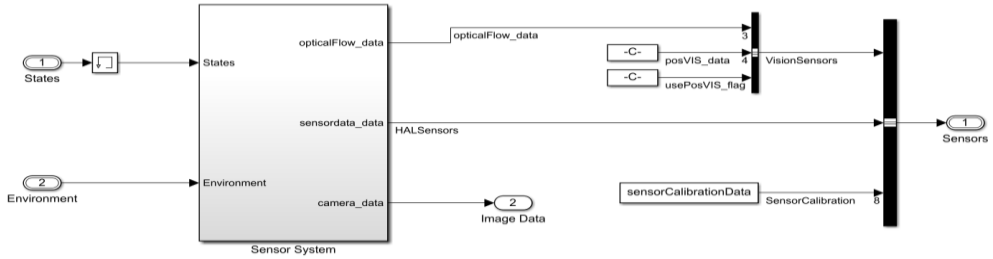


Figure 6.22: Sensors' dynamics

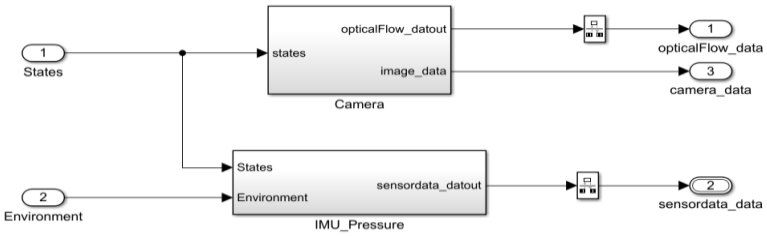


Figure 6.23: Sensor system (Sensors' dynamics subsystem)

It is a crude simulation of the "optical flow (vx,vy,vz)" function as fed into the c-function rsedu_optical_flow() on the mini-drone by internal, inaccessible code of Parrot's firmware. This is assumed to be far from what "optical flow (vx,vy,vz)" actually is.

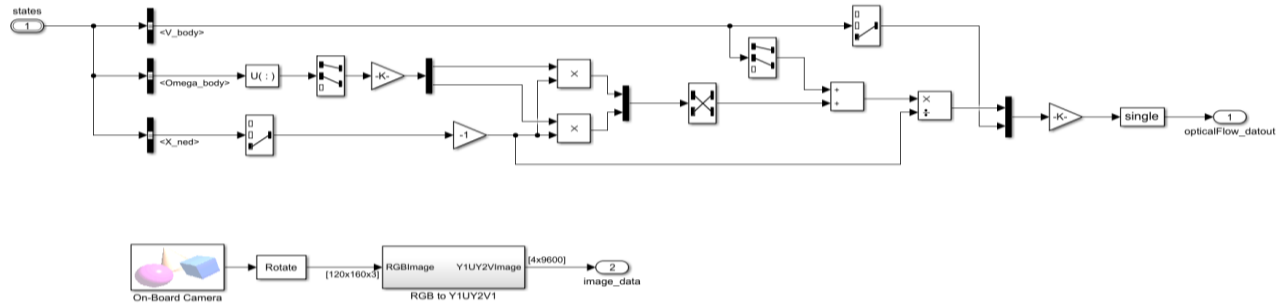


Figure 6.24: Camera model

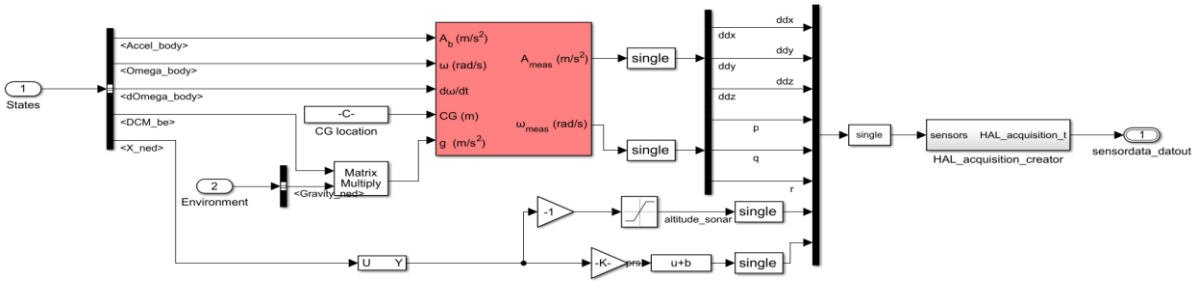


Figure 6.25: IMU and pressure sensor models

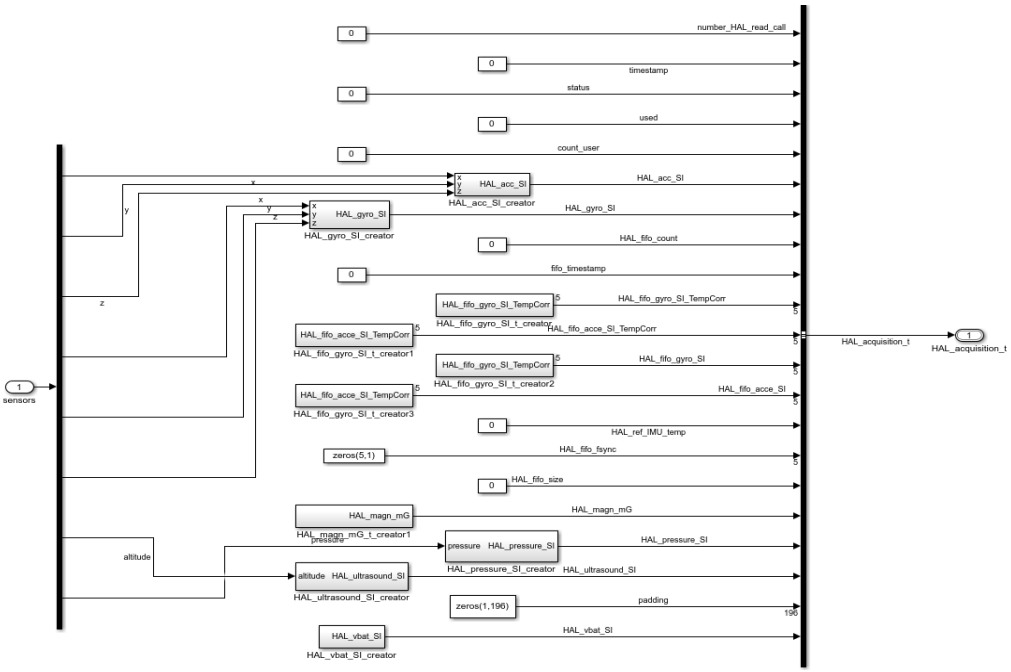


Figure 6.26: HAL acquisition creator model

The model can also be used to safely simulate a failure and see how the system does. For example, we could go in the sensor block, into the IMU model and change the gyro bias. Let us say we estimated the bias poorly, and it is three times worse than expected. Then the simulation can be run. It takes off just about the same, but the gyro bias error quickly causes the drone to roll away from level, and then it runs away and eventually crashes into the ground. If the gyro bias is three times worse, the drone will not perform well, and if there is worry about being this far off on estimating bias, then the stop flag logic should be changed to recognize that it is drifting away and so shut the drone down before the hardware would be hurt.

6.3.2 Flight Control system

A technique for the implementation of the flight controller could be through hand-written C code, then compiling the entire flight code with controller changes and finally deploying the code onto the mini drone (Option 1 in figure 6.27). However, this approach has some drawbacks during the development of the feedback control software. The controllers cannot be easily tuned because they should be tweaked on the hardware.

Moreover, the architecture of control systems realized through C code could be challenging to explain to other people, and the impact of changes on the whole system can be harder to understand than using a graphical method. Instead, the second approach (Option 2 in figure 6.27) consists of a graphical description of the flight controller through block diagrams. In this way, Simulink can be used for development; then, the FCS can be turned automatically into C code, can be compiled, and deployed onto the mini drone.

The Simulink code is easier to read, as well as the performance simulation, and the controller gains tuning through existing tools. Most of the flight code does not have to be written because the Simulink Support Package for Parrot Mini drones will be used to design the customized flight control software. In this way, we can load a new flight firmware to the device, preserving all the standard operating functions, but allowing us to modify only the firmware control part. The same output and input signals should be maintained to ensure that any written code will be placed correctly after Simulink programming, and it will interface appropriately with the rest of the mini-drone firmware.

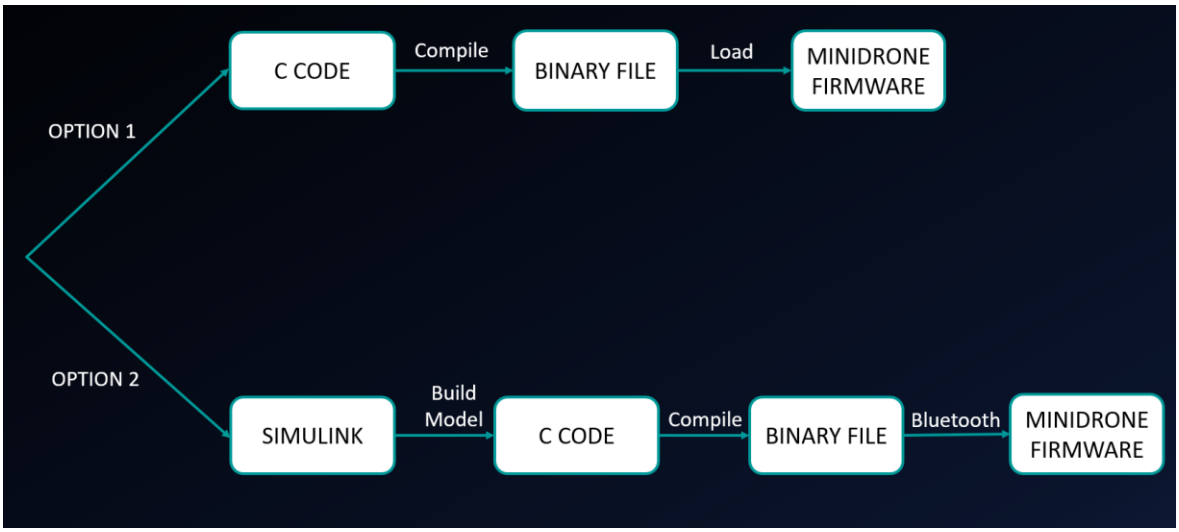


Figure 6.27: the two options by which the FCS could be implemented [35]

During the Hardware-in-the-loop test, the only part that must be deployed on the Hardware is the “Flight Control system” block, which is the orange block (Figure 6.28), which is also the block to be customized to design the flight controller.

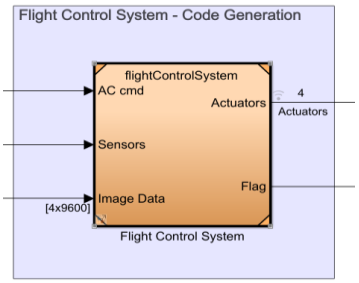


Figure 6.28: Flight Control System block

Inside the Flight Control System, there are two subsystems (Figure 6.29):

- Image Processing System: containing the graphic processing part (the green block).
- Control System: containing the flight logic (the orange block).

The two subsystems work at different rates: The Image Processing System at almost 20 milliseconds (60 Hz as the camera frame rate), while the Control System at 5 milliseconds (200 Hz).

Therefore, they are connected through the "Rate transition" block, which allows the data transfer between systems with a different rate.

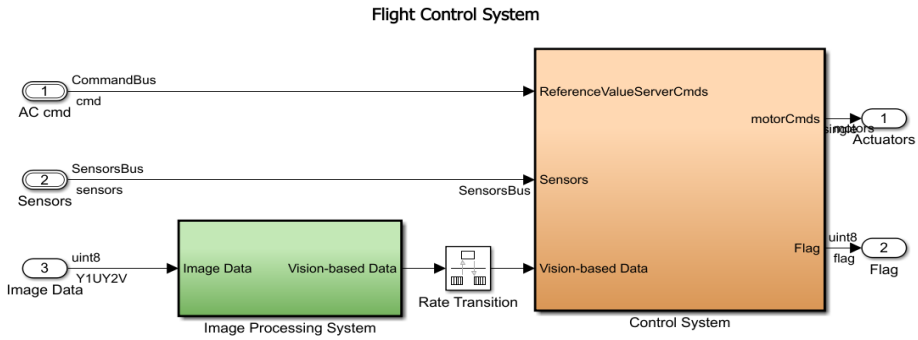


Figure 6.29: FCS structure and its subsystems

Within the Control system subsystem block, there are other different blocks (Figure 6.30):

- Path planning: (the orange one) where the logic for the line-tracking algorithm will be designed.
- Controller: where all the PIDs of the flight controller reside.
- State estimator: contains the state observer.

- Crash Predictor Flags: this contains the logic to turn off the drone in case of anomalies in flight.

On the left, it can be noticed instead of a "constant" block with the value "1" assigned (figure 6.31).

It is used to define the type of drone piloting: if equal to 1: the drone will base its movements according to the X and Y input values, while if the constant is equal to 0: the drone will be piloted changing the Pitch and Roll values. In any case, it is possible to act on the value of the Yaw (in radians).

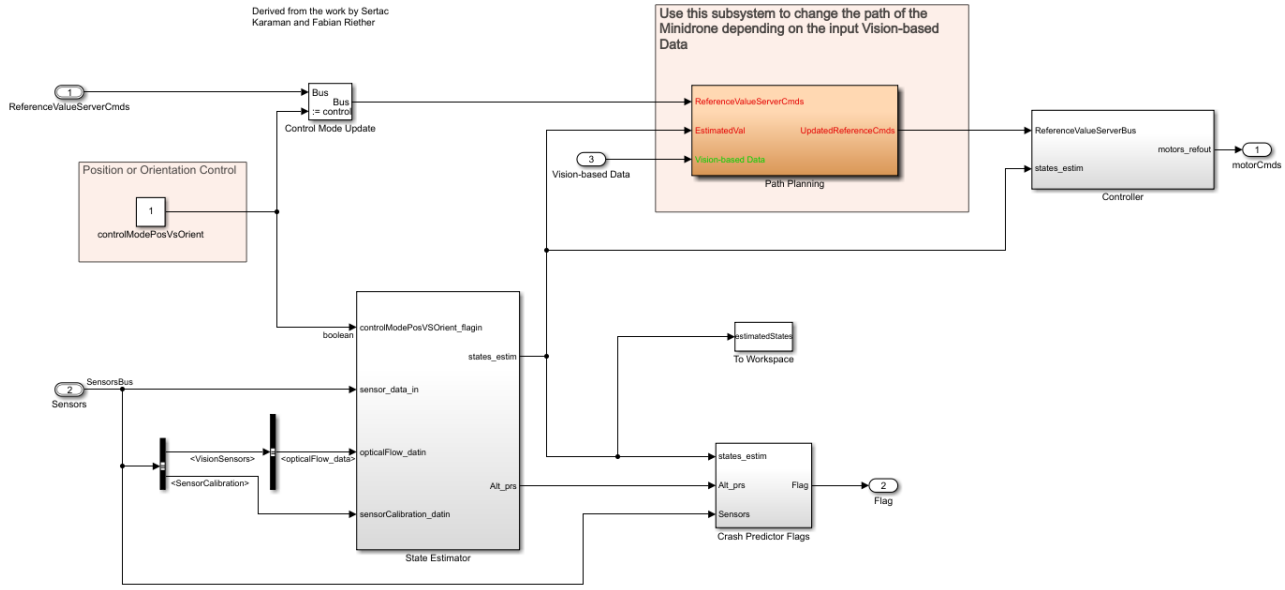


Figure 6.30: "Control System" subsystem structure

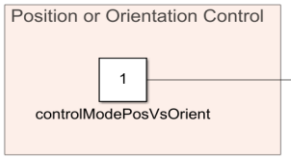


Figure 6.31: Control mode ("1" for Position XYZ, "0" for Orientation Roll-Pitch-Yaw)

The **State estimator block** is the block shown in the next two figures (6.32 and 6.33). A complementary filter is used for attitude estimation and Kalman filters for position and velocity estimations. The variables characterizing the estimator are contained in the "*estimatorVars*" file.

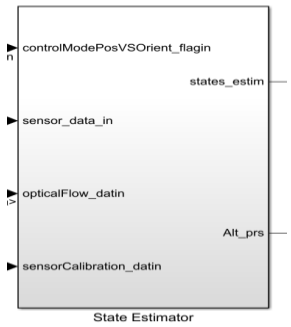


Figure 6.32: State estimator block

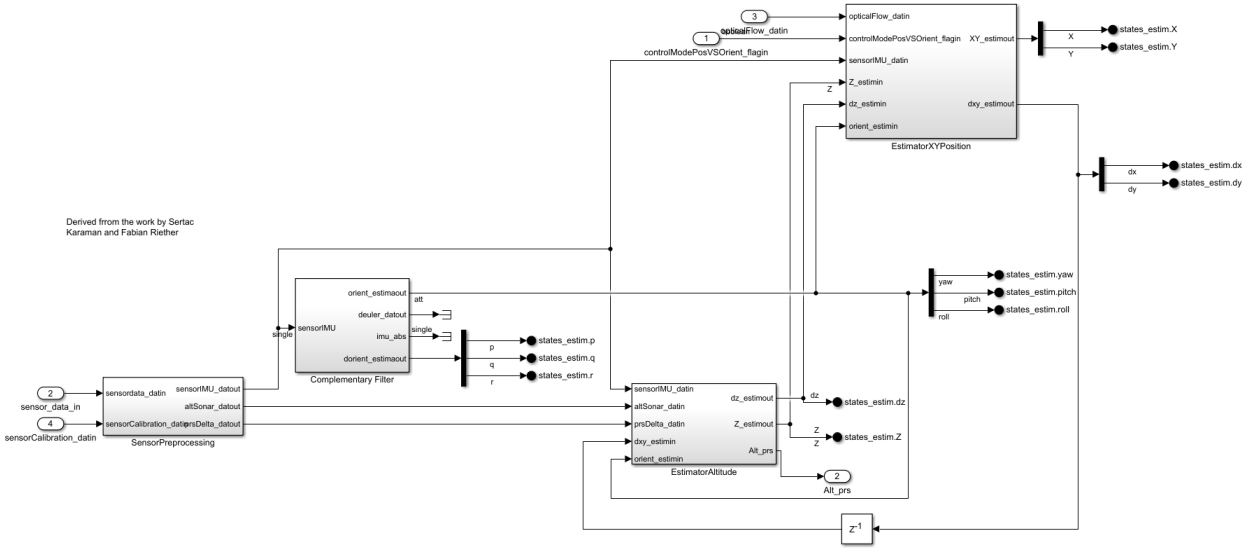


Figure 6.33: State estimator model structure with its subsystems

The acquisition of raw sensor measurements and the state estimation is conducted in two steps. At the first stage, the measurements are processed and then blended with filters to estimate the control states. Therefore, the first stage is the **sensor processing block** (figures 6.34 and 6.35). The sensor data group is needed to extract the individual sensor values from the sensor bus to manage them in the code (figures 6.36).

Firstly, the previously determined bias is subtracted to the gyro and acceleration data to calibrate them. Through bias removal, zero measurement results from zero acceleration and zero angular rates. Next, we should express the measurements from the sensors' frame to the drone frame by performing a rotation transformation. Finally, the measurements must be low pass filtered to cut off the noise at high frequency.

Likewise, we should remove bias from the sonar sensor. The optical flow data is based on a pass/fail criterion: if it has a valid position estimate, then the validity flag is set as true by this block.

Summarizing the sensor preprocessing consists of bias removal, coordinate transformation, and filtering.

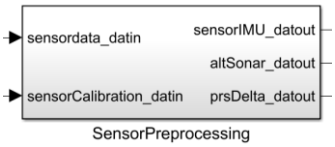


Figure 6.34: Sensor preprocessing subsystem block

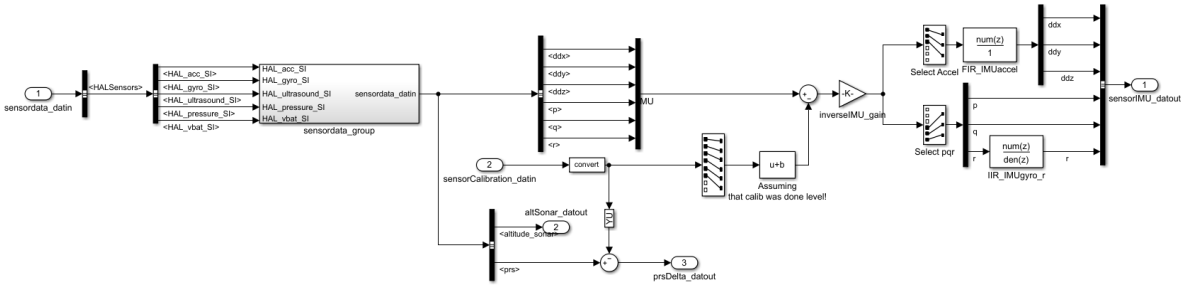
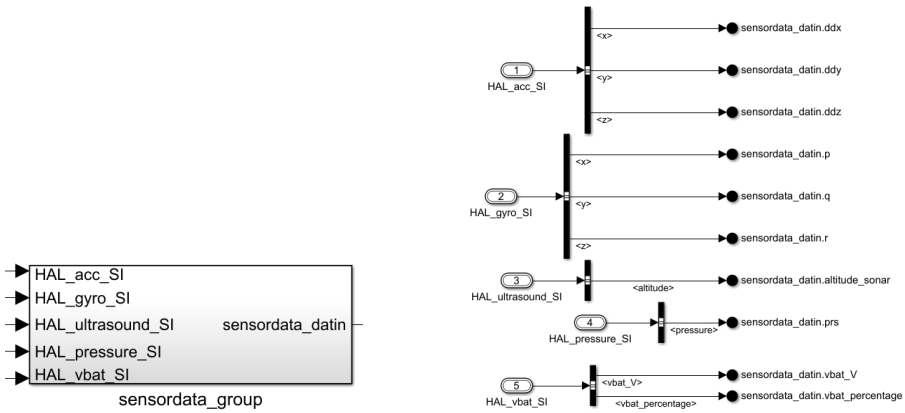


Figure 6.35: Sensor preprocessing subsystem structure



Figures 6.36: Sensor data group subsystem

After filtering and data calibration, we can proceed with combining measurements for state estimation useful for the controller subsystem. The following pictures show the blocks used to estimate the **X-Y position** (from figure 6.37 to 6.46) and **altitude** (from figure 6.47 to 6.50). It can be noticed they employ a Kalman filter to combine the measurements, to predict the dynamical system behavior, and to obtain an optimal

estimation.

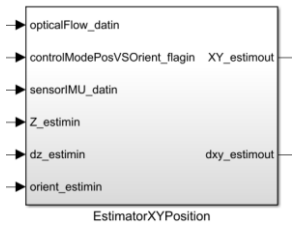


Figure 6.37: X-Y position estimator block

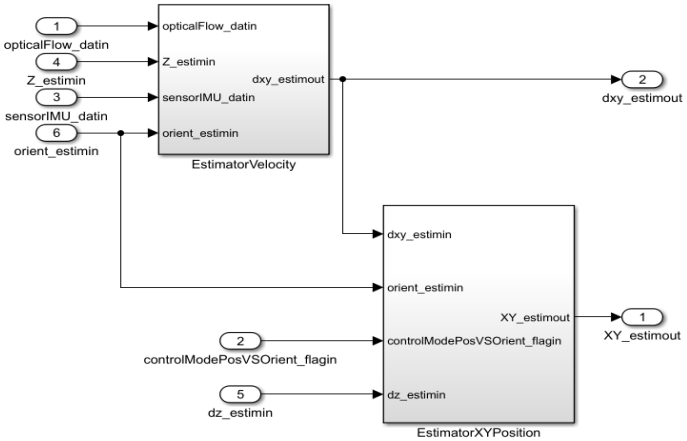


Figure 6.38: X-Y position estimator structure and its subsystems

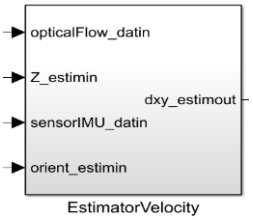


Figure 6.39: velocity estimator subsystem block (part of the X-Y position estimator)

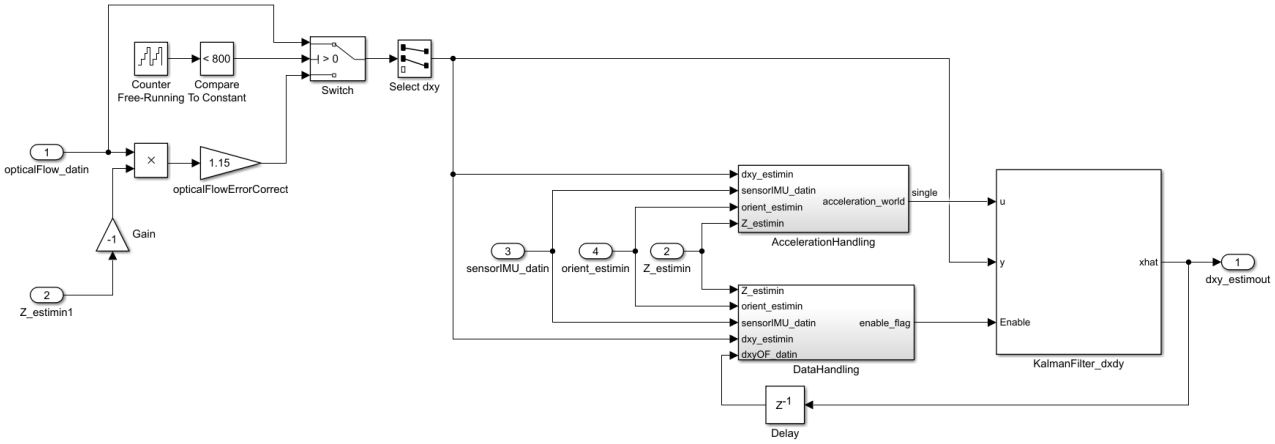


Figure 6.40: velocity estimator subsystem structure

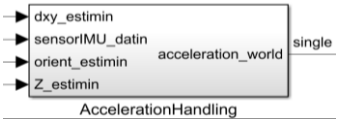


Figure 6.41: Acceleration Handling subsystem block (part of the velocity estimator)

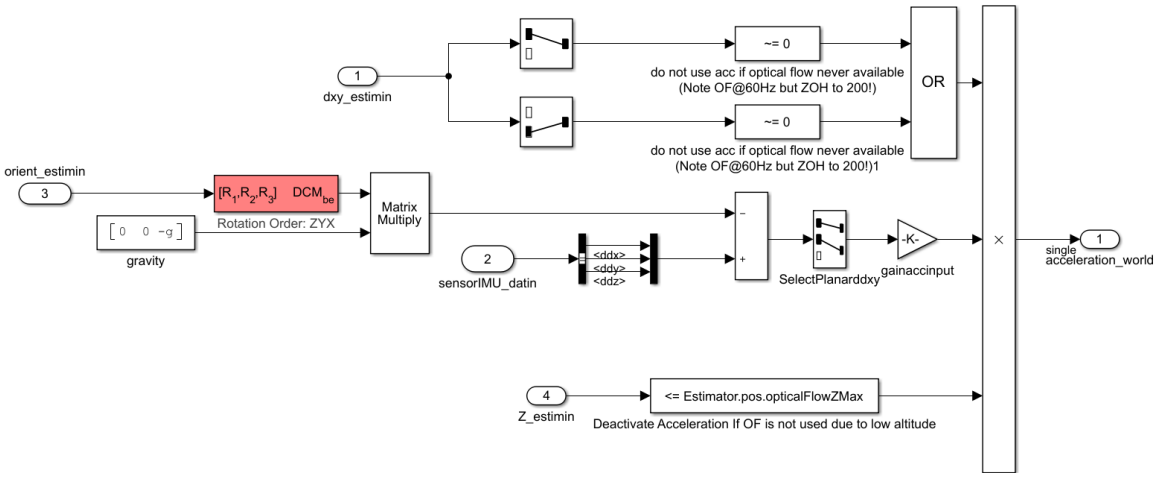


Figure 6.42: Acceleration Handling subsystem structure (part of the velocity estimator)

Figure 6.43: Data Handling subsystem block (part of the velocity estimator)

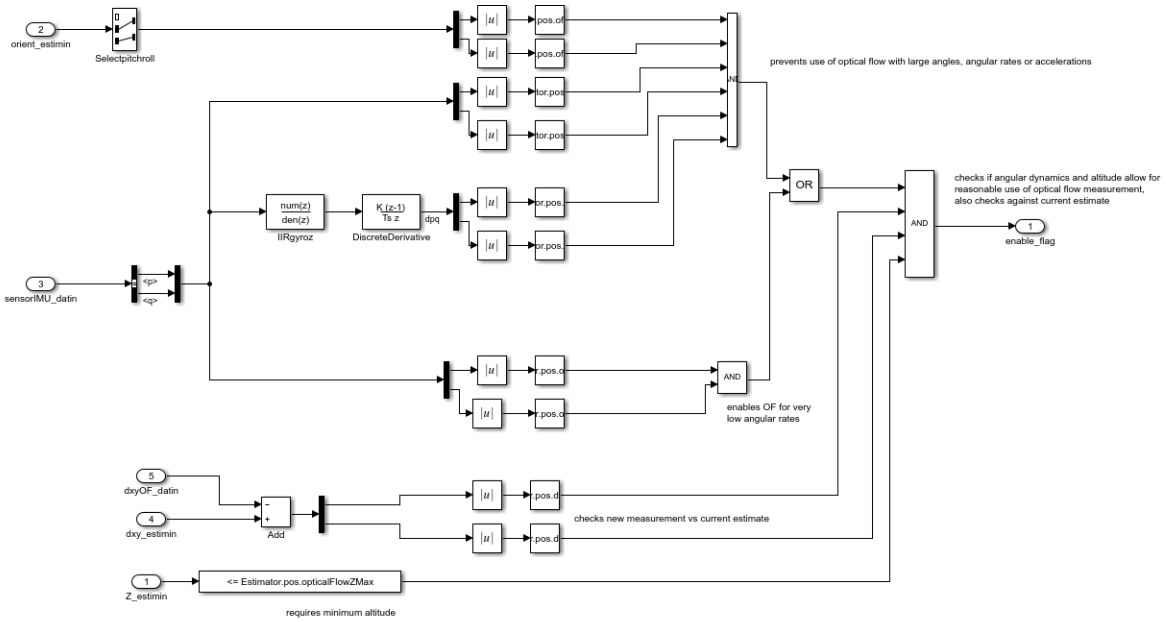


Figure 6.44: Data Handling subsystem structure (part of the velocity estimator)

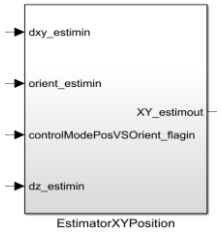


Figure 6.45: X-Y position estimator subsystem block (part of the general X-Y position estimator)

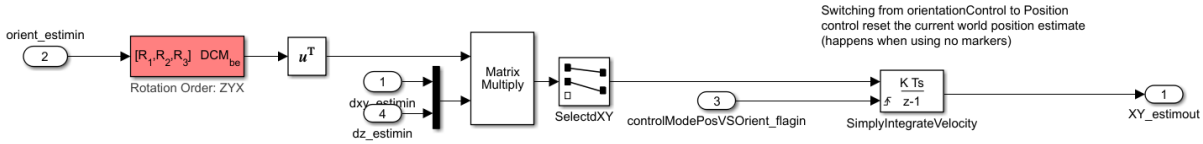


Figure 6.46: X-Y position estimator subsystem structure

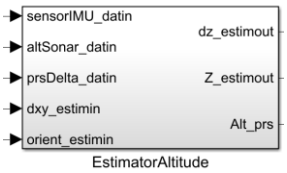


Figure 6.47: Altitude (Z position) estimator block

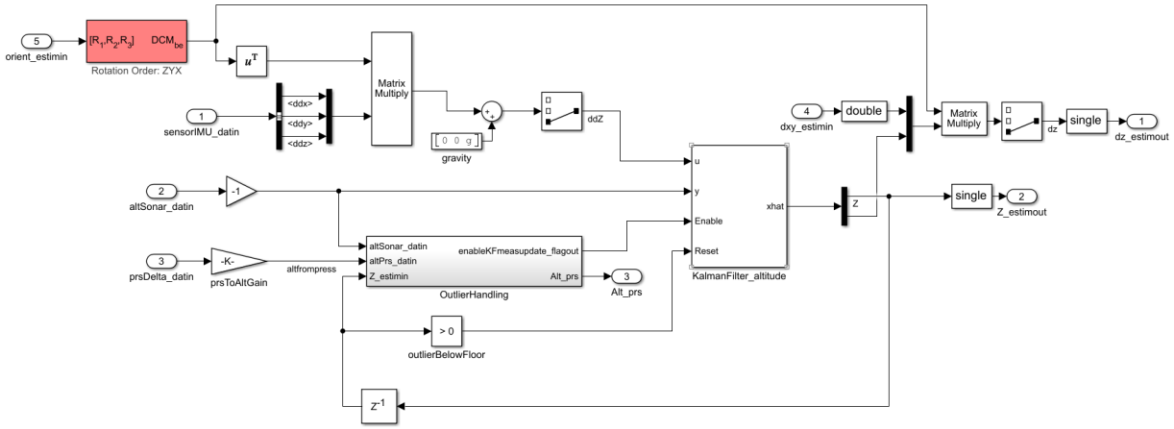


Figure 6.48: Altitude (Z position) estimator structure

Figure 6.49: Outlier Handling subsystem block (part of the Altitude estimator)

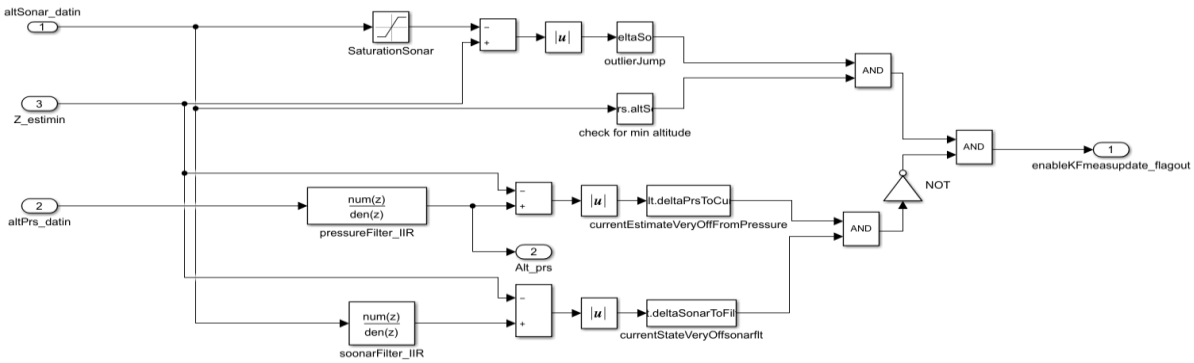


Figure 6.50: Altitude (Z position) estimator block

The block in figures 6.51 and 6.52 (subsystems in figures 6.53 and 6.54) is used to estimate roll, pitch, and yaw angles through a **complementary filter**. This type of filter can simply combine measurements from two sensors, and it is suitable for the subsystem.

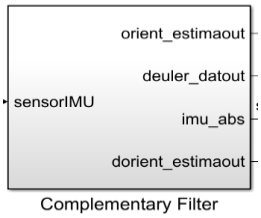


Figure 6.51: Complementary filter block for orientation estimation

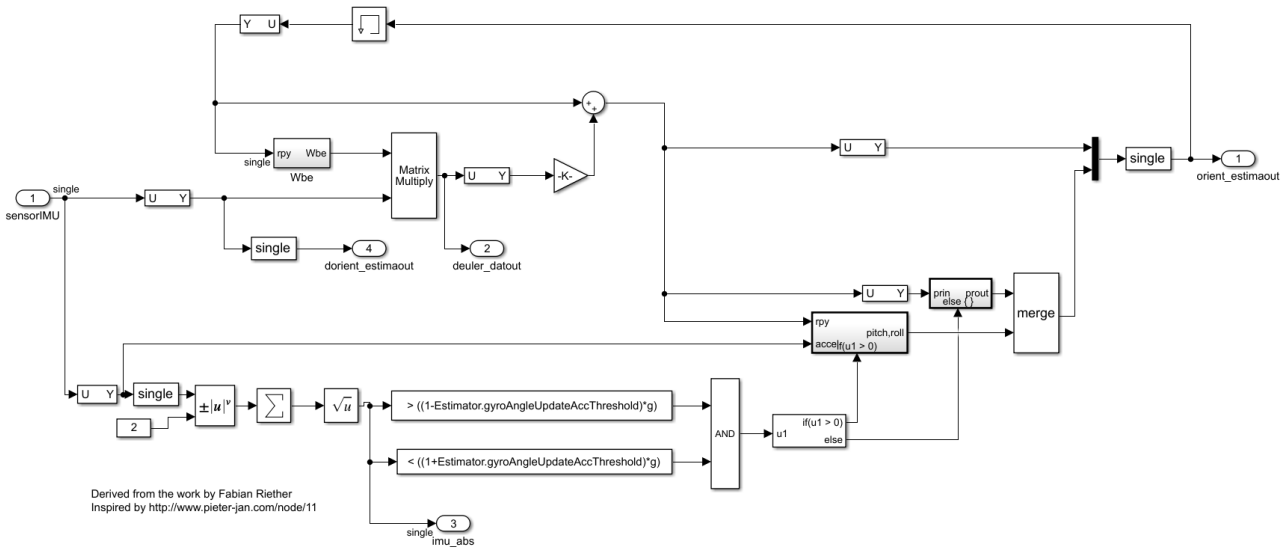


Figure 6.52: Complementary filter structure

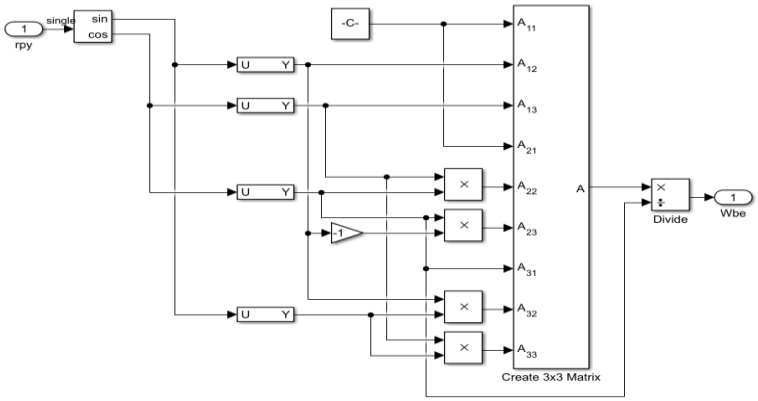


Figure 6.53: Complementary filter subsystem 1

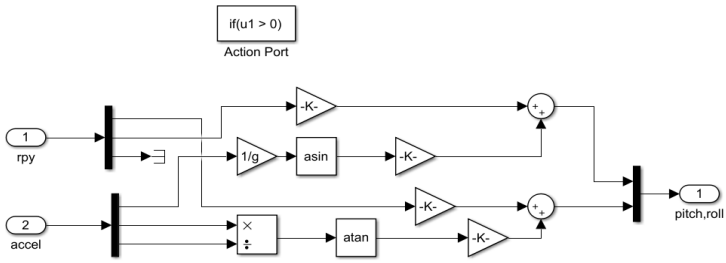


Figure 6.54: Complementary filter subsystem 2

The complementary filter is a more straightforward solution to implement compared to the very capable but intricate Kalman filter, which is often harder to implement on specific microcontrollers. The following description of the complementary filter was adapted from Jouybari et al. [38].

The gyroscope and accelerometer measurements are employed for the same objective: determine the drone's angular position. We can achieve this with the integration over time of the gyroscope angular velocity. Instead, the angular position can be obtained through the accelerometer, using the gravity vector position. We can simply achieve this with the "atan2" function. However, there are two problems, justifying the need for a filter to process data. The first issue concerns the accelerometer. Since it measures all the forces acting on the drone, it can also perceive more components than only the gravity vector. All the small forces could disturb the measurement. The sensor can also sense the forces driving the drone since the system is actuated. Therefore, we should use a low-pass filter because of the reliability of accelerometer measurements exclusively in the long term.

The second issue affects the gyroscope. We can easily obtain accurate measurements not significantly influenced by external forces, but due to the integration over time, there is a tendency of the measurement to drift, that is, not coming back to zero value after a return of the system to its original position. Due to this tendency of drifting in the long term, gyroscope measurements are reliable only in the short term.

Therefore, a complementary filter can be the right tradeoff solution to the two issues. Because of the gyroscope precision, not sensitive to external forces, we can use its measurements in the short term. Instead,

we can use the accelerometer measurements in the long term, for the absence of drift. The filter can be implemented according to the following simple form, adapted from Van de Maele [37].

$$angle = 0.98 * (angle + gyrData * dt) + 0.02 * (accelData) \quad (6.1)$$

In this way, we integrate the gyroscope measurement (*gyrData*) every timestep with the current value of the angle (*angle*). Next, we combine it with the filtered measurement from the accelerometer (*accelData*), after processing it with the "atan2" function. The total sum of the constant parameters in the formula should be 1, and they should be chosen according to our filter tuning. This function implemented through Simulink blocks must be used with a loop. Every cycle, we update the roll and pitch angles with the new measurement from the gyroscope with time integration. Then, the angles are updated with the accelerometer measurement considering 98% of the actual measurement and summing up 2% of the angle computed from accelerometer data. In that way, the measurement is ensured not to drift, and it is accurate in the short term.

Figure 6.55 shows a graph created with GNUPlot that represents the signals from gyroscope and accelerometers and their estimation through a complementary filter. We can notice that the filtered signal (in red) follows the gyroscope (in blue) for quick variations, and it follows the mean value of the accelerometer measurements (in green) for slow variations, not sensing the noise and not drifting away too.

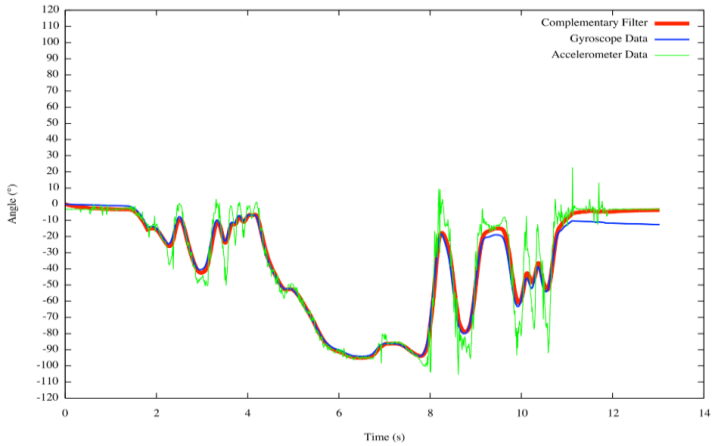


Figure 6.55: gyroscope and accelerometer signals and complementary filter estimation [37]

Now that State estimators have been treated, it can be possible to introduce the controller subsystem (figure

6.56). It takes the reference signals generated from the Path planning algorithm and makes a comparison between them and the estimated states to obtain the errors. They are fed into the PID controllers to produce the actuators' commands (figure 6.57). We have the X-Y position outer loop controller (figures 6.58 and 6.59), sending the signals to the pitch /roll (or attitude) internal loop controller (figures 6.60 and 6.61). Moreover, independently of them, there is a yaw controller (figure 6.62) and the height controller (figures 6.63 and 6.64).

Overall, the position and the attitude of the mini drone are controlled by 6 PID controllers. The "*controllerVars*" folder includes all the variables relative to the controller.

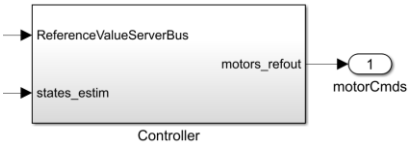


Figure 6.56: Controller block

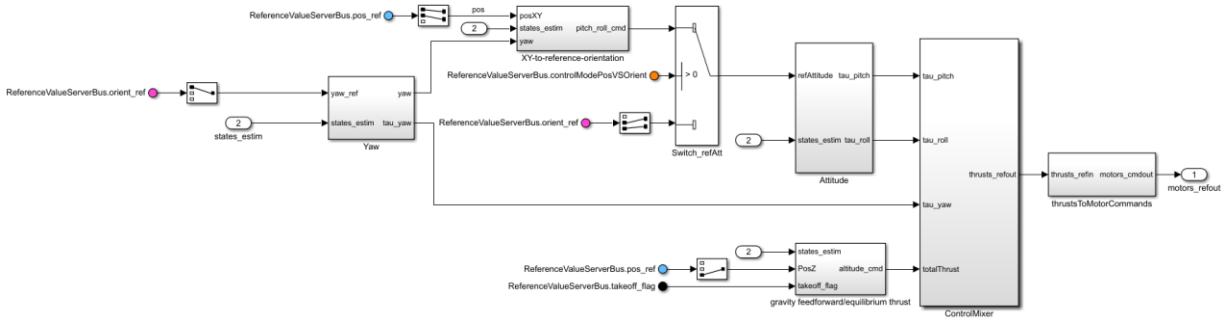


Figure 6.57: Controller model structure

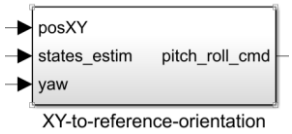


Figure 6.58: X-Y position controller block

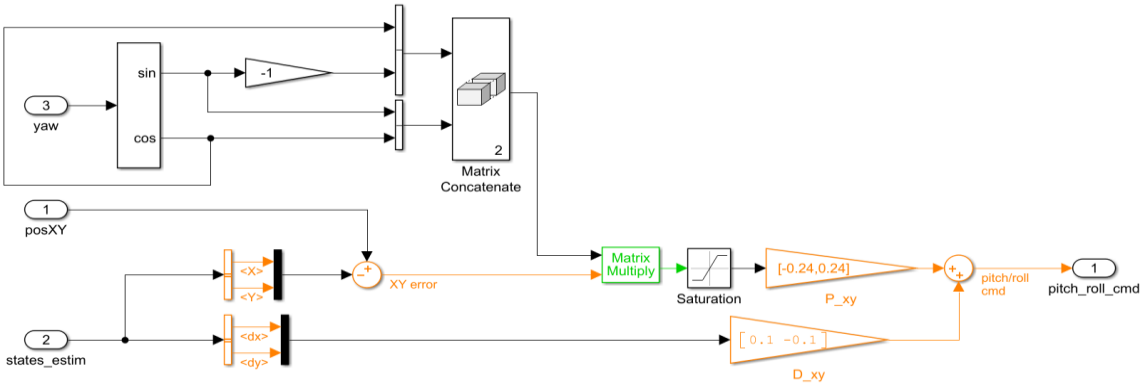


Figure 6.59: X-Y position controller structure

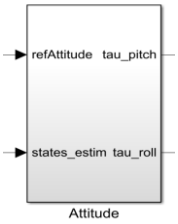


Figure 6.60: Attitude controller block

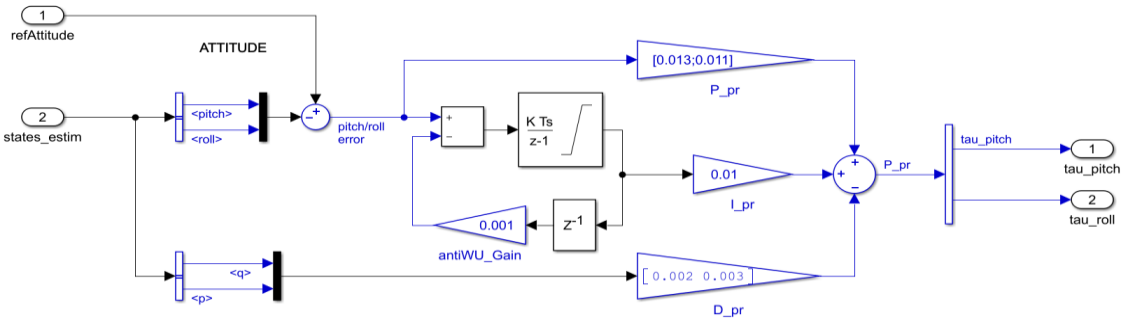
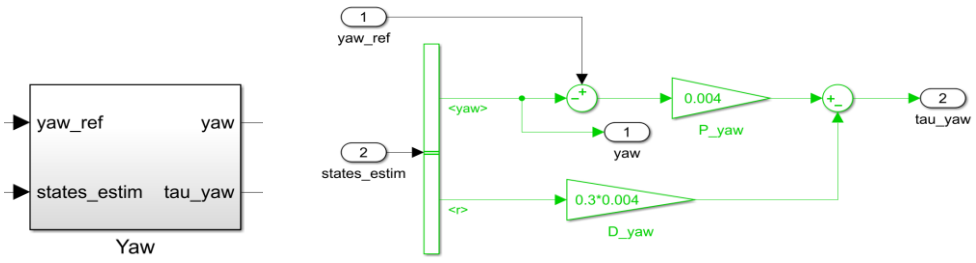


Figure 6.61: Attitude controller structure



Figures 6.62: Yaw controller block and model structure

The altitude controller (figures 6.71 and 6.72) is set up as PID. In this implementation, we multiply the proportional gain by the altitude error derived from the sonar sensor, and the derivative gain by the altitude rate measurement of the gyroscope, which is a less noisy signal compared to the ones from the ultrasound. It would be useful to remember that the z-axis points down in the drone's coordinates system, so in the control system, there will always be a negative sign in front of the altitude number (expressed in meters).

Figure 6.63: Altitude controller block

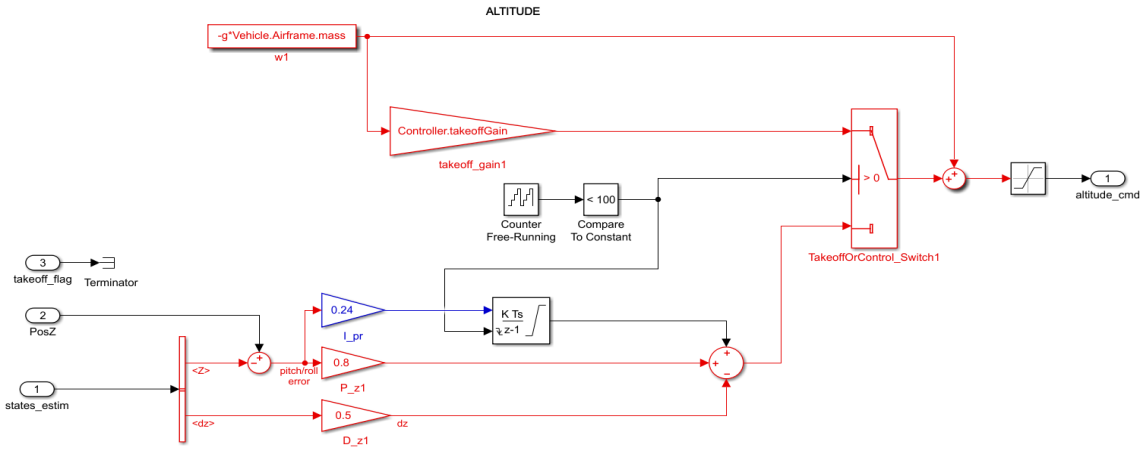


Figure 6.64: Altitude controller structure

Those PID and PD controllers have as outputs the force and torque commands that are subsequently sent to the MMA (figure 6.65). It produces the necessary motor thrusts, and, through the blocks in figure 6.66, the commands are turned into motors' speeds.

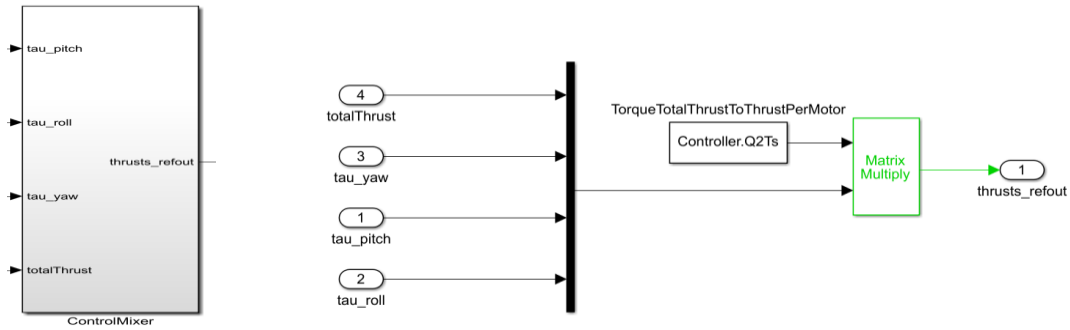


Figure 6.65: Motor mixing Algorithm block and model structure (part 1)

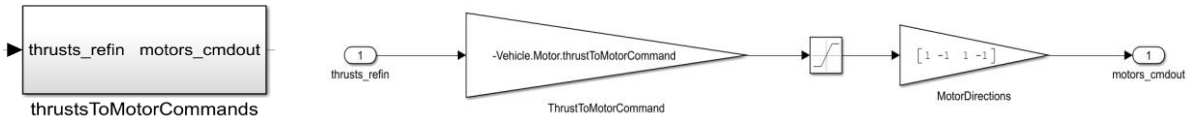


Figure 6.66: Motor mixing Algorithm block and model structure (part 2)

6.4 Controllers Tuning

As previously stated, to tune the controllers, we should build a linear model, because nonlinear models are not suitable for controller design, despite their accuracy in simulation. In this example, the "PID tuner" app of Simulink will be used for tuning the height controller, and the same approach can be applied for the other controllers according to the technique explained by Douglas [35].

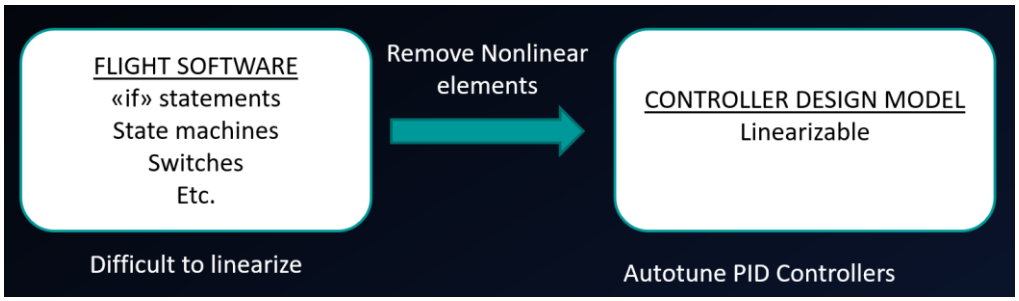


Figure 6.67: Model linearization for Controllers tuning [35]

Once the system is linearized, we can apply the superposition principle, so it is possible to tweak and adjust altitude without influencing yaw, pitch, and roll because the altitude controller is independent of the other ones. We can also assume that sensor dynamics and noise effects are negligible in the controller design.

Therefore, we can remove the state estimation logic and the sensors model. Moreover, we should assume the controller knows the actual height.

Once the controllers are tuned, they will be tested on the original nonlinear model and check that the assumptions were valid.

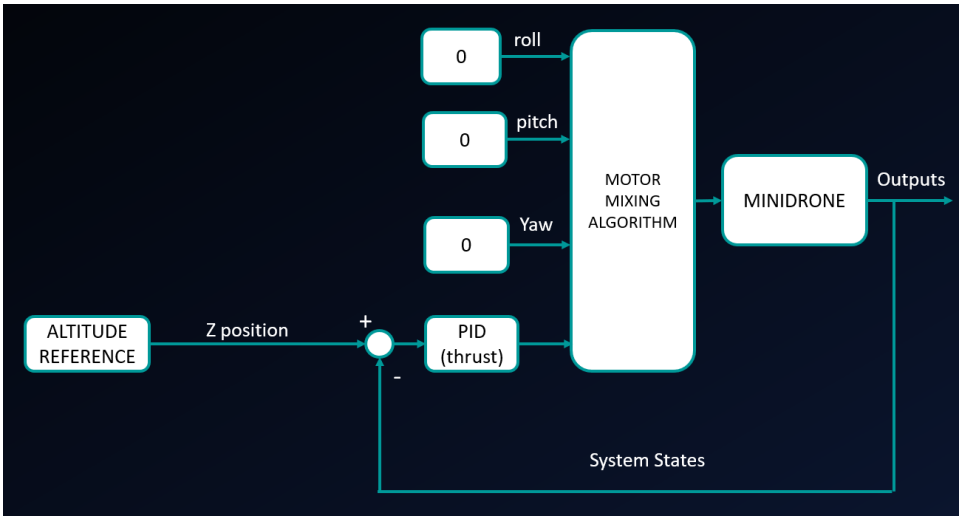


Figure 6.68: Control system simplification for Altitude controller tuning [34]

We should linearize the altitude controller and adjust the gains to obtain the desired dynamics.

After that, the yaw controller will be tuned, with roll, pitch and altitude constant, and a similar procedure for pitch and then roll. Then, the outer loop position controller will be tuned while the inner loop controllers are working and keeping the orientation.

For the linearization of the altitude loop example, we can remove the scope and the sensor block since there is a feedback of a perfect altitude state. In the flight controller block, we can keep the altitude reference, the controller, the MMA, and the thrust-to-motor command block. The RPY torques are set to 0 to ensure that without external forces and torques as disturbances on the drone, it can only increase or decrease altitude. This can be assumed because the environment block does not model external disturbances.

Finally, the perfect altitude state must be fed back to the altitude controller. The linearized system is shown in figure 6.69.

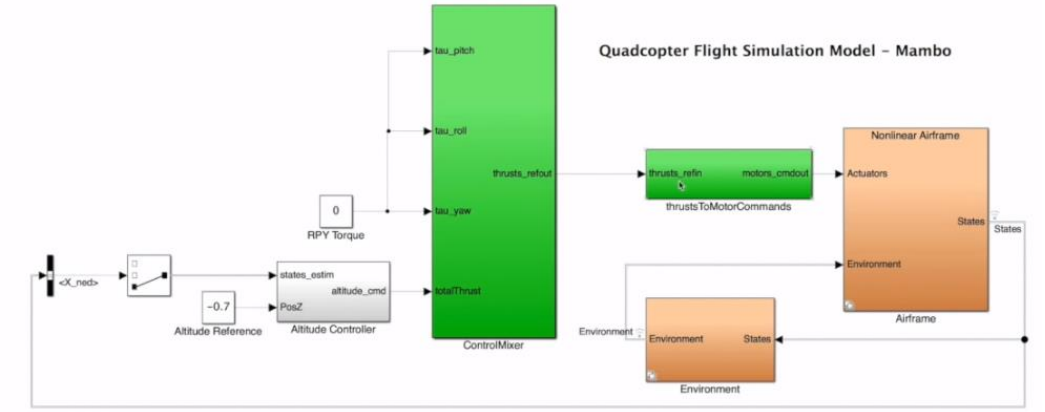


Figure 6.69: Linearized Control system used for Altitude controller tuning [35]

The altitude controller can be implemented with a PD controller through the relative block in Simulink because then it can be used for autotuning. The structure consists of a comparison between the altitude reference and the actual altitude, a PD controller that receives the error, and an addition of a feedforward gravity term.

The last one models the necessary thrust to offset the drone weight. In this way, the controller can simply command the altitude positively to rise and negatively to descend. An alternative to the use of this term could be the addition of an integral term to the controller. The next figure (6.70) shows the linearized controller model used for tuning.

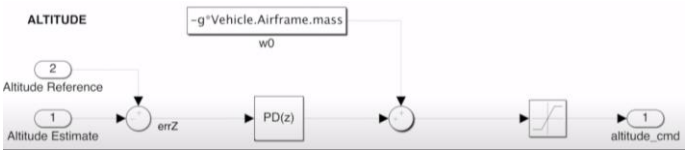


Figure 6.70: Simplified Altitude Controller used for tuning [35]

The “autotuner” can be run by opening the PD block. Firstly, it linearizes the control loop. Then, the app plots the closed-loop response of the linearized version, and it is possible to adjust response time and transient behavior of the system (figure 6.71).

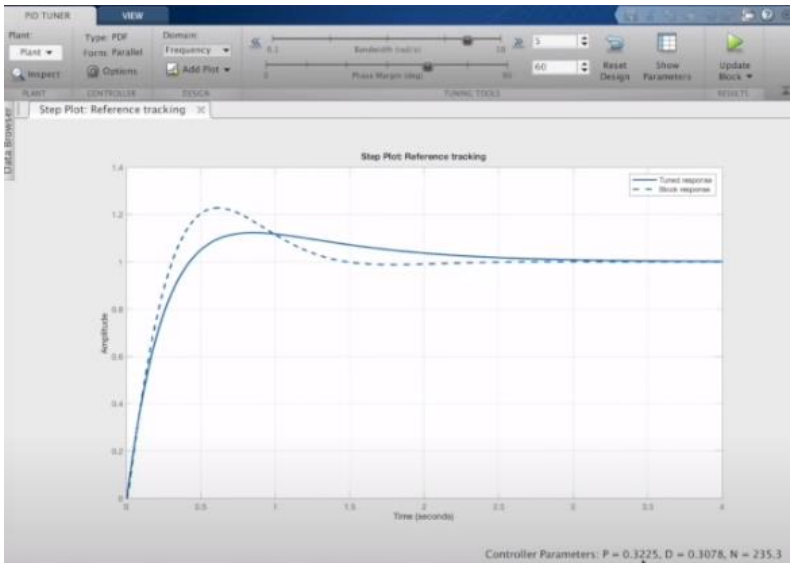


Figure 6.71: PID Tuner App on Simulink [35]

The dashed line of the response signal has not the same behavior of simulation because of the removal of nonlinear components, but it is still suitable for tuning. After gain selection and a quick test with the hardware, it can be noticed that the hardware does not reflect the expected behavior because it cannot take off properly.

The problem, in this case, affects the feedforward term. If it is too little, the system believes the drone weight is smaller than in reality, or the thrust is higher than the actual one. Therefore, the controller has more difficulty in handling the remaining weight because of a reduction of the proportional path, so it cannot take off. If the value is raised to about 25%, the drone can finally takeoff.

The same tuning procedure could be repeated for the other controllers until not just the simulation performances, but also the hardware test performances, are considered acceptable.

Table I shows the gains of the six controllers tuned through simulation and real-time tests by following the technique explained previously.

TABLE I: TUNED GAINS OF CONTROLLERS

CONTROLLER TYPE	GAIN TYPE	VALUE (sign according to the reference frame convention)
X-Y position PD controller	P for X coordinate error	-0.24
	P for Y coordinate error	0.24
	D for X coordinate error	0.1
	D for Y coordinate error	-0.1
Attitude PID controller (pitch/roll)	P for pitch error	0.013
	P for roll error	0.011
	I for pitch error for attitude loop	0.01
	I for pitch/roll error in the altitude loop	0.24
	D for pitch error	0.002
	D for roll error	0.003
Yaw PD controller	P for yaw error	0.004
	D for yaw error	0.0012
Altitude PID controller (Z)	P for altitude error	0.8
	D for altitude error	0.5

6.5 Hovering Hardware-in-the-Loop Test

The Hovering test was performed to check the real-time performances after controllers tuning. During the test, the drone was pushed slightly, with pulses acting as disturbances on the airframe. The figures on the next page show the graphs obtained plotting the data recorded from sensors during flight. They represent the trajectory (figure 6.72), the motor speeds (figure 6.73), x-y-z positions with their linear speeds (figure 6.74), yaw-pitch-roll angles with their angular velocities (figure 6.75). It is easy to identify on the graphs the moments when disturbances occurred (5 times in total), due to a sudden variation in velocities. The signals show that the drone comes back to the reference point with a response suitable to our control goals.

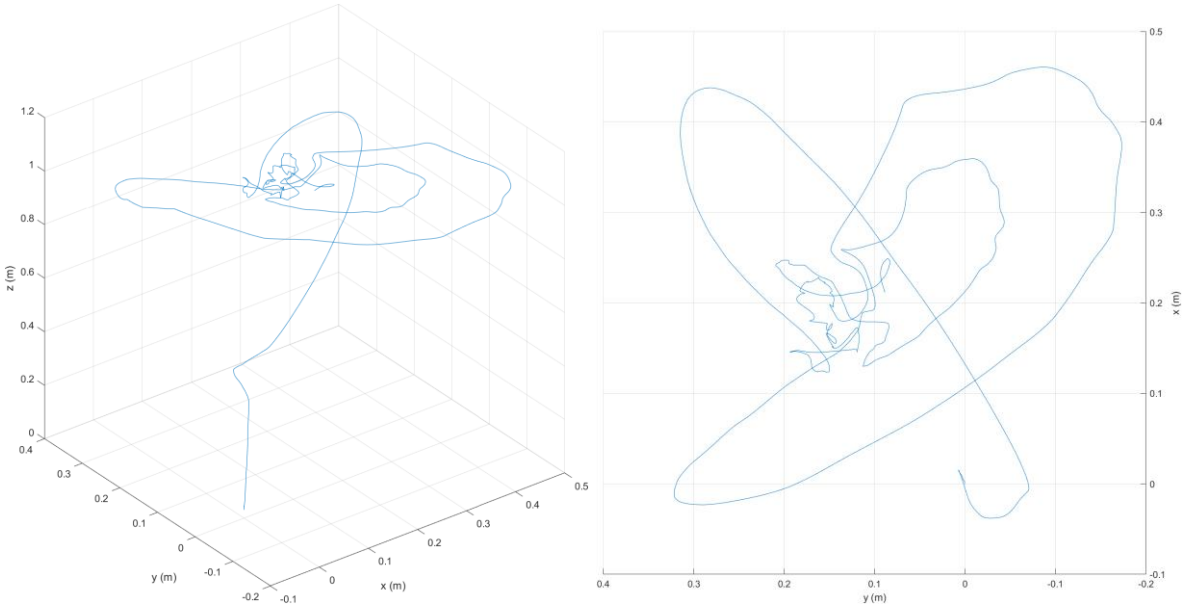


Figure 6.72: real-time Hovering test– Trajectory (3D view and Top view)

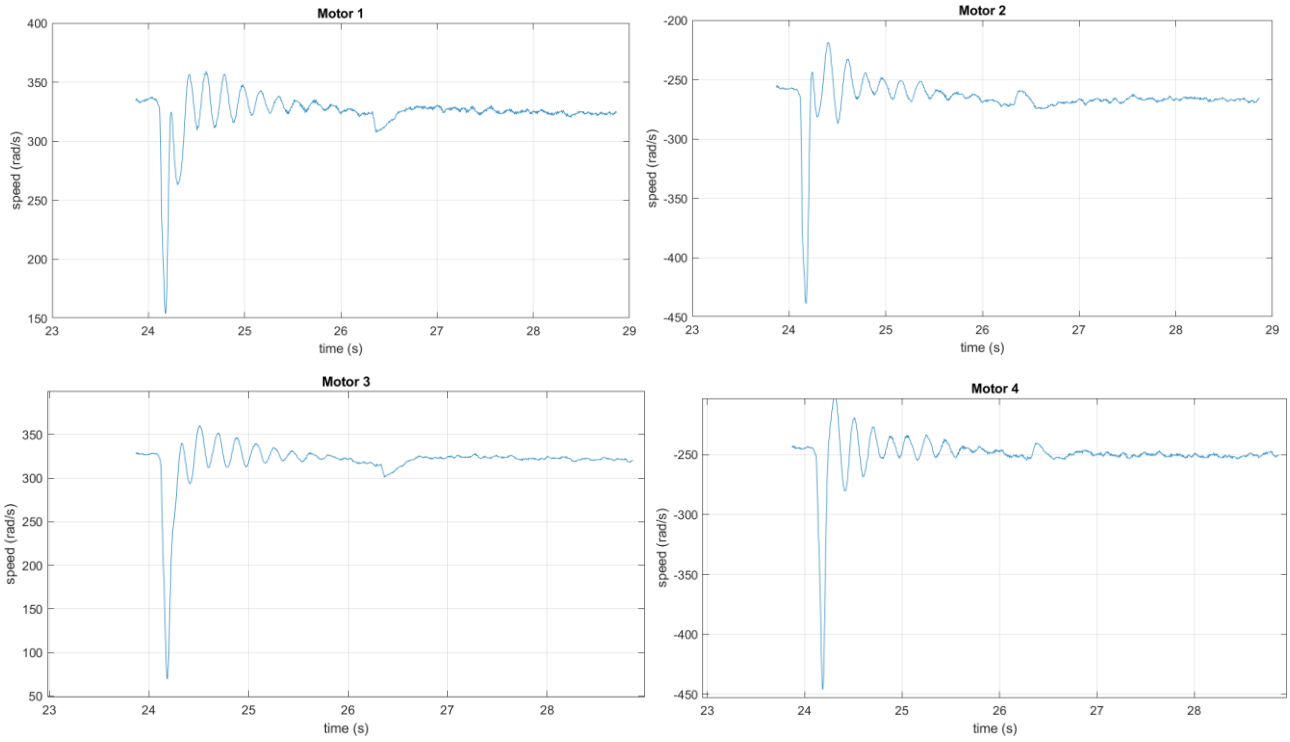


Figure 6.73: Real-time Hovering test– Motor speeds

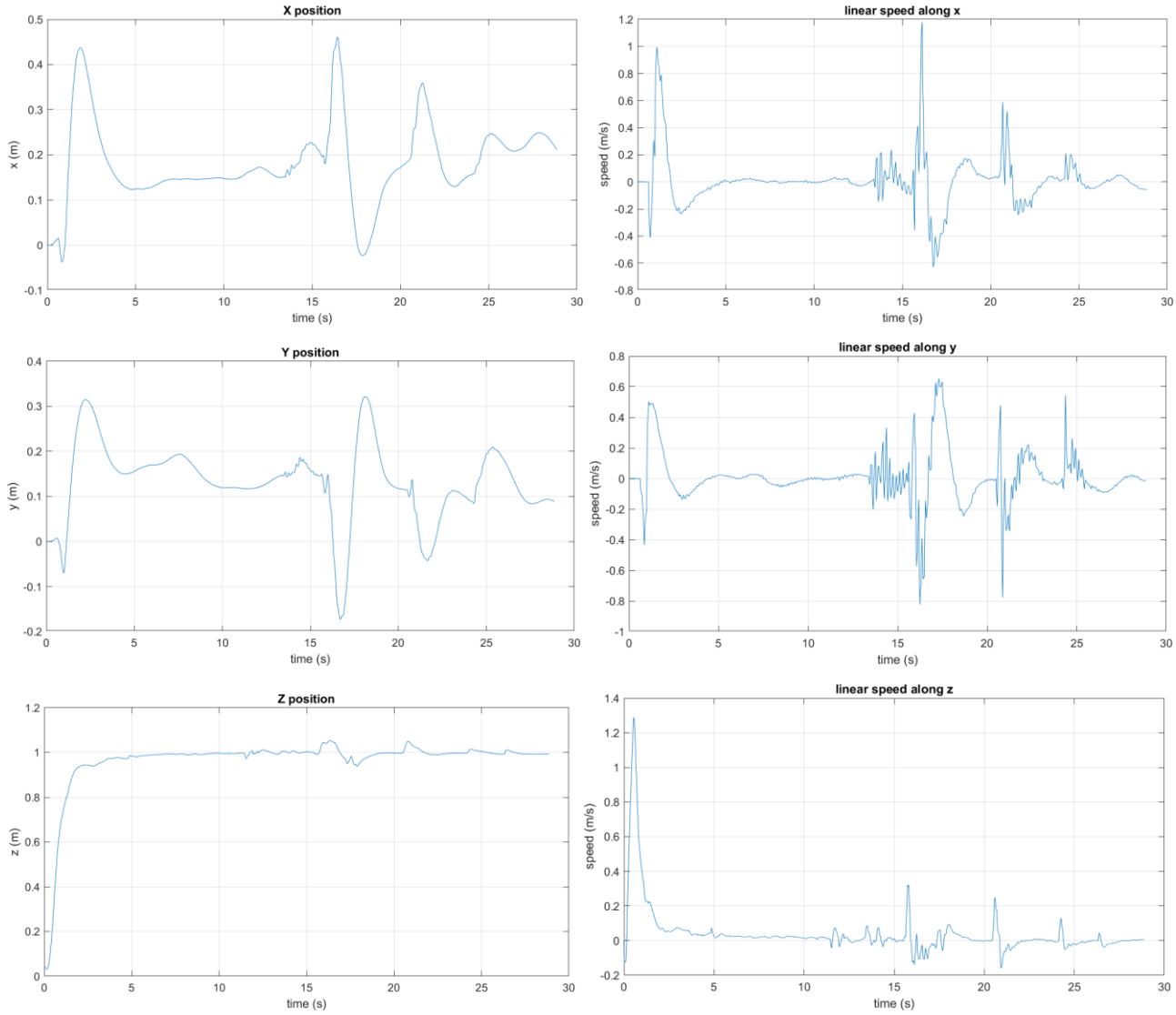


Figure 6.74: Real-time Hovering test – X, Y, Z Positions (left) and corresponding linear speeds (right)

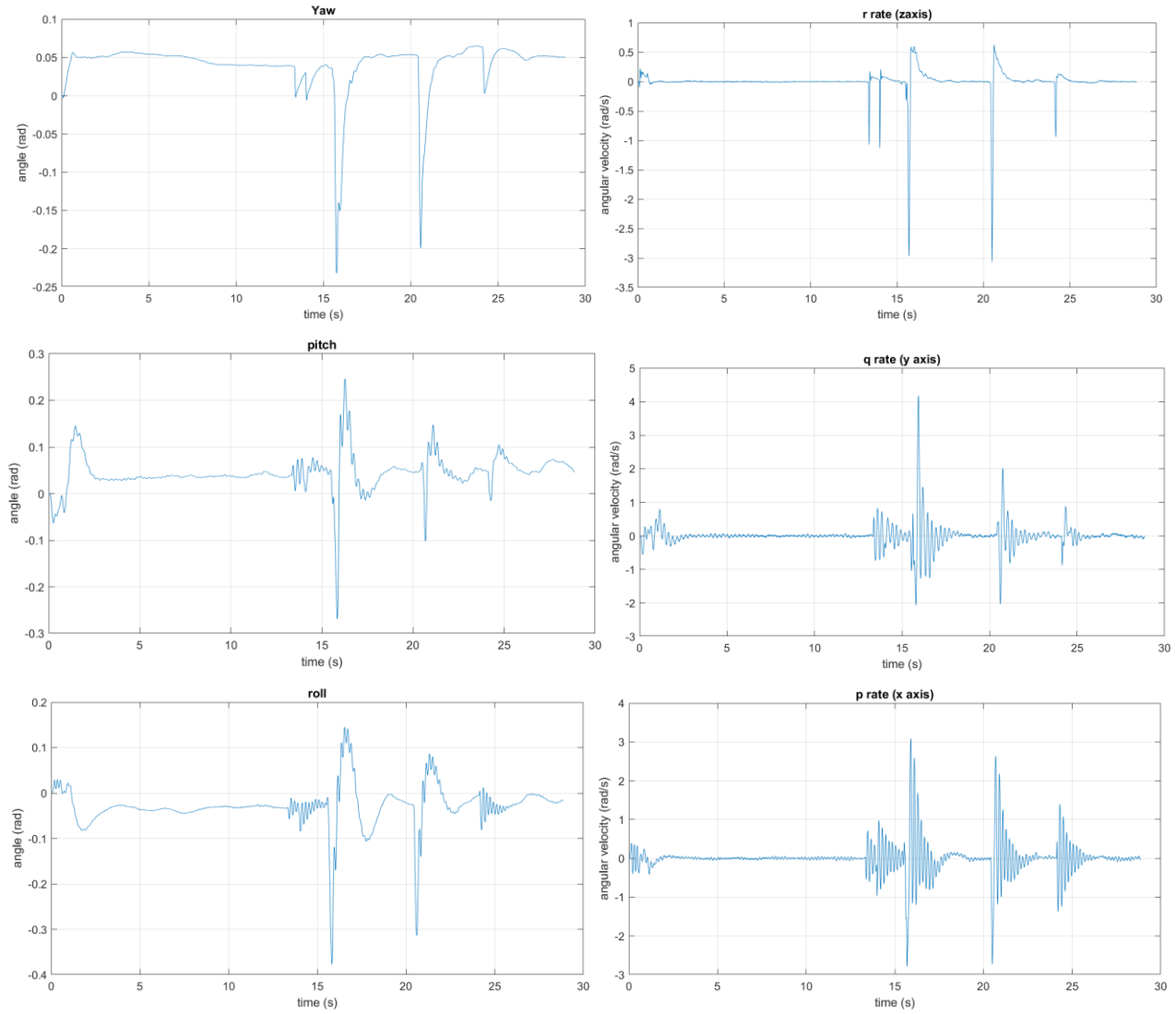


Figure 6.75: Real-time Hovering test – RPY angles (left) and corresponding angular velocities (right)

The performance of the control system response to the disturbances in the HIL test validates the controllers' gains tuned in the MIL simulation test, showing that the system is stable. Thus it is possible to proceed with the image processing and path planning algorithms design.

CHAPTER 7

1st LINE-TRACKING ALGORITHM DESIGN

The path-tracking algorithm to be designed is constituted by the **image processing subsystem** (figure 7.1) and the **path planning subsystem** within the control system (figure 7.2), which will generate the reference commands for the flight controllers. The other components of the Flight Control system like the estimator algorithm and the controllers can be left unchanged at this point, after having already tuned those in the previous design stage.

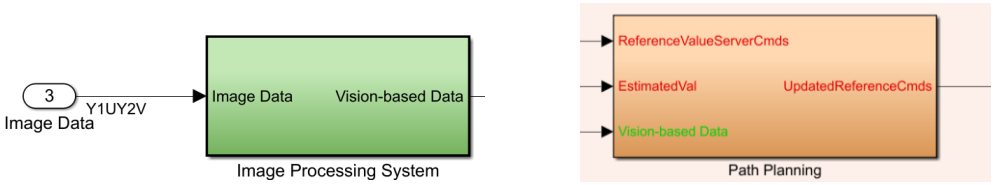


Figure 7.1: Image processing (left) and Path planning (right) subsystems

7.1 Image processing algorithm (Color Thresholder app and sub-images analysis approach)

It is better to use another model to design and test the algorithm that processes images from the camera. A new Simulink project can be opened to create it, from the main MATLAB screen, by following the procedure explained in the next lines.

Firstly, click on "New" and then "Simulink model" (figure 7.3 on the left).

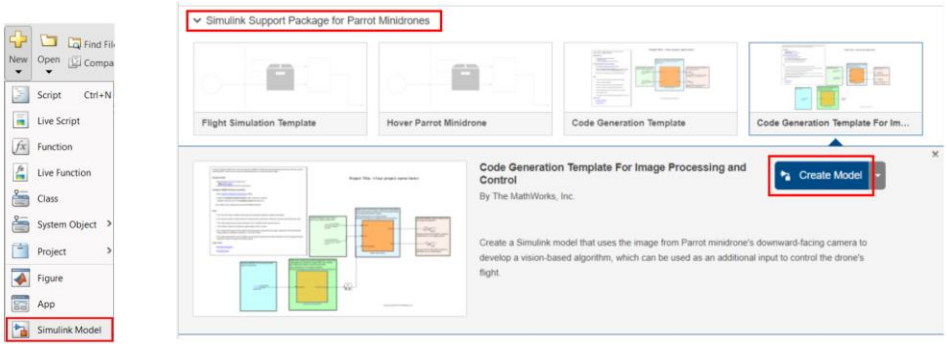


Figure 7.2: template model creation for Image processing

By opening the menu "Simulink Support Package for Parrot Minidrones", select the template "Code Generation Template for Image Processing and Control" (figure 7.3 on the right). Through this simplified model, it is possible to test the code without flying the drone independently; in fact, it will be possible to manually hold it above the path to analyze the behavior of the image processing algorithm. It would be useful to download an image of the route from the drone camera.

Firstly, the app called "***Color Thresholder***", provided by the Computer vision add-on tool in Simulink can be used to calibrate a threshold filter that creates a binary mask containing only the path. The tool can be opened by typing "colorThresholder" in the Command Window. Once the screen is open, we can select the image to be loaded (figure 7.4), or we can look for it inside the PC or from the Workspace. For our purpose, it is preferable to upload an image coming directly from the camera of the drone (with at least one route line included).

After that, we can choose between 4 different spaces: YCbCr, RGB, L*a*b*, and HSV. For our purpose, a color space with a very robust control is HSV, but since the algorithm should be made less complicated, it can be used merely RGB. HSV is a three-dimensional representation of the color based on the tonality components (Hue), Saturation (Saturation), and Luminosity (Value). It was defined in 1978 by Alvy Ray Smith is a representation closer to the form in which we humans perceive colors and their properties, unlike RGB, color shades are grouped.

To the right of the image (in the App window), we have the cursors to adjust the components of the filter. Once we reach the desired segmentation (Figure 7.4), we export the function in a .m file.

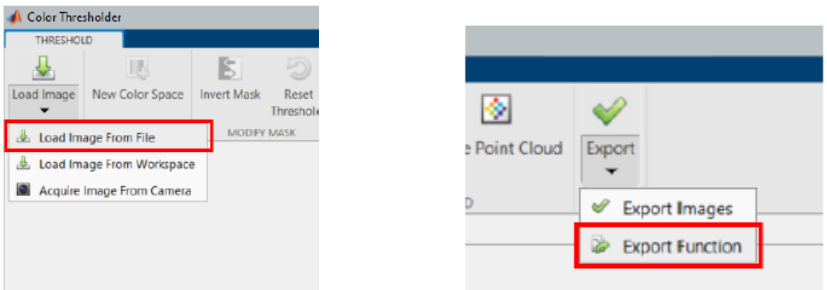


Figure 7.3: Color Thresholder app operations

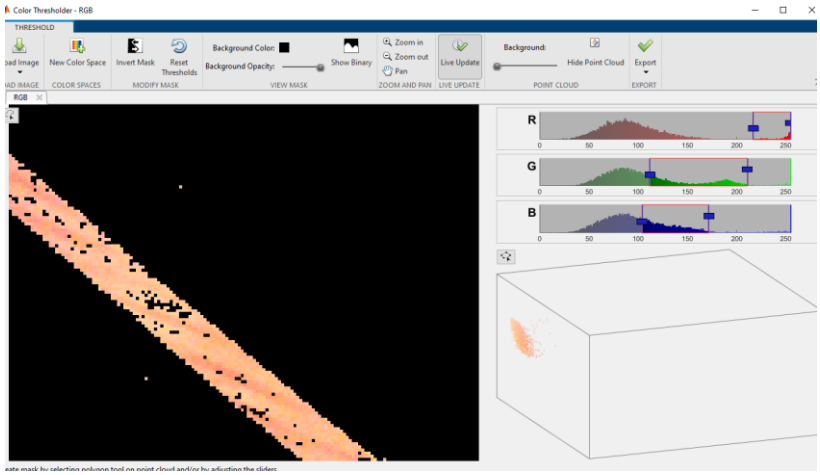


Figure 7.4: Color Thresholder app

This phase can be simplified a lot if the path is made with white tape on a black background, which is already a binary scenario. However, if other elements of different colors are added to help and to improve the Optical flow estimation algorithm, the threshold filter should be able to cancel those out.

After this procedure, we are interested in detecting the edges of the path and their angles. An edge can be modeled as a curve that shows a sudden variation in intensity. The "edge" function can be used to detect the borders. The function seeks for areas in the picture where there is an abrupt variation in intensity, through the following two criteria, as explained by Zhang et al. [39].

- The Points where the value of the 1st derivative of the intensity function exceeds a selected threshold.
- The Points where the 2nd derivative of the same function becomes null.

The “edge” function uses different estimators that implement one of the previous methods for identification. In some of them, it is possible to define if the action must be sensitive to rather vertical, horizontal, or both kinds. The function output is a binary picture assuming “1” close to the object borders, otherwise “0”.

Furthermore, another useful line detection method that can be used is the “Canny” method. The peculiarity of this technique is the use of two thresholds, one for “strong” and another one for “weak” edges. In particular, it highlights the weak ones only if there are connections to strong ones. The advantage of this technique is that

it can be influenced less likely by the noise, and it is more likely to find accurate weak contours.

For edge detection, we can use the "hough" function. It applies the SHT (Standard Hough Transform), and it can be expressed through the following parametric representation:

$$\rho = x \cdot cos(\theta) + y \cdot sin(\theta) \tag{7.1}$$

The parameter "ρ" is the distance between the edges and the reference frame's origin. "Θ", instead, can be interpreted as the orientation of the line, according to the convention shown in the next picture (Figure 7.5).

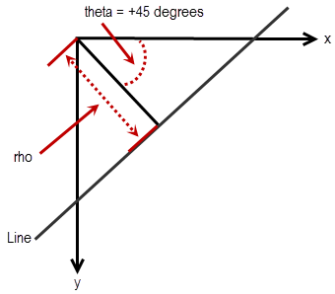


Figure 7.5: parametric representation of a line through SHT [45]

The "Hough" function generates an array of parameters θ and ρ, located respectively on the columns and rows of the matrix. Once the picture is analyzed with the SHT, the "*houghpeaks*" function is used in the parameter space to detect the peaks, which are potentials lines in the binary image. The "*FindAngle"* (Figure 7.6 and Appendix B for the script) function can be employed to estimate the edges' angles, which are then fed to the path planning algorithm.

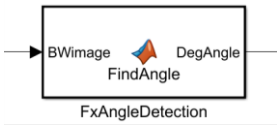


Figure 7.6: user-defined function "FindAngle"

If we look at the next pictures, we can notice that the transform calculates a null angle in the case in which the line is oriented as in the picture in the center, whereas it will calculate a negative degree if it is oriented as in the picture on the left, vice versa, a positive angle will be sent as output if the line is oriented as in the picture on the right.

Figure 7.7: Track lines with different angles (positive angle on the right)

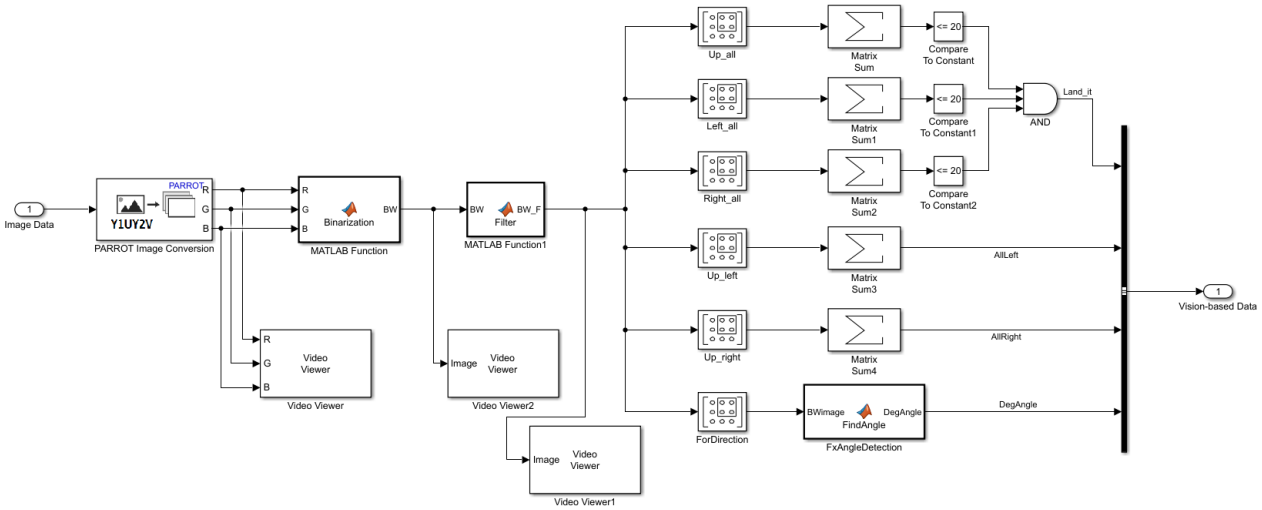


Figure 7.8: Image processing subsystem

As can be seen from figure 7.8, as input to the sub-system, we have the Image Data that comes processed by the PARROT Image Conversion block, which has an Output with the components of the image, coming from the camera, in RGB. The filter generated by the Color Threshold App has been loaded into a MATLAB block Function and has as input the RGB three-dimensional matrix and returns an image to the Output binary (in a matrix of size 160 × 120 pixels) containing "1"s where our track is present, while "0"s elsewhere.

The image was divided into various sub-matrices to analyze their pixels' binary values. In figure 7.9, it can be seen how the image was subdivided into the different areas useful for the path planning algorithm logic.

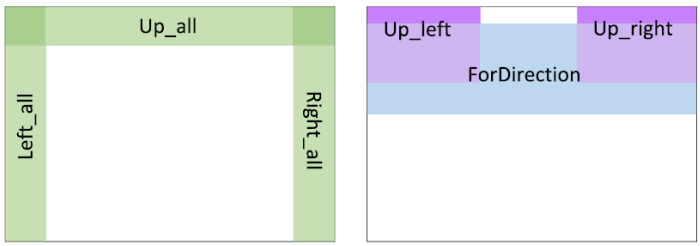


Figure 7.9: Image subdivision into areas for the path planning algorithm

Through this technique the path planning logic can establish when the drone should land above the circle.

In fact, in every sub-image of the binary image, the number of the "1"s is counted and then compared with a constant; therefore, we will have a Boolean value at the output. There are three Boolean signals which are conveyed to an AND block. This causes that the condition must be true on all three sub-images. In this case, the value of the constant to be compared with the sum of each sub-matrix is low enough to consider the residual noise from the filtered image. Through Hardware tests, this value will be adjusted so that the landing happens in the right condition.

Regarding the orientation, i.e., the part of code that acts on the Yaw angle of the aircraft, two different gains are used to control it: the first one has a higher gain to align the orientation with the line approximately, and then the second gain lower, to align the drone more precisely above the line (using the Hough transform).

However, when the line is positioned horizontally (from the image of the camera), the transform calculates an angle that could oscillate rapidly between −90° and 90°. If in input to the X-Y displacement function, we give these very oscillating values, the drone will not be able to follow the path. Therefore, a function has been placed to align faster the drone as soon as it sees the line and then continues with a more precise alignment. Two portions of the image have been dedicated to this function, called "Up_left" and "Up_right" (purple areas in Figure 7.9). A Matrix Sum block has been placed in output to the two sub-matrices to count the number of pixels; this will tell us how many white pixels of the path are present in that portion.

Instead, the portion of the image dedicated to the Hough transform analysis is the one called "*ForDirection"* (blue area in Figure 7.9).

7.2 Path planning algorithm (Stateflow approach)

Stateflow is a Simulink tool that consists of a schematic language made of truth tables, transition diagrams, flow diagrams, and state transition blocks. It allows us to design and develop supervisory logic, communication protocols, GUI (user interfaces), state machines, and hybrid systems. Through this powerful tool, it is possible to realize the sequential supervisory logic that can be modeled as a subsystem in Simulink. Various checks

during modification and execution ensure consistency and completeness of the project before implementation. In the part of the movement management system, there are two Stateflow charts, two Bus Selector, two Bus Creators, two Switches, two Gain, a Constant block, a Delay block, and a block conversion to "Single" type.

A bus arrives from the input states “estim” which is divided by a Bus selector; in this case, there is only one way out because we only need the Yaw estimate. The same goes for input Vision-based Data, but the Bus selector has multiple outputs: “*Land it*”, “*DegAngle*”, “*UpLeft*” and “*UpRight*”.

In Output, instead, we have another bus that is composed of the input bus *ReferenceValueServerCmds* together with the pos ref and orient ref channels, which respectively receive the values yes set-points x, y, z, and those of Yaw, Pitch, Roll via a Bus creator. In this case, being the constant “controlModePosVsOrient” (Figure 6.31) equal to 1, the values of Pitch and Roll cannot be given as references.

The Delay block, together with the Add block, manages the task of varying the Yaw angle. The block Delay keeps in memory the previous value that will be added (Add type block) to the variation provided by Switch1. The Switch, instead, activates and deactivates the rotation, varying between the value of the constant (equal to zero) and the output value from Chart 2. The operation of the Switch, together with the two Gain, will be explained more in-depth in the section dedicated to Chart2.

In Chart 1 is implemented the task of managing the drone's x, y, and z coordinates in space. It can be seen in figure 7.10 that it is modeled as a State machine and is divided into three states: Take Off Cruise and Landing. The picture on the next page (figure 7.11) shows the Path planning subsystem structure.

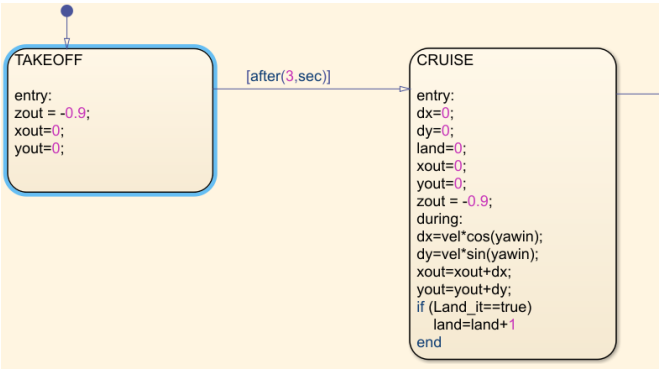


Figure 7.10: Chart 1 subsystem

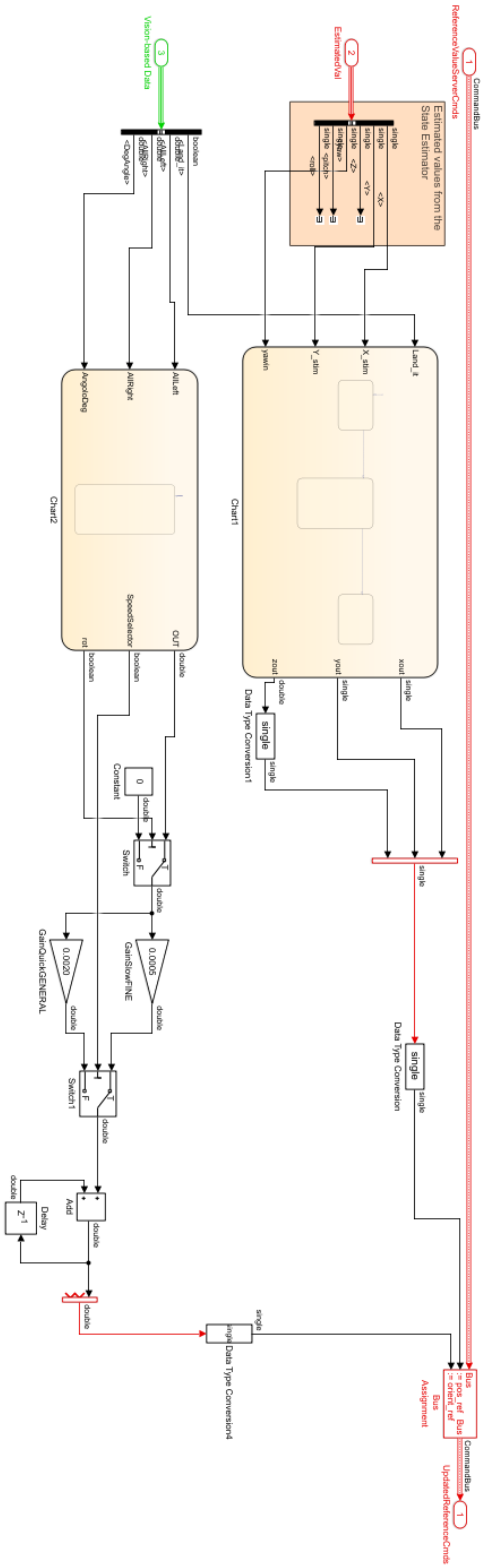


Figure 7.11: Path planning subsystem

The first state, which is the default one (the first to be performed) with the name "*TAKEOFF*" has the task of raising the drone up to 0.9 meters from the ground and assigns the value False to the variable *rot.* This variable, which will be explained in detail in the section of Chart 2, is used to "activate" the change of heading of the drone (i.e., the yaw angle). During the take-off, it is necessary to turn off rotation due to the image recorded by the camera when it is too far near the line; with the image completely white or completely black, the video processing algorithms are not accurate and could lead the drone to instability. After hovering for 3 seconds, it switches to the "CRUISE" state (this time it is defined on the body of the arrow that connects the "TAKE OFF" state with the "CRUISE" state: "[after (3, sec)]"). During the change, the value "rot" is assigned the value True, thus activating the rotation.

In the entry part of this state (which is performed only once), they are initialized to zero the variables dx, dy, and land. While during, the increase values dx and are calculated dy calculated as follows:

$$dx = vel \cdot cos(yawin) \tag{7.2}$$

$$dy = vel \cdot sin(yawin) \tag{7.3}$$

Where the variable speed is a multiplicative constant equal to $4.5 \cdot 10{-}4$. At each iteration, the values "dx" and "dy" are added respectively to the variables "xout" and "yout".

In the last three lines, the algorithm checks if the landing flag is active; if this condition occurs, the "land" variable is increased. The reason for this variable is intended to delay the landing slightly and not to create false landings if the filter does not detect the path for short moments.

When the land variable reaches at least the value 150, it enters the LANDING state. During this state, the xout and yout variables are reassigned (in a redundant form) their value, to keep the drone in the last assigned x, y coordinates. Plus, the value "zout" is increased by $2.5 \cdot 10{-}3$ meters at each iteration; in this way, the drone is lowered of altitude more slowly until it touches the floor and land above the circle.

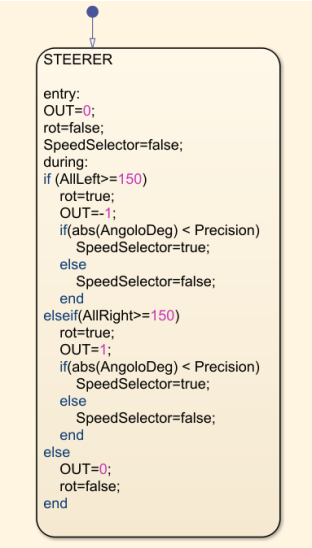


Figure 7.12: Chart 2 subsystem

Chart 2 (shown in figure 7.12) realizes the task of aligning the drone above the line of the path. It is divided into two states: “General Steerer” and “Fine steerer”.

The default state is the GENERAL STEERER. In the "entry" part, the variables: “OUT”, “p” and “r” are initialized to zero, and then, “SpeedSelector” will set as False. Instead, in the iterations of this state, two types of checks are made: the first acts on a cursor, represented by the variable p, which is decremented if the number of pixels present in the upper-left area of the picture (UpLeft) is greater than or equal to 150; conversely, the variable p will be incremented if the number of pixels contained in the upper-right area of the picture (UpRight) will be greater than or equal to 150.

In the second control, the cursor value is analyzed. If the variable p is higher than the threshold value (Threshold), the output “OUT” will be equal to 1. On the contrary, if the variable p is less than the threshold value, the output “OUT” will be equal to -1. Instead, if it is between the two values −Threshold <p <Threshold, the OUT variable will be zero. This state here is used to start the drone spinning, for the Hough transform has angle limits. We have a reasonable estimate when the angle is between −77° and 77°. When the line reaches an angle between −55° and 55°, the “END” state is activated in the Steerer. Upon entry, the variable “q” is initialized to zero, and the variable “SpeedSelector” becomes true. Now the “OUT” variable directly receives

the angle value calculated from the Hough transform. Finally, a check is carried out, that is, if the drone is aligned with the path with an error defined by the "Precision" variable, which is equal to 2°. The "SpeedSelector" Flag acts on a switch (Figure 7.12), which selects the suitable gain depending on the state assumed in Chart 2. In fact, in the "GENERAL STEERER" state, the variable OUT can only assume the values: -1, 0, and 1, while in the "FINE STEERER" state, the variable OUT takes the values between −55° and 55°. This method solves the problem of very oscillating values for the Hough transform, and it also makes the rotation speed decrease as the error angle decreases, in other words, the smaller the error angle, the more the drone turns slowly so as not to exceed the zero angle. To be considered valid, the algorithm must work on different path configurations.

7.3 Simplified Version of the previous algorithm

We can simplify the model by removing the function for angle detection and the Canny method and make the image processing algorithm rely just on the pixel count of the left and right sub-images to make the drone turn. The next figure (7.13) shows the model of the image processing subsystem.

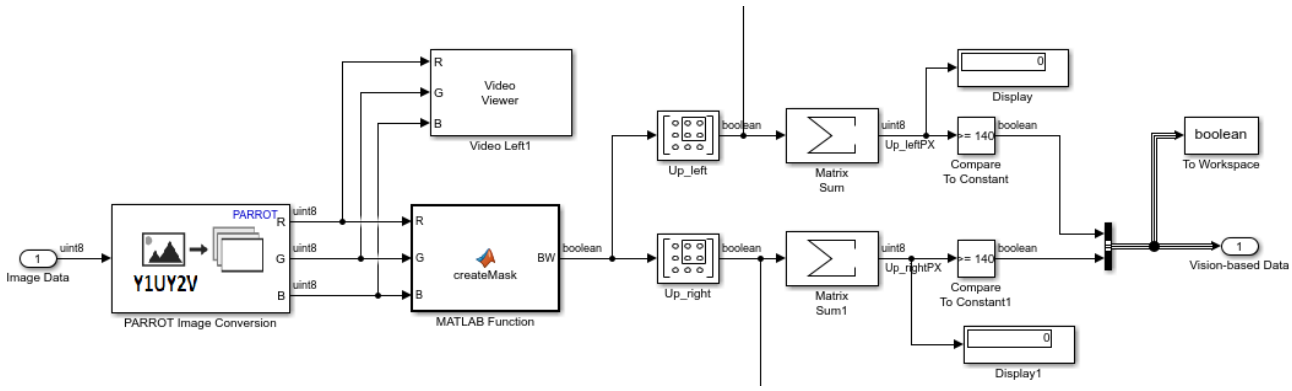


Figure 7.13: Simplified version of the Image processing subsystem

We can also simplify the Path planning algorithm, using just one gain for turning with yaw, instead of having two (one more precise than the other for small angles). Moreover, the algorithm can allow the drone to stop before turning by changing the yaw angle, and in this way, it would be slower but more precise. Figure 7.14 shows the simplified path planning algorithm.

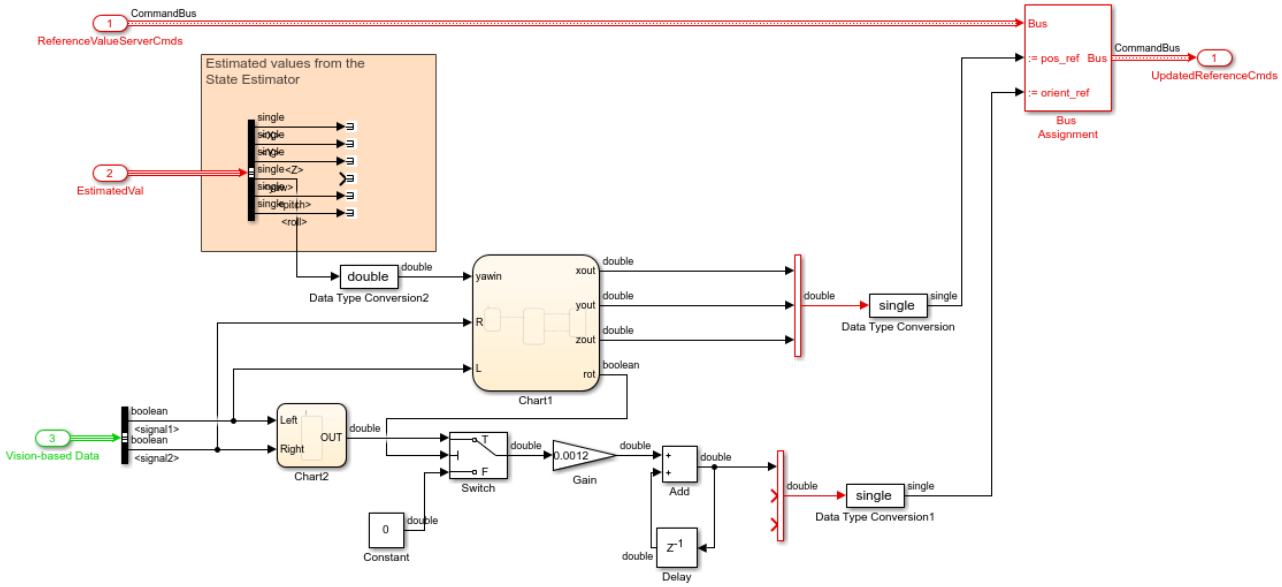


Figure 7.14: Simplified Path planning subsystem

The next picture (7.15) shows Chart1 and Chart2. In Chart1, we can notice that there is a new state called "Rotation," and it is activated only for half of a second, and it let the drone keep the same coordinates, as we said before, without changing the position while turning. It allows for more accurate maneuvers. In Chart 2, there is only a general state for steering with just one gain value with two outputs. They give the sign to the gain, depending on the direction of turning, if it is left, then "-1", otherwise "1".

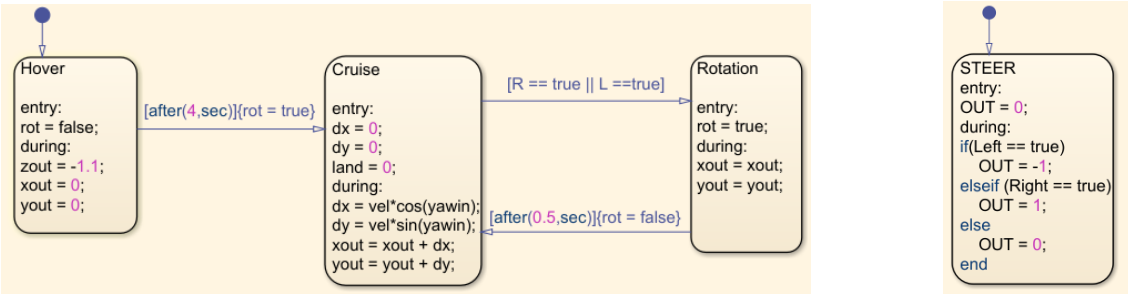


Figure 7.15: Chart1 (left) and Chart2 (right)

7.4 Model-in-the-loop test (standard and simplified implementations)

When the two algorithms were tested, they showed a satisfying performance, proving that the drone can follow the track until the end, with their design and their parameters tuned. The next pictures represent some screenshots taken from the Model-in-the-loop simulation of the standard implementation (first). The drone can precisely track the line, and it can deal with curves of different angles. Its performance relies on the tuning combination of different parameters that characterize the algorithm. There is the altitude at which the drone has to fly to see clearly the path; moreover, there are the sub-images dimensions for steering control coupled with the threshold on the pixel sum that activates the command to change yaw orientation (increasing or decreasing the angle). Another critical parameter is the drone speed, which has to be set according to the previous parameter; otherwise, the drone could oscillate too much or even lose the vision of the path in the sub-images.

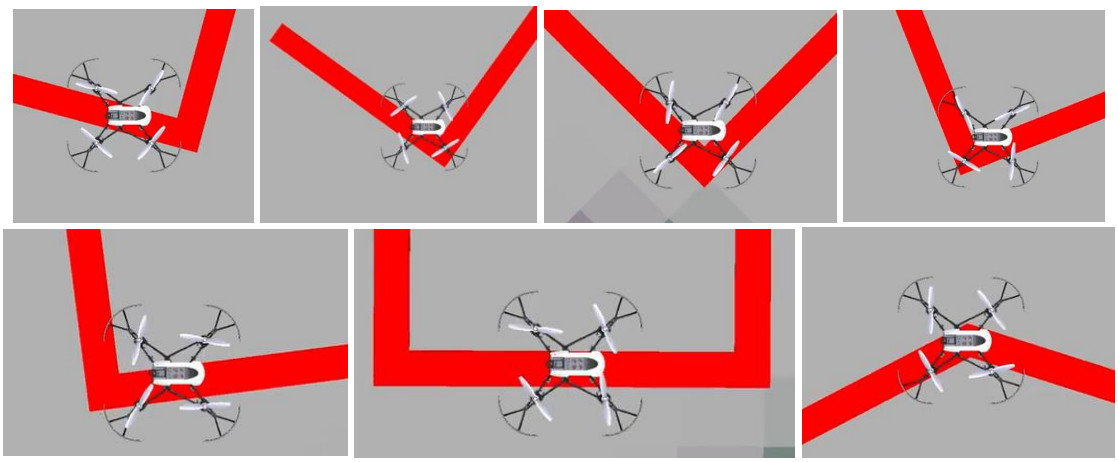


Figure 7.16: Model-in-the-loop test of the first algorithm - standard version

The simplified model is also able to track the line, but it does so with less fluidity in movements above the curves because, as was explained in the previous paragraph. If the steering control activates, the drone stops to update the x-y position with increments and change just the yaw angle, having a slower but more precise maneuver while tracking the curve. On the next page, there are tables II and III with the tuned values of the parameters that characterize the algorithm (respectively, the standard and the simplified).

TABLE II: PARAMETERS VALUE OF THE FIRST ALGORITHM

SPEED (“VEL” CONSTANT)	$4.5 * 10^{-4}$
ALTITUDE (“Z”)	1.1 m (3.61 ft)
SUB-IMAGES DIMENSION FOR STEERING CONTROL	Up_Left: rows 1-39/columns 1-55 Up_Right: rows 1-39/columns 103-157
THRESHOLD ON THE SUM OF TRACK PIXELS (≥)	140

As we can see in the next pictures (figure 7.17), the simplified algorithm makes the drone follow the curve slower and with more precision compared to the standard version.

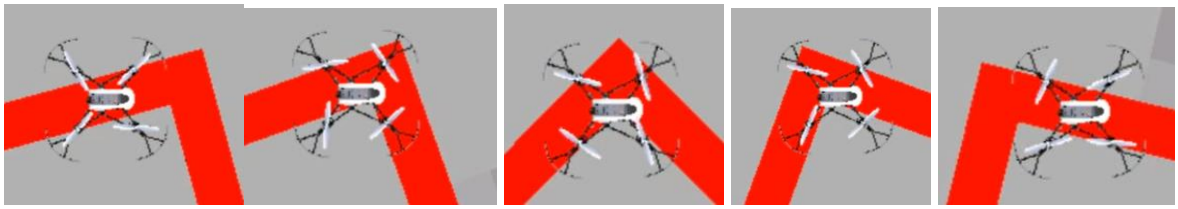


Figure 7.17: Model-in-the-loop test of the first algorithm – simplified version

CHAPTER 8

2nd LINE-TRACKING ALGORITHM DESIGN

8.1 Image processing algorithm

The 2nd Image processing algorithm implemented is made differently; in fact, it is built mainly with functions defined through MATLAB. The functions' scripts are provided in the Appendix. The only components that this algorithm has in common with the first Image processing algorithm are the first two stages. These are the "Binarization" function (figure 8.1 on the left and Appendix A.1.1), done through the "Color Threshold App" that allows us to obtain the Binary image together with path and background detection. It is followed by another filter (figure 8.1 on the right and Appendix A.1.2) that removes the noise pixels and makes the contours of the path smoother and continuous. The aim is to improve the image for the operations of the following stages.

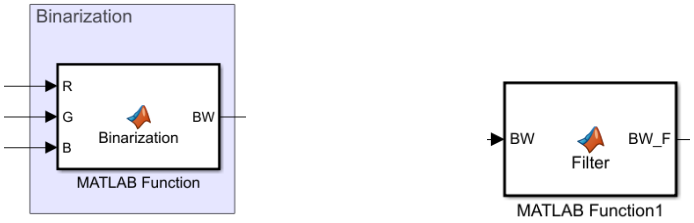


Figure 8.1: "Binarization" (left) and "Filter" (right) function blocks

At this point, the image is processed by two different branches: one needed for the path detection and tracking, the other one which tells the drone that the path has ended and identifies the landing point that, in this case, is a circle. The branch aimed to track the path is made with a function called "Ctrl_Radar" (figure 8.2 on the left and Appendix A.1.3), which tells the drone the direction to follow, generating the signals corresponding to the increments in position, once the shape of the path has been analyzed. During its cycles of path analysis, the output signals "x_incr" and "y_incr" are processed through a discrete digital filter and then

fed back as inputs for the next analysis. The increments are then normalized (figure 8.2 on the right) before being sent to the Path planning algorithm to allow more precision in the movements and a "smoother" trajectory.

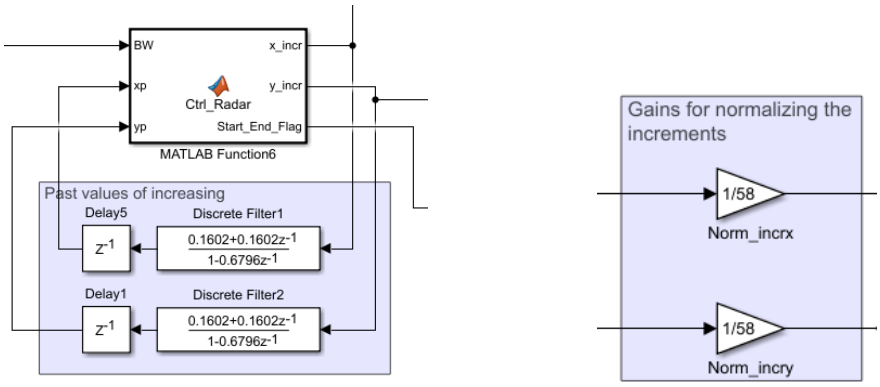


Figure 8.2: "Ctrl_radar" function with discrete filter (left) and gains for increments (right)

The signal "Start_End_Flag" tells us if the path is ended in that direction of movement or not and it is fed to the Edge detection subsystem (figures 8.3 and 8.4) that as output has a signal which indicates if the End of the line was detected, and it is included in the Bus "Vision-based data" provided as input to Path planning algorithm.

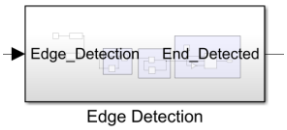


Figure 8.3: Edge detection subsystem block

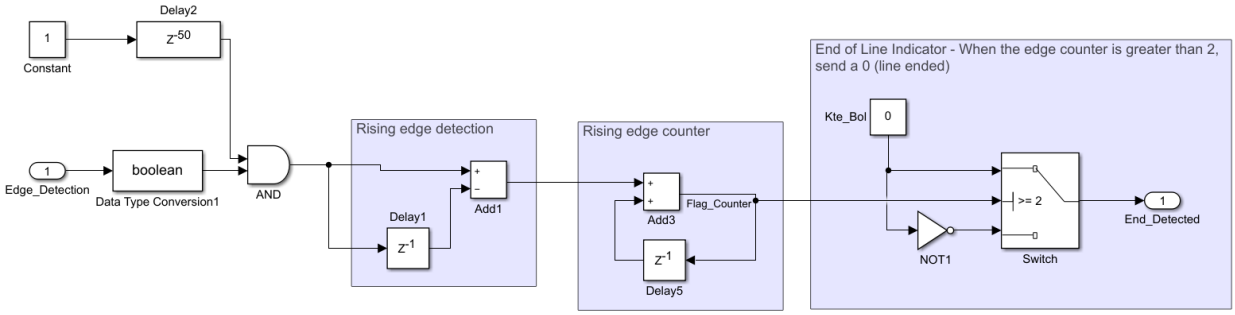


Figure 8.4: Edge detection subsystem model

The complete schematic of the Image processing subsystem is shown in figure 8.5 on the following page.

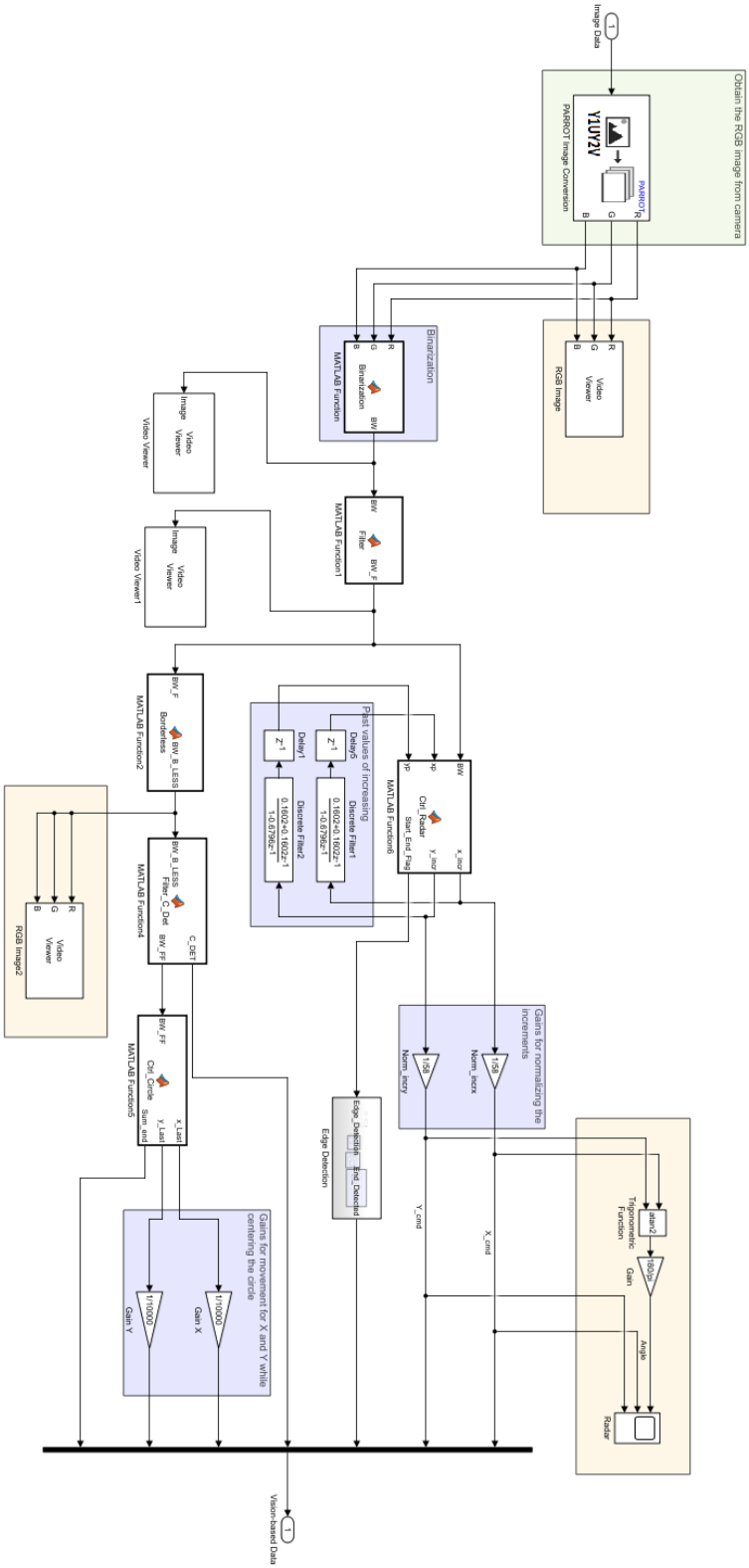


Figure 8.5: Image processing subsystem

As previously anticipated, the other branch that takes the filtered binary image as input is used to detect the circle to land on and to lead the drone to land safely on the circle. The first stage of this branch is the function "Borderless" (figure 8.6 on the left and Appendix A.1.4), which removes the objects that touch the borders of the image (only closed shapes in the image will be shown). This is needed when the image has the circle centered and the rest of the path line that does not have to influence the path planning algorithm once the path has ended, and so during the landing phase, the final portion of the path will not be visible in this image.

Then, the function "Filter_C_Det" (figure 8.6 in the center and Appendix A.1.5) is a filter that threshold the image removes noise pixels, and detects the final circle when almost centered (in the central area of the image).

Next, the function "Ctrl_circle" (figure 8.10 on the right and Appendix A.1.6) counts the pixels of the circle, subdividing the image into four quadrants and counts the "unbalanced" pixels between left and right halves and between upper and lower halves. So it gives the errors of the X-Y position of the circle center compared to the center of the image (this algorithm works with all symmetric shapes like squares or rectangles, not just circles).

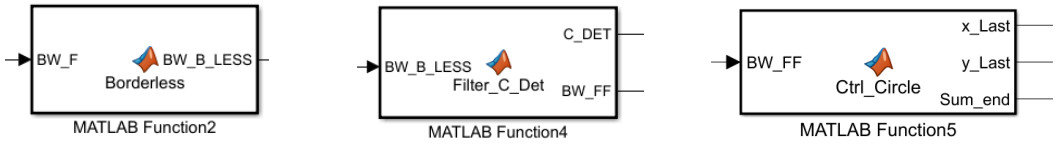


Figure 8.6: "Borderless" (left), "Filter_C_Det" (center) and "Ctrl_Circle" (right) functions

The signal "C_DET" is a boolean value that is true if the circle is detected in the center of the image, and it is fed to the Path planning algorithm to activate the landing phase on the circle. The signal "Sum_end" represents the sum of the pixels of the circle.

Finally, the output signals "x_Last" and "y_Last", as the increments of tracking function are normalized before being fed to the Path planning algorithm, so these signals are multiplied by the gains needed to control the X-Y movements more precisely while centering the circle (figure 8.7).

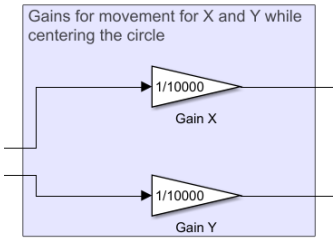


Figure 8.7: Gains for positioning phase before landing

8.2 Path planning algorithm

The 2nd Path planning algorithm is also very different than the 1st solution because the latter was implemented through State machine representation and Stateflow, while the following one is implemented through user-defined Matlab functions. The Vision-based data bus coming from the Image processing algorithm contains the X-Y increments during tracking and the circle increments for positioning during landing. They are filtered with a Butterworth 1st order filter with a cut-off frequency at 0.3 Hz (figure 8.8). It is used to filter out the noise from the signals before they are processed in the Path planning subsystem logic.

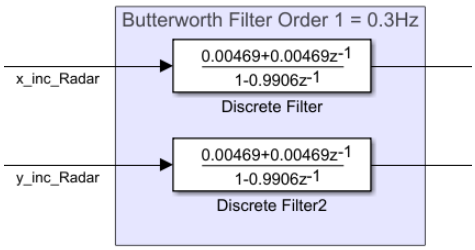


Figure 8.8: Butterworth Filter implementation

Figure 8.9 on the next page shows the complete schematic of the Path planning subsystem.

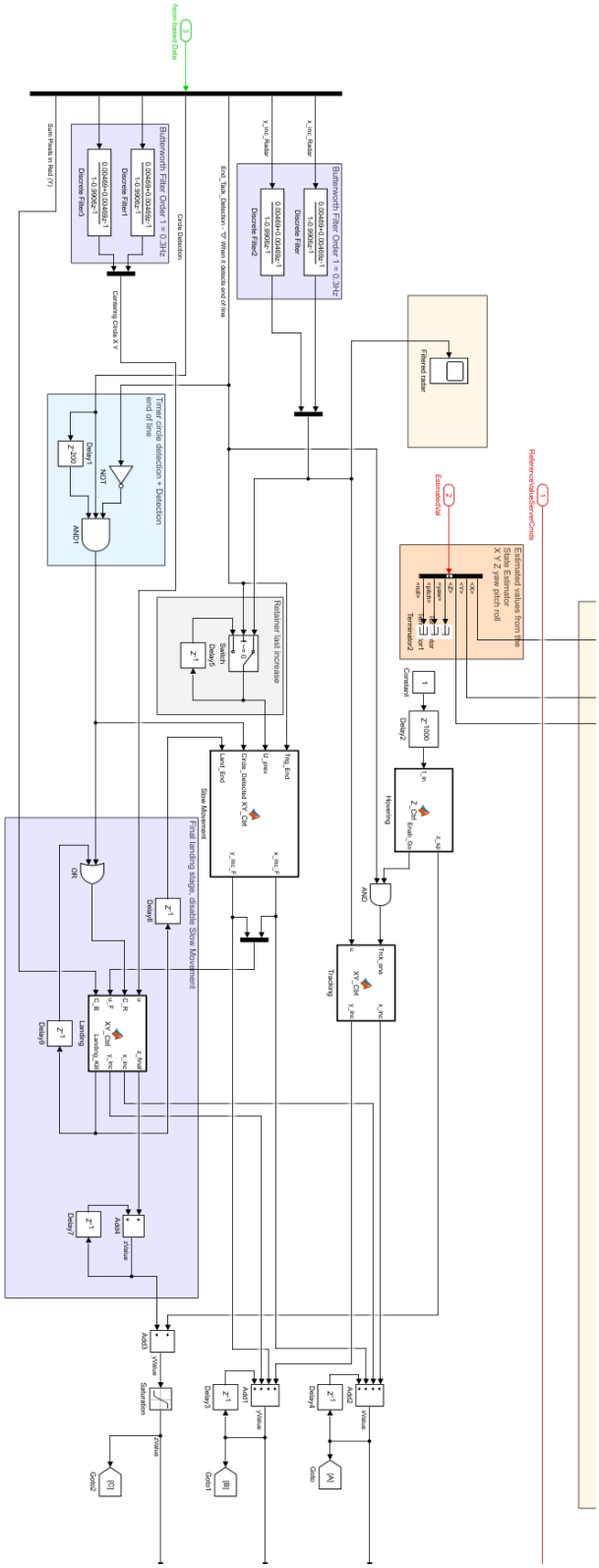


Figure 8.9: Path planning algorithm

The function “Z_Ctrl” (figure 8.10 and appendix A.2.1) provides a constant Z reference for the drone that must hover at a fixed altitude.

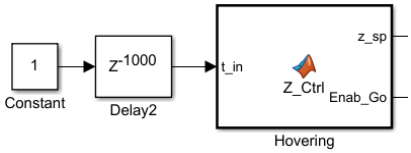


Figure 8.10: Altitude planning

After 5 seconds (1000*Ts) the tracking state is activated, and if both the signals “Enab_Go” and “End_Track_Detection” are both true (1 if the end of the track is not detected), the tracking function “XY_Ctrl” (figure 8.11 and appendix A.2.2) is activated and multiplies the position signals by the gains selected in the function.

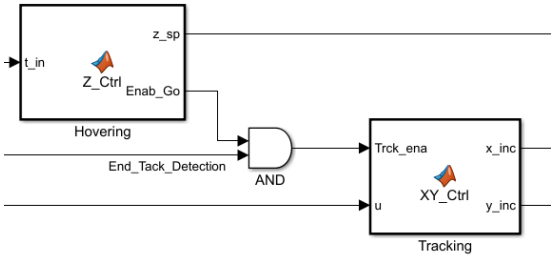


Figure 8.11: Tracking function

If the circle is detected for more than 1 second, and the end of the track is detected with the signal becoming 0 (figure 8.12), the slow movement tracking function is activated to center the circle before and during landing (figure 8.13 and appendix A.2.3).

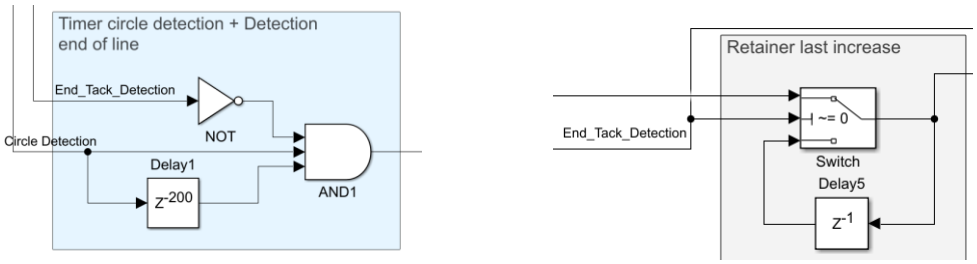


Figure 8.12: End of line detection (left) and logic for switching to landing phase (right)

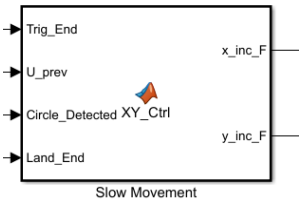


Figure 8.13: Tracking with slow movements for positioning

The function "XY_Ctrl" for landing (figure 8.14 and appendix A.2.4) disables the slow movements tracking and make the drone reduces the altitude until it touches the ground.

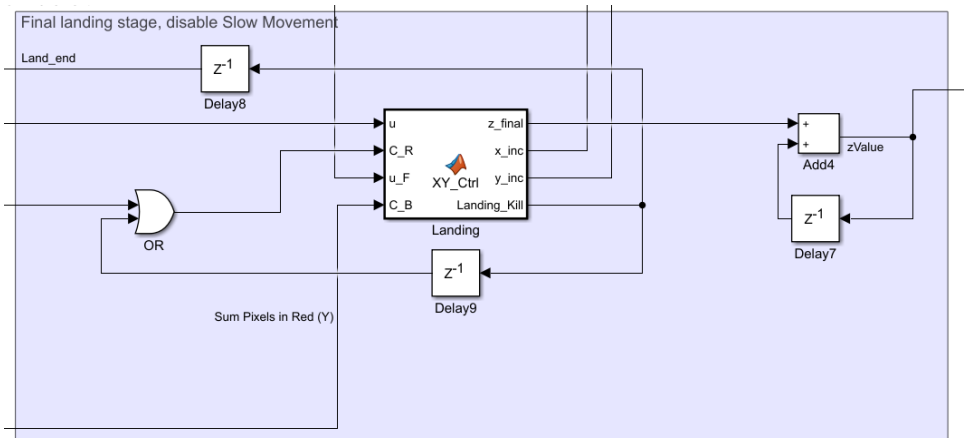


Figure 8.14: Landing control block

At the end of the path planning stages, all the increments in position signals X, Y, and Z are summed up and generates the reference signals for the controller subsystem.

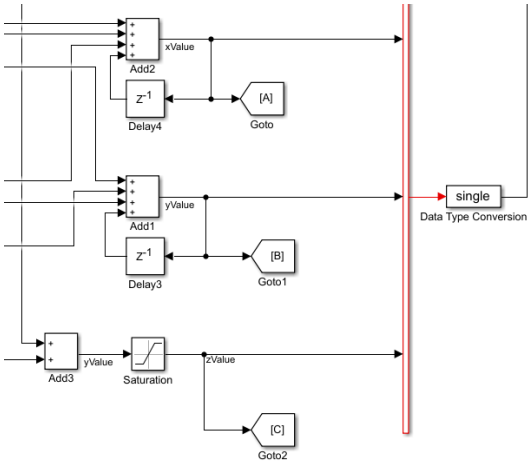


Figure 8.15: position signals generation with increments addition

8.3 Model-in-the-Loop test with Simulation

The algorithm was tested with a simulation in the Simulink 3d environment, and the next pictures show the track used (figure 8.16) and the screenshots taken from the Model-in-the-loop test, showing that this algorithm does not rotate the drone but make it track the line just according x-y position. The algorithm can follow the line, and it lands on the circle at the end, satisfying the requirements.

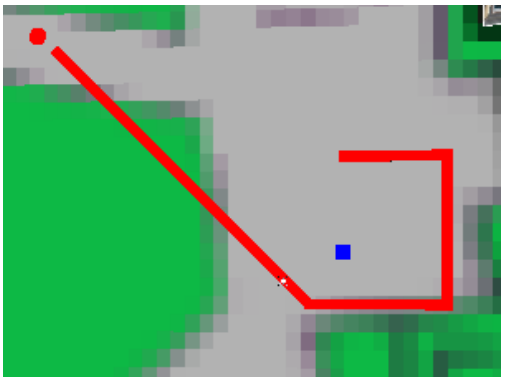


Figure 8.16: track of the simulation test for the second algorithm

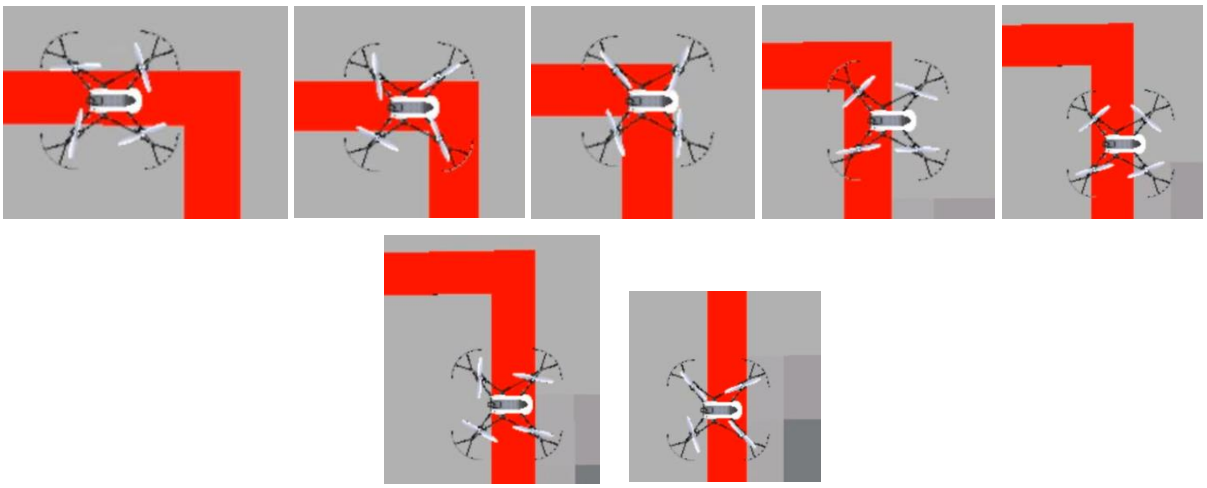


Figure 8.17: Model-in-the-loop test of the second algorithm

The pictures on the next page (figure 8.18) represent the signals over time of filtered increments (dx and dy) coming from the image processing subsystem and the position coordinates of the drone relative to the earth frame.

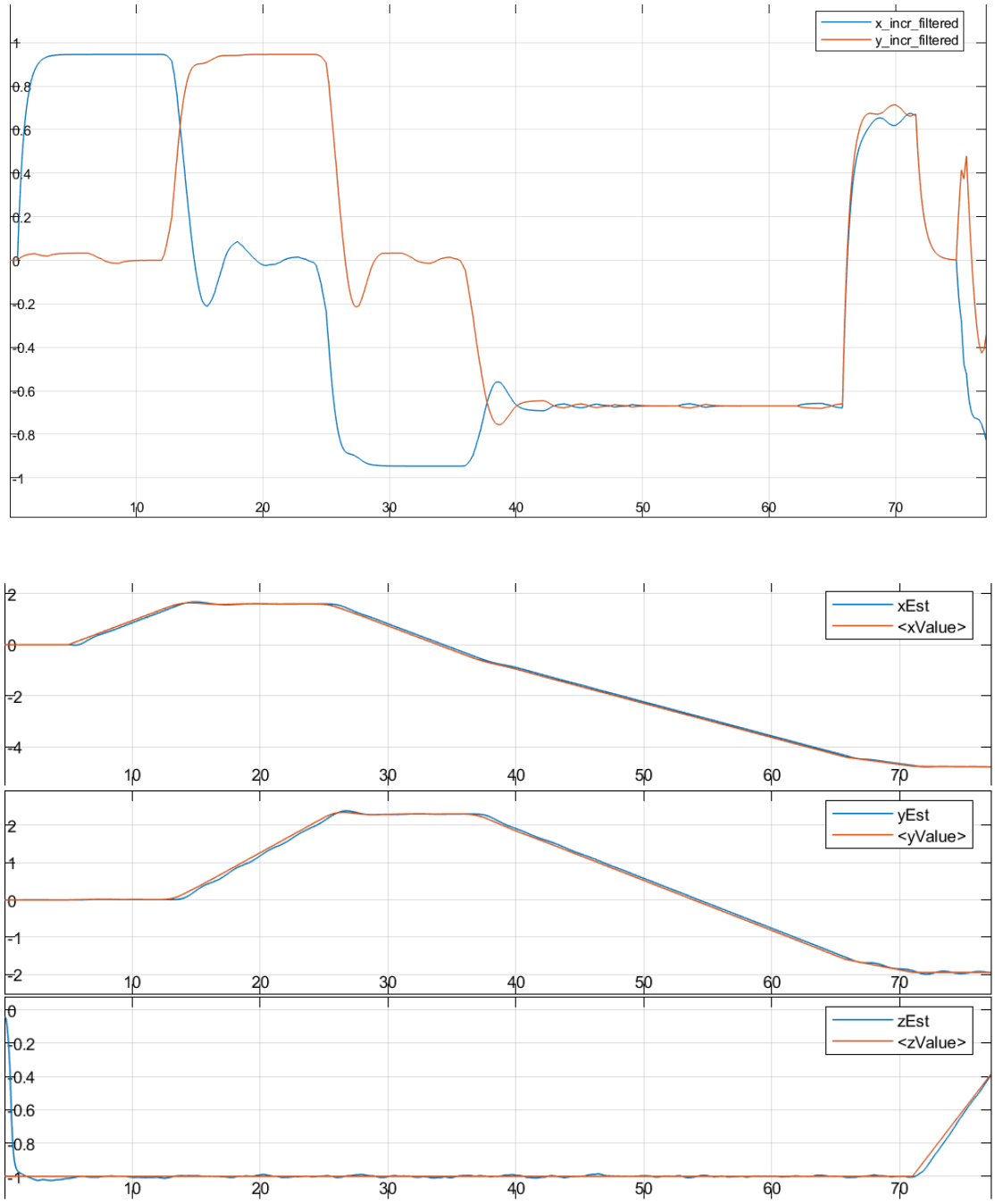


Figure 8.18: x-y increments signals filtered (top) and x-y-z position coordinates of the drone (bottom)

CHAPTER 9

HARDWARE IN THE LOOP TEST AND VALIDATION

9.1 Environment setup and HIL test of first design version (with issues explanation)

The last phase of Model-based design consists of deploying the algorithm on the Hardware and test it in real-time. The environment for the test was prepared, as shown in figure 9.1. The only available space has the minimum required dimensions to fly the drone safely, and the presence of other objects around the path and lighting conditions not easy to adjust has created many difficulties in preparing the tests. This is due to the extreme sensitiveness of the ultrasound sensors and camera also because of the low-cost hardware according to the project goal.

Figure 9.1: Example of the track in the environment used for the HIL Test

The path was made with white tape on a black background so that the contrast between these two is optimized, and these conditions can favor the Image processing algorithm. The use of the black cloth was also justified because initially, the floor was too glossy, and it created too much noise and disturbance for the camera and the optical flow estimation algorithm.

Moreover, the optical flow estimation was helped by putting the red pieces of paper randomly on the floor

to create a pattern that can be a reference for the optical flow algorithm (figure 9.2). In fact, without these pieces, the absence of other objects and shapes other than the path could confuse the estimator, and the flag can be activated terminating the flight, or if we deactivate the flag previously, we can see the drone drifting away from the path. However, the pattern should be filtered out by the image processing algorithm, and the binary image must not show the pattern shapes as white pixels.

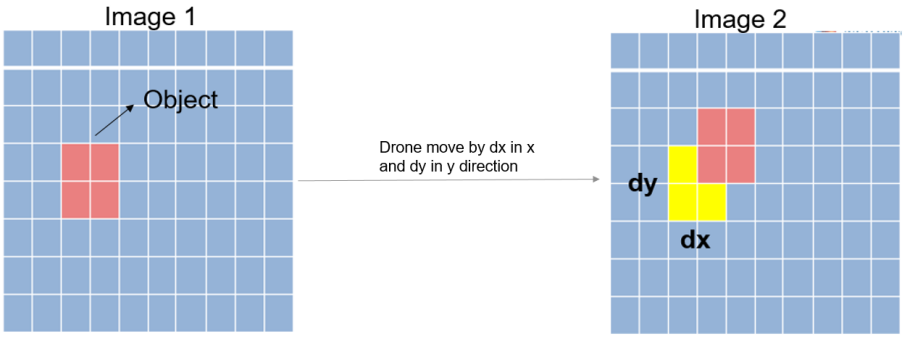


Figure 9.2: Optical flow estimation [40]

Dealing with sensors measurements and estimation algorithms in the real-time test is very different than in simulation. In the hardware-in-the-loop test, the most problematic aspects are the disturbances on the ultrasound sensor that can lead the drone to drift up to the roof or the noise of the low-resolution camera coupled with light conditions.

While the first design implementation worked satisfactorily in the simulation, the real-time test highlighted issues connected with the factors previously treated, and it also showed how sensitive was the image processing algorithm to them. This implementation relies on pixels count. As it was explained in chapter 7, the drone only turns when the number of pixels overcomes a certain threshold, meaning they are pixels of the track and not noise pixels. The issue occurs if the camera detects, for example, a large bright zone of noise due to light reflection on an object. It also worsens if we notice that the drone orientation can influence the colors' shades, in our case the white color of the path could appear as grey and can be filtered out by the binary filter (figure 9.3), or some zones of the pieces of paper of the pattern could appear brighter, and so they could be detected instead as part of the path by the image processing subsystem.

Figure 9.3: Issue related to wrong filtering of the track due to darker regions of the path

Another issue is related to altitude estimation. In the model-in-the-loop test, the drone was set to fly at a constant altitude. Therefore the parameters, like the threshold value or the dimensions of the sub-images used for steering, were tuned accordingly to the path dimensions seen by the camera at that particular altitude.

However, during the real-time test, the altitude is subject to variations or oscillations due to the ultrasound sensitivity, and so if the altitude changes coupled with pitch and roll motion for disturbances, the track shape can be seen very distorted, greater or smaller, especially if the drone hovers too close to the ground. Therefore the pixel count technique is less robust, and the assumptions according to which the drone hovers at a fixed altitude and roll and pitch angles do not affect path shape could not be valid in this case. Furthermore, the other characteristic parameters of the algorithm, like speed, sub-images dimensions, and threshold, are tuned according to the altitude. Thus even a small change in altitude could meaningfully affect the performances and the effectiveness of the algorithm.

The sensor data plots of the real-time test of the first implementation are shown in the pictures on the next page (figure 9.4 to 9.6). The trajectory plot from the test data shows that in the first curves of the track, the drone approximately follows the path, oscillating frequently and changing yaw attitude in an irregular way along the path, resulting in less precision and more time taken to accomplish the task. Furthermore, the drone was often shut down during the test by the flag activated due to the wrong altitude estimation of the ultrasound sensor for the oscillations in the yaw motion. Therefore, to improve the poor performances in the real-time test, it was decided to design another version that tries to overcome the weaknesses of the first algorithm with

another implementation of Image processing and Path planning subsystems (2nd line-tracking algorithm explained in chapter 8).

TABLE III: ANALYSIS OF THE TIME OF COMPLETION OF THE TRACK (1st ALGORITHM)

	TIME OF COMPLETION OF THE TRACK [s]		
	AT LOWER SPEED "vel" = 0.00040	AT MEDIUM SPEED "vel" = 0.00045	AT HIGHER SPEED "vel" = 0.00050
TEST 1	60	48	43
TEST 2	57	50	41
TEST 3	55	45	46
TEST 4	59	54	50
TEST 5	54	49	49
MEAN VALUE	57	49.2	45.8
STANDARD DEVIATION	2.28	2.92	3.49

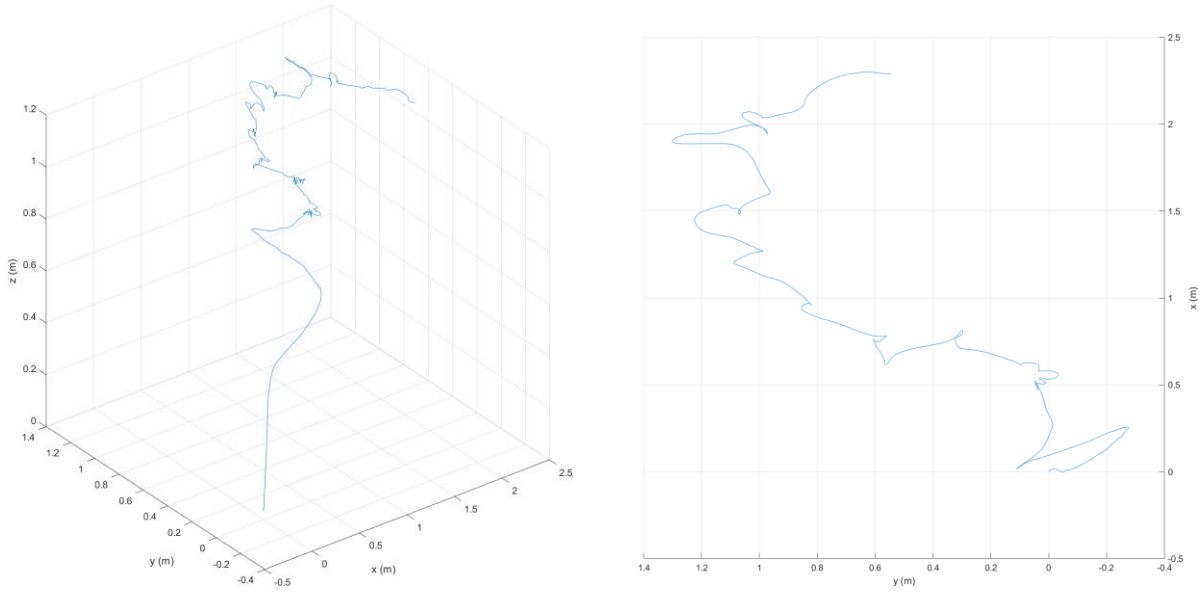


Figure 9.4: HIL test with 1st algorithm – Trajectory (3D view and Top view)

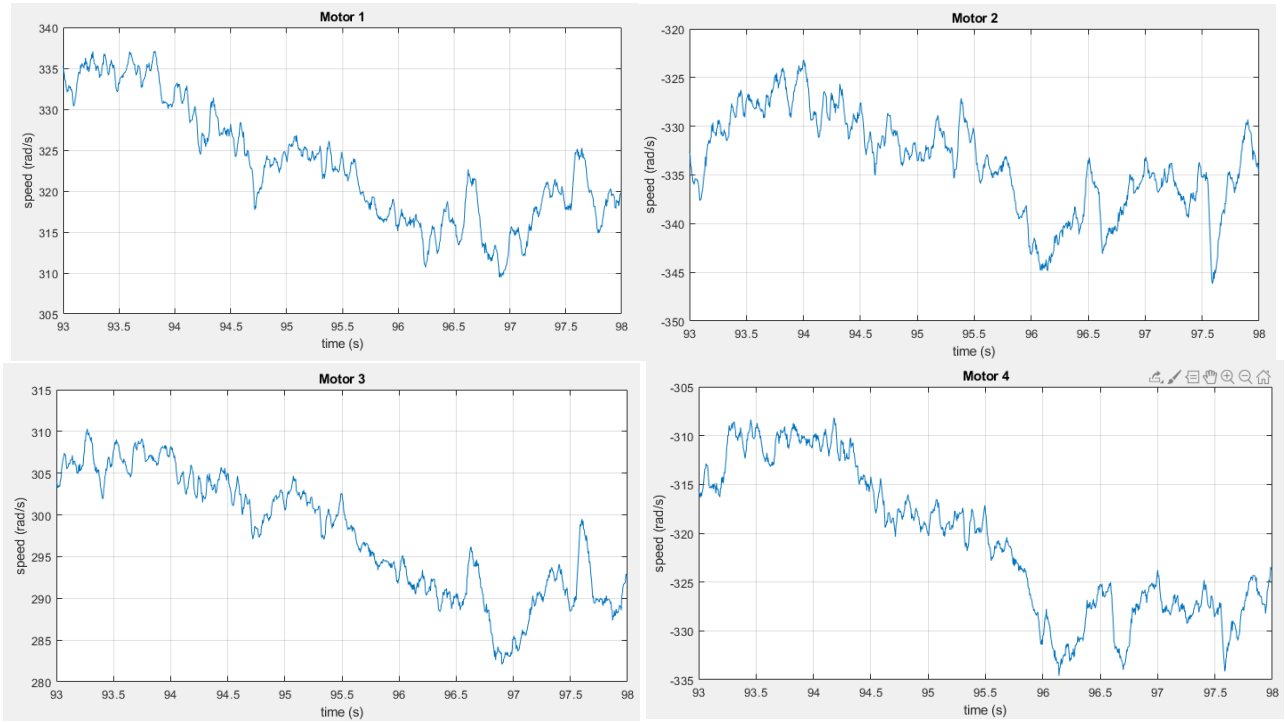


Figure 9.5: HIL test with 1st algorithm – Motor speeds

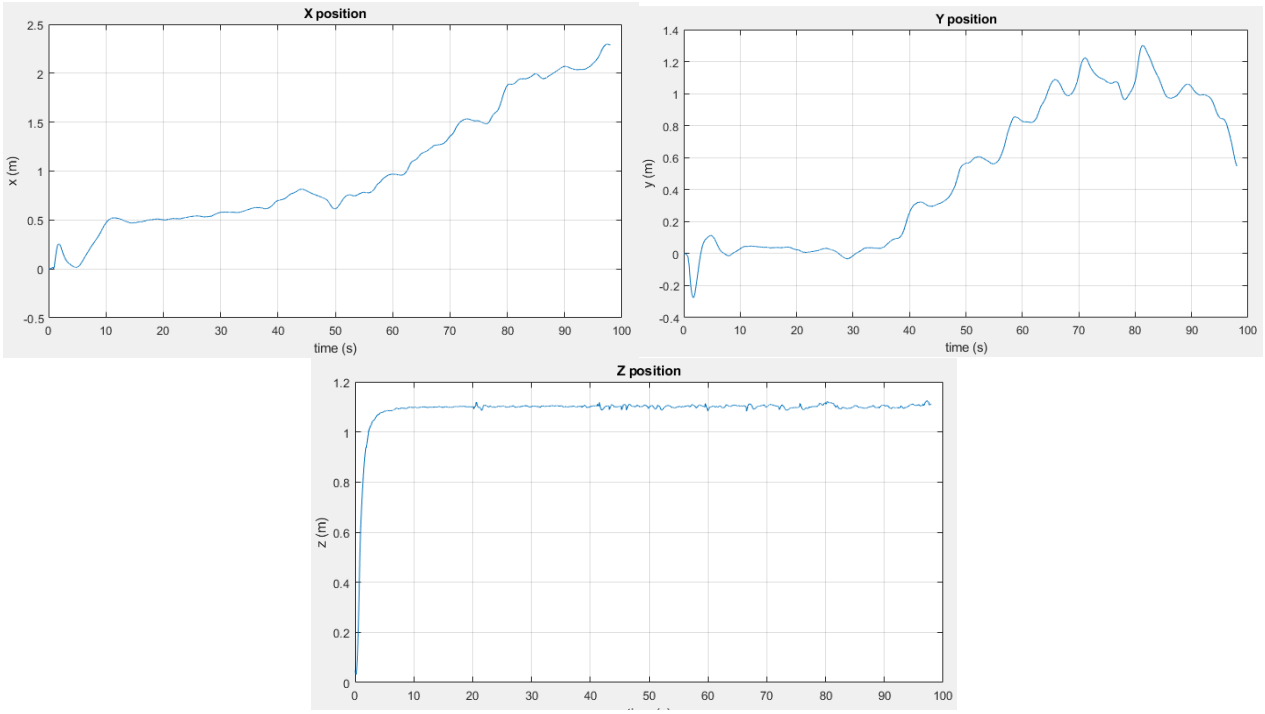


Figure 9.6: HIL test with 1st algorithm – X, Y, Z Position and Yaw angle

9.2 HIL Test of the second design version

Due to the lower reliability of the first algorithm in the real-time test, the second algorithm was implemented according to the technique explained in chapter 8. In this test, the 2nd algorithm revealed to be affected minimally by the issues described in the first paragraph, validating the effectiveness and reliability of it compared to the first that showed more difficulty to follow the track in a real-time test.

The next picture (9.13) shows some screenshots taken from a video that proves how the drone is capable of following the line, and land at the end of the line on the circle satisfying the project goal.

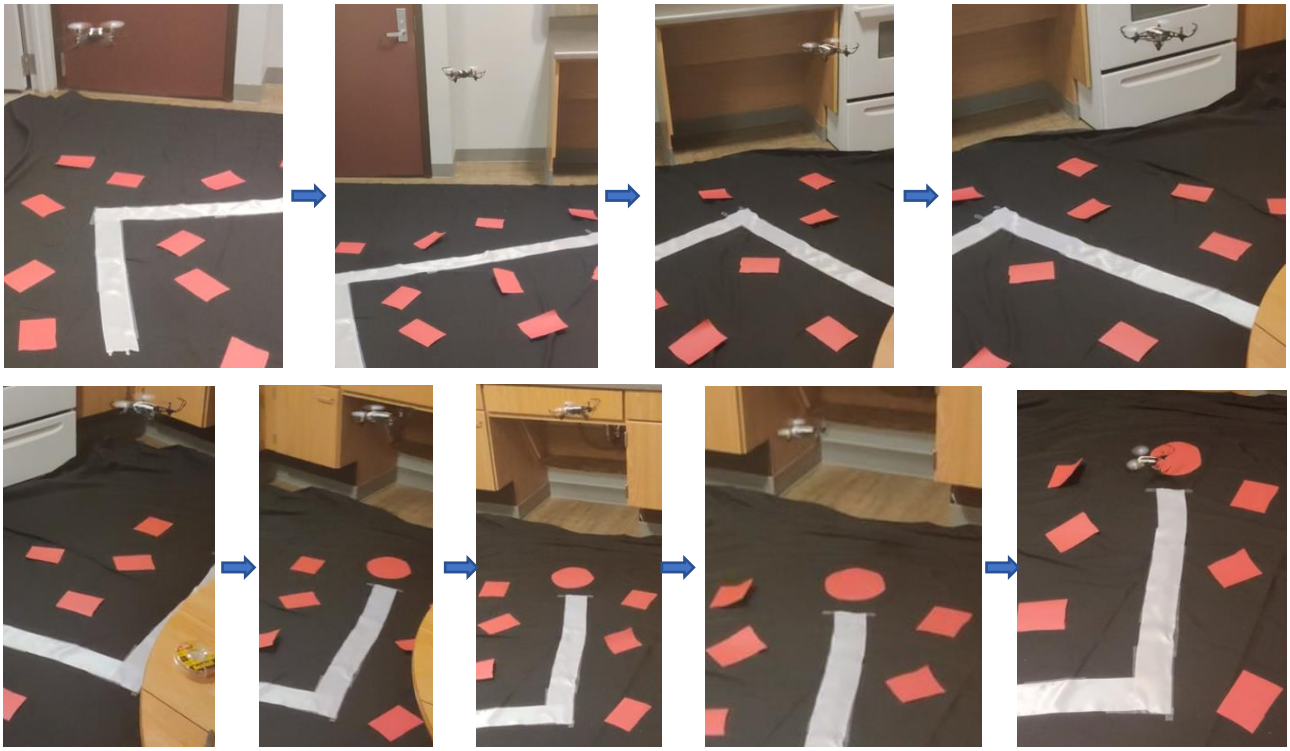


Figure 9.7: screenshots of the HIL test with 2nd algorithm

(Link to the video: https://youtu.be/ehN1Mk-clvs)

The sensor data plots of the real-time test of the first implementation are shown in the next pictures.

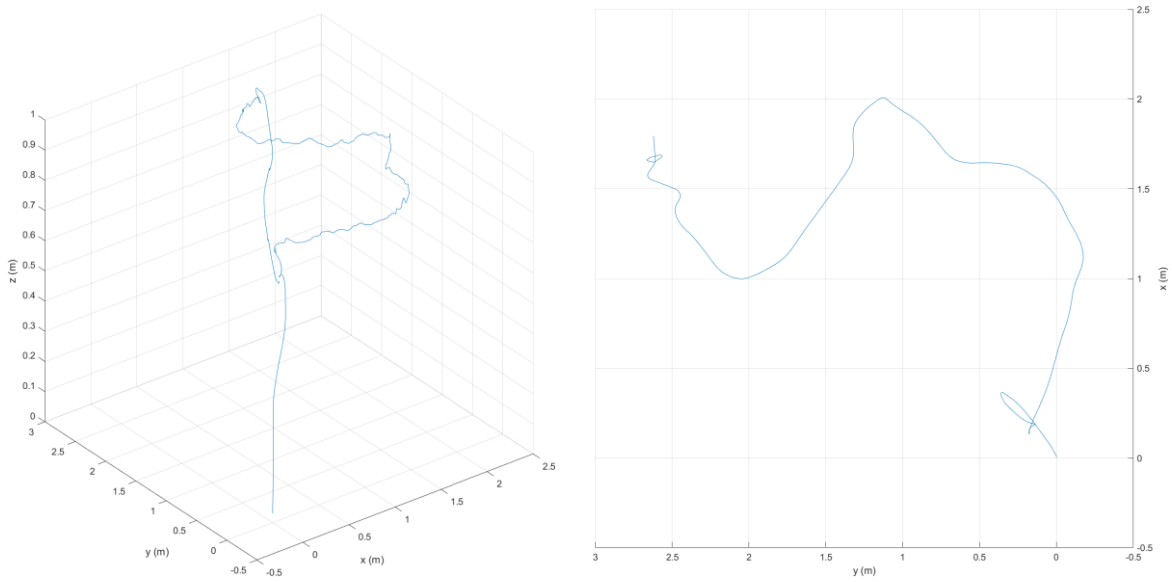


Figure 9.8: HIL test with 2nd algorithm – Trajectory (3D view and Top view)

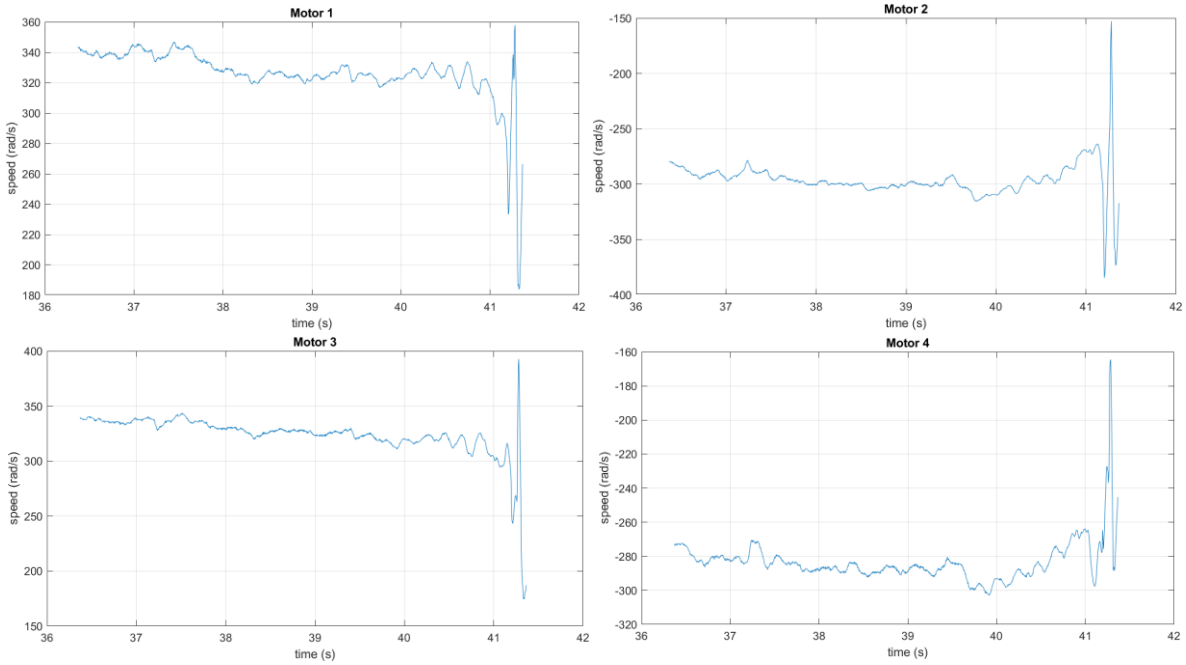


Figure 9.9: HIL test with 2nd algorithm – Motor speeds

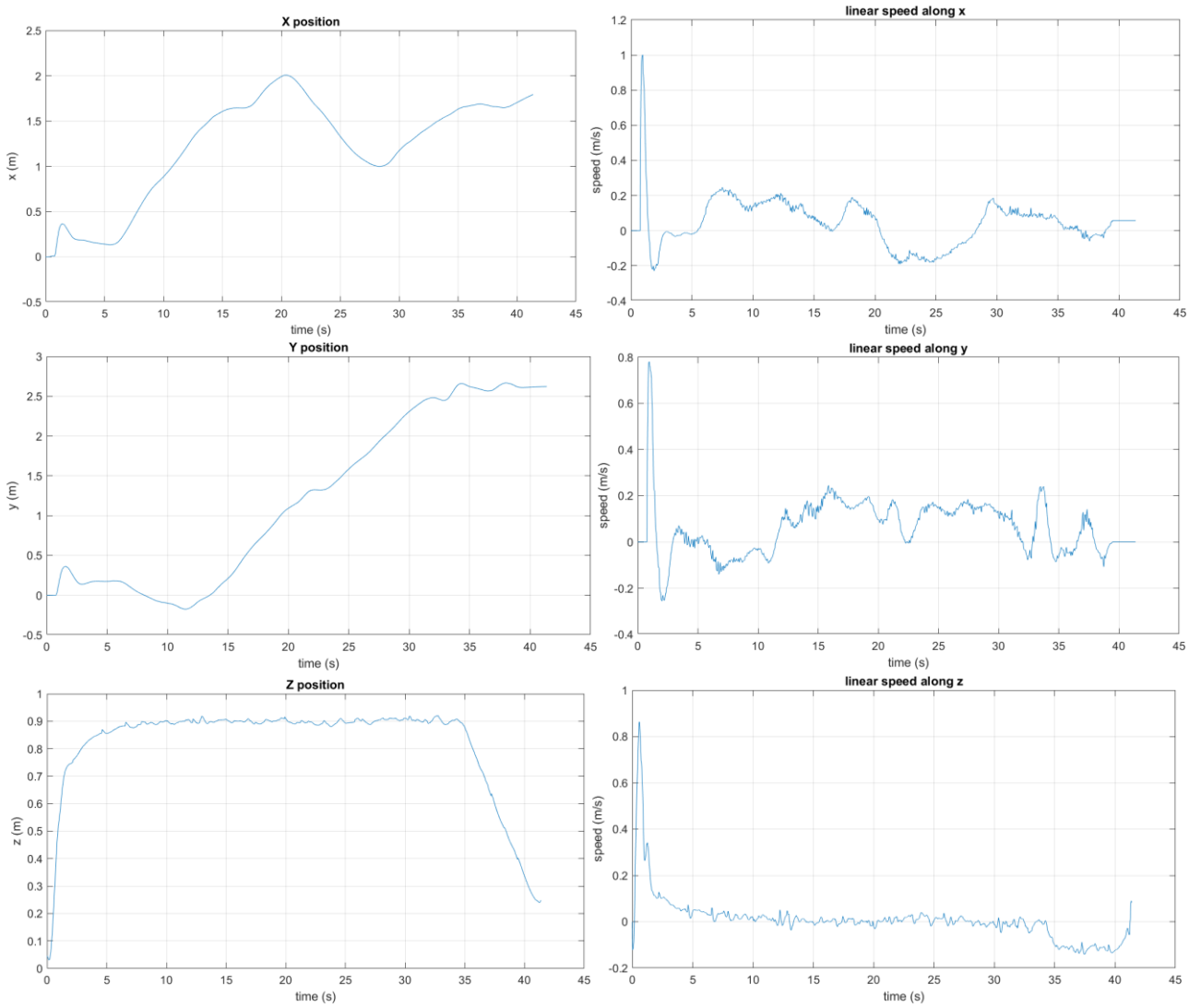


Figure 9.10: HIL test with 2nd algorithm –X, Y, Z Positions (left) and corresponding linear speeds (right)

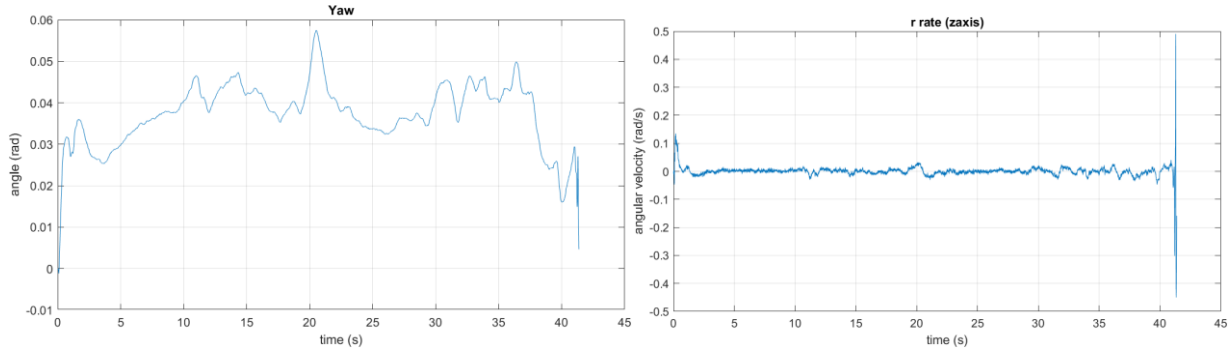


Figure 9.11: HIL test with 2nd algorithm –RPY angles (left) and corresponding angular velocities (right)

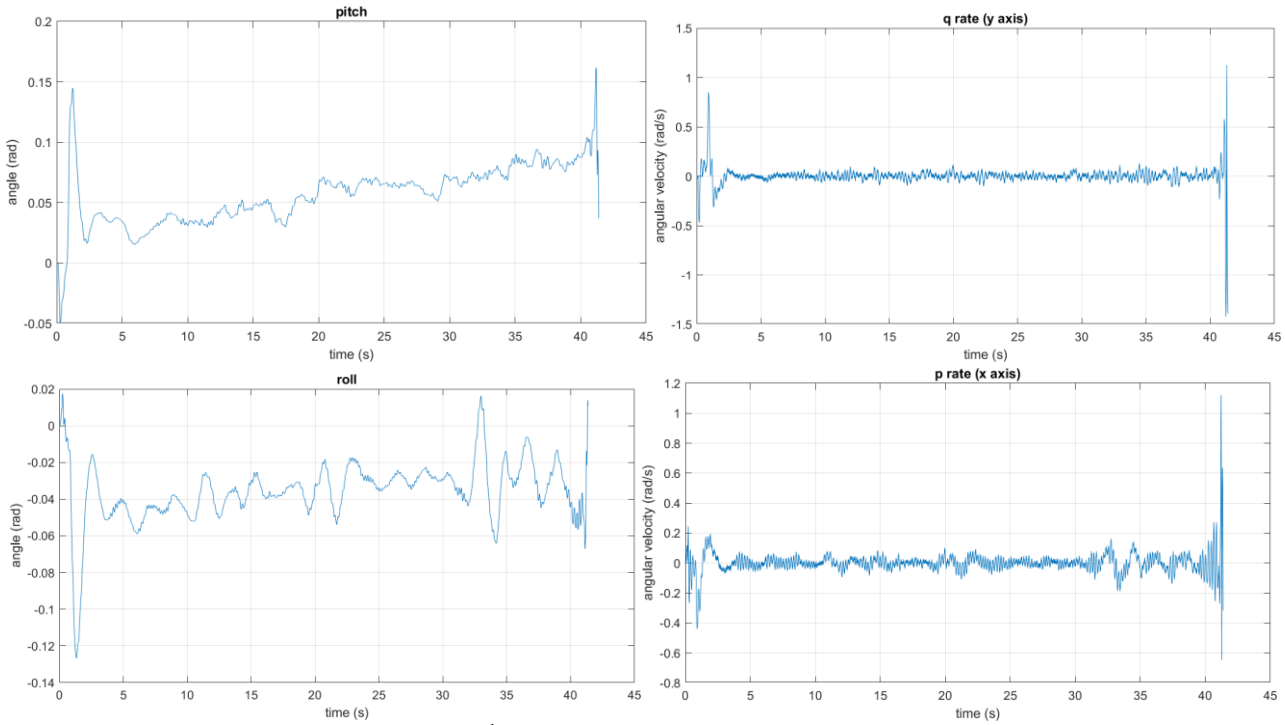


Figure 9.11 (continued): HIL test with 2nd algorithm –RPY angles (left) and corresponding angular velocities (right)

The next table (II) shows the time measurements of 5 different tests for 3 different speeds. They are defined by the gains corresponding to X and Y increments in the Tracking function block of the Path planning algorithm explained in chapter 8 and written in appendix A.2.2. The higher is the gain, the higher is the speed. Three different gains were picked. The optimal gain chosen is 0.001, and it was considered as the value in the middle, whereas 0.006 and 0.0014 were respectively chosen as the lower and the higher values.

During the simulation test, it was already noticed that gains higher than 0.0015 lead the drone to lose the track, for example in correspondence of right angles, because the image processing algorithm and the path planning algorithm cannot generate the reference signals on time to make the drone slow down and turn following the curve.

TABLE IV: ANALYSIS OF THE TIME OF COMPLETION OF THE TRACK (2nd ALGORITHM)

	TIME OF COMPLETION OF THE TRACK [s]		
	AT LOWER SPEED Gains of the increments in x and y position coordinates = 0.0006	AT MEDIUM SPEED Gains of the increments in x and y position coordinates = 0.0010	AT HIGHER SPEED Gains of the increments in x and y position coordinates = 0.0014
TEST 1	47	41	35
TEST 2	48	43	36
TEST 3	46	45	43
TEST 4	49	40	34
TEST 5	48	42	41
MEAN VALUE	47.6	42.2	37.8
STANDARD DEVIATION	1.02	1.72	3.53

Looking at the mean values, we can notice that the average time of track completion decreases if the gain, and therefore the speed, is increased, as it could be easily predicted. However, it could be more interesting to notice that the variance and standard deviation increased with speed. It means that at higher speeds, the uncertainty on the dynamics during the path tracking grows, and some curves require the drone to slow down to allow to follow the path and to oscillate while positioning to center the line. The motions in these conditions take additional time, and in some of the tests with higher gain, we can notice that the drone took the same time or even longer than in the tests with the medium gain value. This is the reason why the medium gain value was chosen as the optimal one for this implementation after the Hardware-in-the-loop test, and the implementation was validated.

The next figures (9.12 to 9.15) show the results of a test with a higher gain that gave slightly different results. In particular, the drone was less precise in tracking, having some issues at the first curve because it

was losing the track, but then it recovers oscillating, and it was still able to complete the track. This is the proof of the observation made before: a higher gain leads to a higher speed but with less precision in tracking the line.

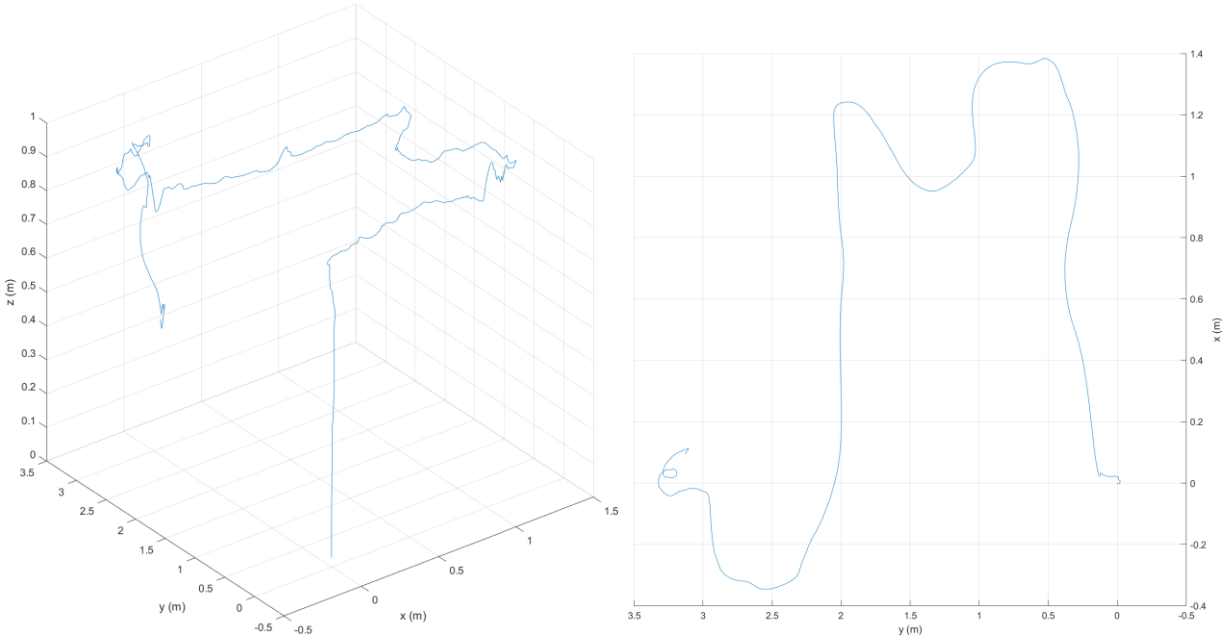


Figure 9.12: 2nd HIL test with 2nd algorithm – Trajectory (3D view and Top view)

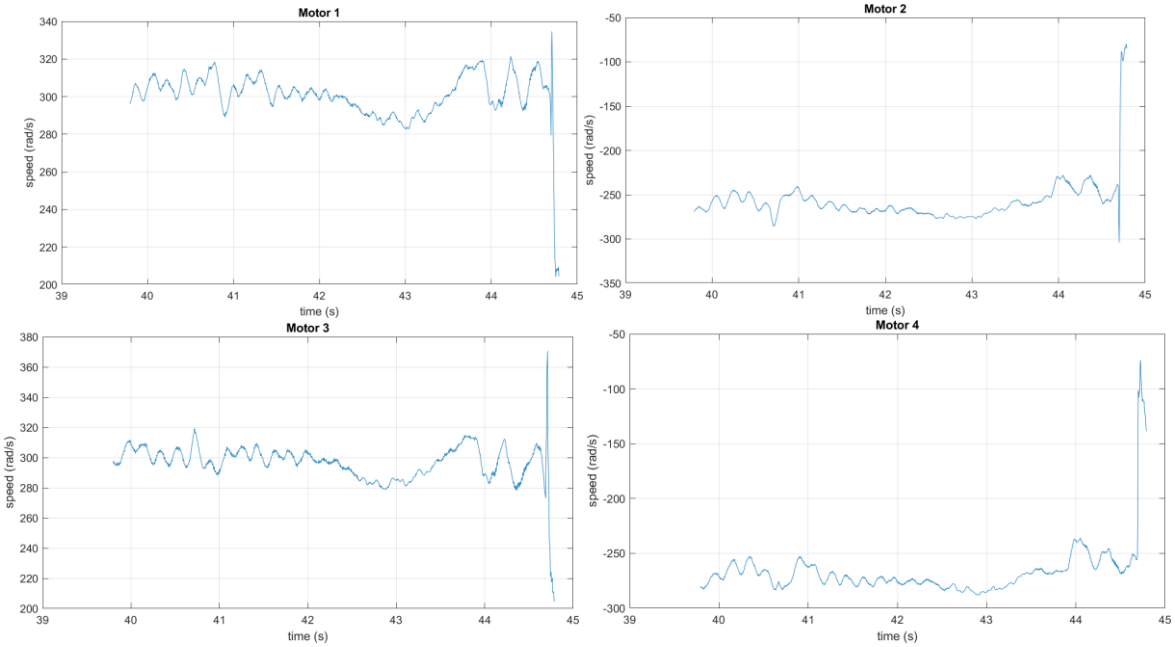


Figure 9.13: 2nd HIL test with 2nd algorithm – Motor speeds

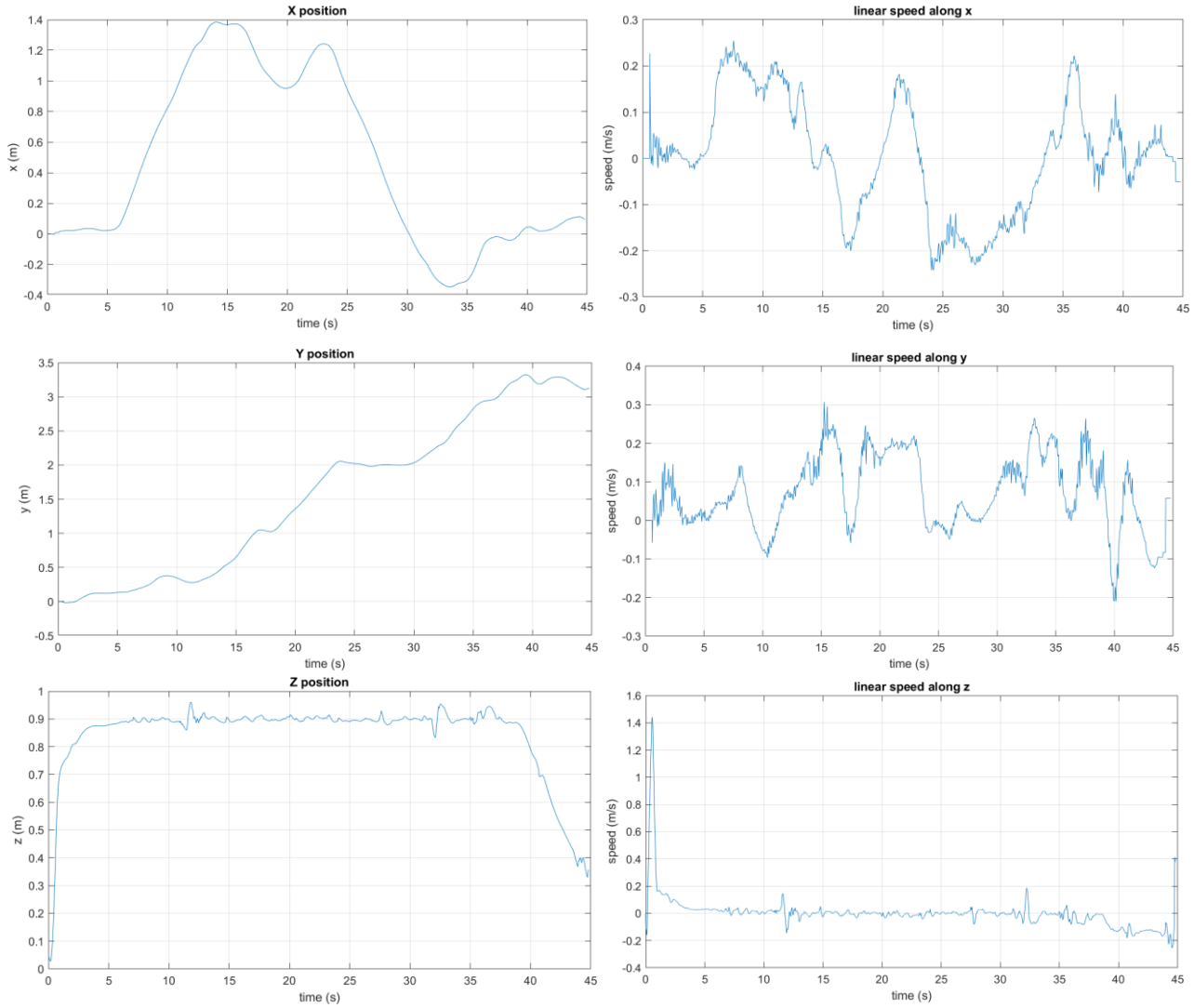


Figure 9.14 : 2nd HIL test with 2nd algorithm – X, Y, Z Positions (left) and corresponding linear speeds (right)

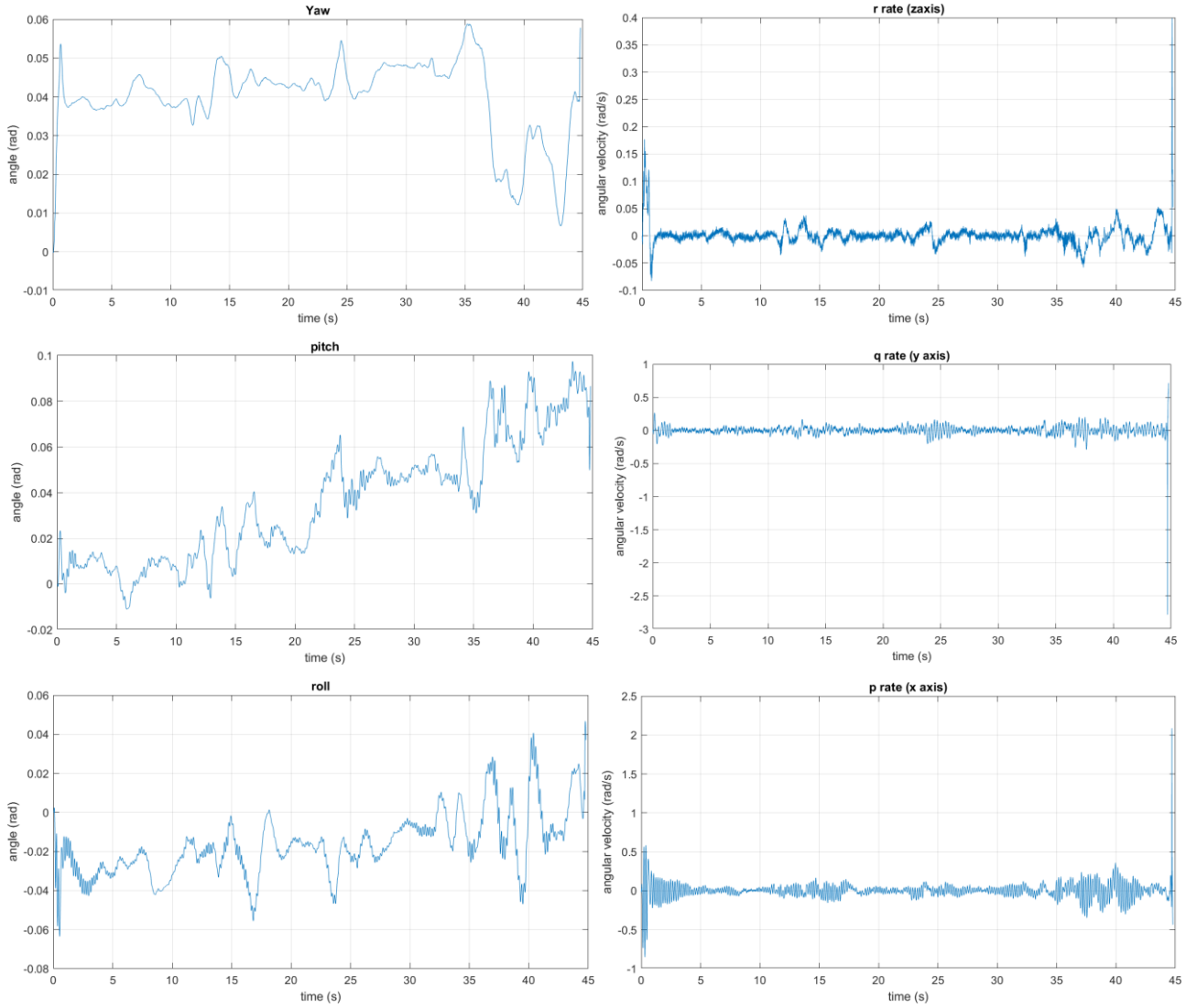


Figure 9.15: 2nd HIL test with 2nd algorithm –RPY angles (left) and corresponding angular velocities (right).

CHAPTER 10

CONCLUSION AND RESULTS COMPARISON

This Thesis project was useful to consolidate the awareness of the importance of a Model-based design technique in Control system applications, like the design of a drone Flight Control system, and to show its potential and its advantages during the development process. In particular, it was useful to design and tune the line-tracking algorithms in a more accessible way, allowing to simulate and test with real-time performances the effectiveness and the reliability of the control system.

Model-based design method gives the idea of how in the design process, the model performances change drastically from simulation to real-time execution, and it shows how each phase is essential to validate the control system and to ensure the robustness of the model.

The two different implementations with their variants showed how differently the system could behave between the simulation and real-time conditions, and especially in the latter, they revealed their weaknesses but also their strengths. In this conclusion, a comparison will be made to summarize the project work.

The first algorithm shows the advantages of being able to follow the line by changing its yaw angle and its orientation. Its image processing subsystem has a balanced load on the Hardware. Simple operations to detect the track directions with an analysis of the sub-images, based on the binary pixels count and the comparison of this sum with a certain threshold, are combined with the more complex operations of Hough transformation and Canny method to detect the values of the angles. The path planning algorithm has an easy way to implement the logic using State machine diagrams realized through Stateflow.

The second version of the first algorithm is characterized by even less complexity, without having the Hough and Canny functions and having a simplified steering control that stops the drone every time it changes the yaw angle, allowing more precision but decreasing the fluidity of movements.

However, these implementations have a weakness not to be neglected, which is highlighted in the real-time test: they strongly rely on the altitude, and they are susceptible to its little variations. In the Model-in-the-loop test, the hovering altitude was set as a constant, and the other parameters, like the drone speed and the sub-images dimensions, are tuned according to the altitude value. In the Hardware-in-the-loop test, the drone may not maintain the same altitude constant due to its estimation based on ultrasound sensors. It means the environment has a strong influence on the performances, and if it is not set with the proper condition, the algorithm does not work correctly, as it was proved by real-time test observation.

This issue does not affect the second algorithm implementation, whose image processing subsystem has a more adaptive and reliable way to detect track shape and to guide the path planning algorithm. Small altitude variations caused by the ultrasound sensors do not affect its performance significantly as in the first implementations, and the second algorithm was realized to overcome this issue that occurs in real-time performance. It has a different way of tracking the line, which is only based on x-y position in the space and not on yaw angle, as needed in the first algorithm to turn to change orientation in the curves.

However, this second version is more complex, and it weighs more on the Hardware computational power. Moreover, the second algorithm design is susceptible to the gain that controls the increments of position coordinates, meaning that the image processing algorithm is sensitive to speed. If the gain is high, the drone can still track the line, but it oscillates more, and it could more likely lose the position on the path if we think that in a real-time test, the states are estimated and are not exact as in simulation.

Finally, it is possible to conclude that the design satisfies the requirements through the low-cost hardware used in the experiments and through the final version of the line-tracking algorithm that can adequately perform the tasks even in a noisy environment used for real-time tests.

APPENDIX A

USER-DEFINED FUNCTIONS IN MATLAB

This Appendix contains the codes of all the user-defined Matlab functions of the implemented algorithms in Image processing and Path planning subsystems.

A.1 Image processing algorithm for the 2nd software implementation

A.1.1 "Binarization" function (using Color Thresholder App)

```
function BW = Binarization(R,G,B)

I = cat(3, R, G, B);

% Define thresholds for channel 1 based on histogram settings
channel1Min = 254.000;
channel1Max = 255.000;

% Define thresholds for channel 2 based on histogram settings
channel2Min = 254.000;
channel2Max = 255.000;

% Define thresholds for channel 3 based on histogram settings
channel3Min = 254.000;
channel3Max = 255.000;

% Create mask based on chosen histogram thresholds
sliderBW = ( (I(:,:,1) >= channel1Min) | (I(:,:,1) <= channel1Max) ) & ...
    (I(:,:,2) >= channel2Min ) & (I(:,:,2) <= channel2Max) & ...
    (I(:,:,3) >= channel3Min ) & (I(:,:,3) <= channel3Max);
BW = sliderBW;

end
```

A.1.2 "Filter" function

```
function BW_F = Filter(BW)

%%%% Tuning Parameters
% The lower is the Threshold, the more filtered is the image
Threshold_Filter = 3;

%%
% Output Image Dimensions
```

```
Width = 158;
Height = 118;

BW_F = zeros(Height,Width);
% Filter with a 3x3 box

for i = 2:(Height + 1)
    for j = 2:(Width + 1)
        BW_F(i-1,j-1) = sum(BW(i-1:i+1,j-1:j+1),'all');
        if BW_F(i-1,j-1) > Threshold_Filter
            BW_F(i-1,j-1) = 1;
        else
            BW_F(i-1,j-1) = 0;
        end
    end
end

end
```

A.1.3 "Ctrl_Radar" function

```
function [x_incr,y_incr,Start_End_Flag] = Ctrl_Radar(BW,xp,yp)

%%%% Tuning parameters
% Radar case size (must be odd dimensions)
% The larger is the size, the more robust is the system
box = 9;

% Detection threshold
Thrs = box*box/2;

% Search Resolution
Red_Srch = 1;  %in degrees (1/360)

%%

% Image Dimensions
Width = 158;
Height = 118;

Mag_vec = (Height/2)-((box-1)/2);

Angle_detection = zeros(1,(359+1)/Red_Srch);        %it has as many elements as the total
degrees (360) divided by the resolution
Angle_detection_centers = zeros(1,(359+1)/Red_Srch);

%Variable for centroid detection
Start_line = 0;

%square and centered matrix
BW2 = BW(:,20:139);

% Circular radar search
for i=0:Red_Srch:359

    y_b_pos = round(Mag_vec*sin(i*pi/180) + Height/2);
    if y_b_pos+((box-1)/2) > Height
        y_b_pos = Height-((box-1)/2);
```

```
    elseif (y_b_pos-((box-1)/2) < 1)
        y_b_pos = 1 + ((box-1)/2);
    end

    x_b_pos = round(Height/2 - Mag_vec*cos(i*pi/180));
    if x_b_pos+((box-1)/2) > Height
        x_b_pos = Height-((box-1)/2);
    elseif (x_b_pos-((box-1)/2) < 1)
        x_b_pos = 1 + ((box-1)/2);
    end

    Angle_detection(i+1) = sum(BW2(x_b_pos-((box-1)/2):x_b_pos+((box-1)/2),...
        y_b_pos-((box-1)/2):y_b_pos+((box-1)/2)),'all');
    if Angle_detection(i+1) >= Thrs
        Angle_detection(i+1) = 1;
    else
        Angle_detection(i+1) = 0;
    end
end

% Search for line centers
Angle_detection_Concatenated = [Angle_detection, Angle_detection];

Off_set = 1;
while Angle_detection_Concatenated(Off_set) == 1
    Off_set = Off_set + 1;
end

Angle_detection_Offset = Angle_detection_Concatenated(Off_set:Off_set+359);
White_Line = 0;

Pulse_rise = 0;
Pulse_down = 0;

for i=0:Red_Srch:359
    if (Angle_detection_Offset(i+1) == 1)
        White_Line = White_Line + 1;
        Pulse_rise = 1;
        if (i == 359)
            Pulse_down = 1;
        end
    elseif (Pulse_rise == 1) && (Angle_detection_Offset(i+1) == 0)
        Pulse_down = 1;
        Pulse_rise = 0;
    else
        Pulse_rise = 0;
        Pulse_down = 0;
    end
    if (Pulse_down == 1)
        White_Line = round(White_Line/2);
        Angle_detection_centers(i+1-White_Line) = 1;
        White_Line = 0;
        Pulse_down = 0;
        Pulse_rise = 0;
    end
end

% Detection of all line centers
Points_detected = 6;
```

```
x_c_det = 1000*ones(1,Points_detected);
y_c_det = 1000*ones(1,Points_detected);
Actual_pount_found = 1;
for i=0:Red_Srch:359
    if (Angle_detection_centers(i+1) == 1)
        x_c_det(Actual_pount_found) = Mag_vec*cos((i + Off_set)*pi/180);
        y_c_det(Actual_pount_found) = Mag_vec*sin((i + Off_set)*pi/180);
        Actual_pount_found = Actual_pount_found + 1;
    end
    if (Actual_pount_found > Points_detected)
        break;
    end
end

% Distance detection with respect to previous position
Dist_centers_wrt_previous = 1000*ones(1,Points_detected);
for i=1:Points_detected
    Dist_centers_wrt_previous(i) = sqrt((xp - x_c_det(i))^2 +...
        (yp - y_c_det(i))^2);
end

% minimum distance
[Min_dist,Indx_min] = min(Dist_centers_wrt_previous);

if (y_c_det(Indx_min) < 100) && (x_c_det(Indx_min) < 100)
    y_incr = y_c_det(Indx_min);
    x_incr = x_c_det(Indx_min);
else
    y_incr = 0;
    x_incr = 0;
end

if sum(Dist_centers_wrt_previous <= 1000) > 1
    Start_End_Flag = 1;
else
    Start_End_Flag = 0;
end

end
```

A.1.4 “Borderless” function

```
function BW_B_LESS = Borderless(BW_F)

% Figures that touch the edge of the image are removed
%%%% Tuning parameters
Border_size = 5;

%%
BW_B_LESS = BW_F;

% Image dimensions
Width = 158;
Height = 118;

% Top edge
BW_F(1:Border_size,1:Width) = BW_F(1:Border_size,1:Width).*-1;
% Bottom edge
```

```
BW_F(Height-Border_size+1:Height,1:Width) = BW_F(Height-Border_size+1:Height,1:Width).*-
1;
% Left edge
BW_F(Border_size+1:(Height-Border_size),1:Border_size) =...
    BW_F(Border_size+1:(Height-Border_size),1:Border_size).*-1;
% Right edge
BW_F(Border_size+1:(Height-Border_size),Width-Border_size+1:Width) =...
    BW_F(Border_size+1:(Height-Border_size),Width-Border_size+1:Width).*-1;

% Left - Right Search
for i = Border_size+1:(Height-Border_size)
    for j = Border_size+1:(Width-Border_size)
        if (BW_F(i,j) == 1) && ...
                ((BW_F(i-1,j) == -1) || ...
                (BW_F(i+1,j) == -1) || ...
                (BW_F(i,j-1) == -1) || ...
                (BW_F(i,j+1) == -1) || ...
                (BW_F(i-1,j-1) == -1) || ...
                (BW_F(i-1,j+1) == -1) || ...
                (BW_F(i+1,j-1) == -1) || ...
                (BW_F(i+1,j+1) == -1))

            BW_F(i,j) = -1;

        end
    end
end

% Right - Left Search
for i = (Height-Border_size):-1:Border_size+1  %decreasing
    for j = (Width-Border_size):-1:Border_size+1  %decreasing
        if (BW_F(i,j) == 1) && ...
                ((BW_F(i-1,j) == -1) || ...
                (BW_F(i+1,j) == -1) || ...
                (BW_F(i,j-1) == -1) || ...
                (BW_F(i,j+1) == -1) || ...
                (BW_F(i-1,j-1) == -1) || ...
                (BW_F(i-1,j+1) == -1) || ...
                (BW_F(i+1,j-1) == -1) || ...
                (BW_F(i+1,j+1) == -1))

            BW_F(i,j) = -1;

        end
    end
end

% Down - Top search
for j = (Width-Border_size):-1:Border_size+1      %decreasing
    for i = (Height-Border_size):-1:Border_size+1   %decreasing
    if (BW_F(i,j) == 1) && ...
                ((BW_F(i-1,j) == -1) || ...
                (BW_F(i+1,j) == -1) || ...
                (BW_F(i,j-1) == -1) || ...
                (BW_F(i,j+1) == -1) || ...
                (BW_F(i-1,j-1) == -1) || ...
                (BW_F(i-1,j+1) == -1) || ...
                (BW_F(i+1,j-1) == -1) || ...
                (BW_F(i+1,j+1) == -1))
```

```
            BW_F(i,j) = -1;

        end
    end
end

% Top - Down search
for j = Border_size+1:(Width-Border_size)
    for i = Border_size+1:(Height-Border_size)
    if (BW_F(i,j) == 1) && ...
                ((BW_F(i-1,j) == -1) || ...
                (BW_F(i+1,j) == -1) || ...
                (BW_F(i,j-1) == -1) || ...
                (BW_F(i,j+1) == -1) || ...
                (BW_F(i-1,j-1) == -1) || ...
                (BW_F(i-1,j+1) == -1) || ...
                (BW_F(i+1,j-1) == -1) || ...
                (BW_F(i+1,j+1) == -1))

            BW_F(i,j) = -1;

        end
    end
end


for i = 1:Height
    for j = 1:Width
        if (BW_F(i,j) == -1)
            BW_B_LESS(i,j) = 0;
        end
    end
end

end
```

A.1.5 "Filter_C_Det" function

```
function [C_DET, BW_FF] = Filter_C_Det(BW_B_LESS)

%%%% Tuning parameters
% The lower is the threshold, the more filtered is the image
Filter_Threshold = 3;

% Pixel threshold to detect circle
C_pixels = 500;

%%
% Image dimensions
Width = 156;
Height = 116;

BW_FF = zeros(Height,Width);

for i = 2:(Height + 1)
    for j = 2:(Width + 1)
        BW_FF(i-1,j-1) = sum(BW_B_LESS(i-1:i+1,j-1:j+1),'all');
```

```
        if BW_FF(i-1,j-1) > Filter_Threshold
            BW_FF(i-1,j-1) = 1;
        else
            BW_FF(i-1,j-1) = 0;
        end
    end
end

% Circle detection in the center
if sum(BW_FF,'all') > C_pixels

    C_DET = 1;

else

    C_DET = 0;

end

end
```

A.1.6 "Ctrl_Circle" function

```
function [x_Last, y_Last, Sum_end] = Ctrl_Circle(BW_FF)

%Trying to center the circle to land in the center
% Image dimensions
Width = 156;
Height = 116;

y_inc = sum(BW_FF(:,1:(Width/2)),'all');  %upper half sum
y_dec = sum(BW_FF(:,((Width/2) + 1):Width),'all'); %lower half sum
x_inc = sum(BW_FF(((Height/2)+1):Height,:),'all'); %right half sum
x_dec = sum(BW_FF(1:(Height/2),:),'all'); %left half sum

y_Last = y_dec - y_inc;
x_Last = x_dec - x_inc;

Sum_end = y_inc + y_dec; %whole circle image sum

End
```

A.2 Path-planning algorithm for the 2nd software implementation

A.2.1 "Z_Ctrl" function

```
function [z_sp, Enab_Go] = Z_Ctrl(t_in)

%%%% Tuning parameters
% SP altitude
Zsp = -0.9;

%%

if (t_in == 1)
    Enab_Go = 1;
```

```
else
    Enab_Go = 0;
end

z_sp = Zsp;
end
```

A.2.2 "XY_Ctrl" function (for Tracking)

```
function [x_inc, y_inc] = XY_Ctrl(Trck_ena,u)

%%%% Tuning parameters
Gain_x_dir = 0.001;
Gain_y_dir = 0.001;

%%

if Trck_ena == 1

    x_inc = u(1)*Gain_x_dir;
    y_inc = u(2)*Gain_y_dir;

else

    x_inc = 0;
    y_inc = 0;

end

end
```

A.2.3 "XY_Ctrl" function (for precision slow movements)

```
function [x_inc_F, y_inc_F] = XY_Ctrl(Trig_End,U_prev,Circle_Detected,Land_End)

%%%% Tuning parameters
Gain_x_dir = 0.0005;
Gain_y_dir = 0.0005;

%%

if (Trig_End == 0) && (Circle_Detected == 0) && (Land_End == 0)

    x_inc_F = U_prev(1)*Gain_x_dir;
    y_inc_F = U_prev(2)*Gain_y_dir;

else

    x_inc_F = 0;
    y_inc_F = 0;

end
end
```

A.2.4 "XY_Ctrl" function (for Landing)

```
function [z_final, x_inc, y_inc, Landing_Kill] = XY_Ctrl(u,C_R,u_F,C_B)
```

```
%%%% Tuning parameters

Last_Gain = 0.00035;
Z_gain = 0.0005;

Landing_Final_Size_Circle = 3000;

%%

if C_R == 1 && (u_F(1) == 0 && u_F(2) == 0)
    if C_B <= Landing_Final_Size_Circle
        x_inc = u(1)*Last_Gain;
        y_inc = u(2)*Last_Gain;
    else
        x_inc = 0;
        y_inc = 0;
    end
    Landing_Kill = 1;
    z_final = Z_gain;

else
    Landing_Kill = 0;
    x_inc = 0;
    y_inc = 0;
    z_final = 0;

end

end
```

A.3 Image processing for the 1st software implementation

```
function DegAngle = FindAngle(BWimage)

edgedBW = edge(BWimage,'canny'); %Canny;
[H, T] = hough(edgedBW); %Hough Transform;
P = houghpeaks(H,2); %peak values;
DegAngle = mean(T(P(:,2))); %found angle value;

end
```

CITED LITERATURE

1. PaperCraftSquare: Leonardo da Vinci's Aerial Screw Invention. http://www.papercraftsquare.com/ leonardo-da-vincis-aerial -screw-invention-free-paper-model-download.html [Online; accessed 02/15/20].

2. Pantalone, M.: Introduzione. In Modellazione e Simulazione Di Un Quadricottero Multirotore. Università di Bologna. 2015. https://amslaurea.unibo.it/9038/1/TESI_CORRETTA.pdf [Online; accessed 02/17/20].

3. Wikipedia: Unmanned Aerial Vehicle. https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle [Online; accessed 02/20/20].

4. International Business Times: Drones Aid Archaeologists in Exploring Ancient Sites, UAVs Like A 'New Set of Eyes' in Remote Areas. In Science. https://www.ibtimes.com/drones-aid-archaeologists-exploring -ancient-sites-uavs-new-set-eyes-remote-areas-1575023 [Online; accessed 02/23/20].

5. Microdrones: fast, Reliable, safe, cost-effective, uav/drone inspection tools from microdrones. https://www.microdrones.com/en/industry-experts/inspection/ [Online; accessed 02/22/20].

6. Resonon: Airborne Remote Sensing System. https://resonon.com/airborne-remote-system [Online; accessed 02/27/20].

7. Residence Style: The Use of Drones in Architecture Soars To New Heights. In Architect Design. https://www.residencestyle.com/the-use-of-drones-in-architecture-soars-to-new-heights/ [Online; accessed 03/01/20].

8. Wikipedia: Global Hawk. https://en.wikipedia.org/wiki/Northrop_Grumman_RQ4_Global_Hawk#/ media /File: Global_Hawk_1.jpg [Online; accessed 03/01/20].

9. Swissinfo: Using Swiss AI and drones to count African wildlife. https://www.swissinfo.ch/eng/ conservation_using-swiss-ai-and-drones-to-count-african-wildlife/44686308 [Online; accessed 03/01/20].

10. Youtube: DJI - M200 Series – Search and Rescue in Extreme Environments. https://www.youtube.com /watch?v=GkIJ2NJQHys [Online; accessed 03/02/20].

11. Vmire: Смотреть видео MIL SIL Tutorial: Part 1 на ВМире бесплатно. https://vmire.city/video/ dzgMYGWXpUc [Online; accessed 02/12/20].

12. Mathworks: Run on Target Hardware. Simulink Support package for Parrot Minidrones. https://www.mathworks.com/help/supportpkg/parrot/run-on-target-hardware-main.html [Online; accessed 02/17/20].

CITED LITERATURE (continued)

13. MathWorks: MathWorks Minidrone Competitions. https://www.mathworks.com/academia/student-competitions/minidrones.html [Online; accessed 02/10/20].

14. MathWorks: Rules and Guidelines. In MathWorks Minidrone Competitions. https://www.mathworks.com/content/dam/mathworks/mathworks-dot-com/academia/student-competitions/minidrone-competition/mathworks-minidrone-competition-guidelines.pdf [Online; accessed 02/12/20].

15. Totu, L., C.: Introduction to Quadrotors and Control Theory. Udemy. https://www.udemy.com/course/quadrotors/ [Online; accessed 02/15/20].

16. Orsag, M., Korpela, C., Oh P., Bogdan S.: Coordinate Systems and Transformations. In Aerial Manipulation. Advances in Industrial Control. 08 2018. Springer, Cham. ISSN 1430-9491 doi:10.1007/978-3-319-61022-1.

17. Aerodynamics for students: Blade Element Propeller Theory. Propulsion. http://www.aerodynamics4students.com/propulsion/blade-element-propeller-theory.php [Online; accessed 03/05/20].

18. Drayer, A., G.: Programming Drones with Simulink. Videos and Webinars. Mathworks. https://www.mathworks.com/videos/programming-drones-with-simulink-1513024653640.html [Online; accessed 02/10/20].

19. Douglas, B.: Drone Simulation and Control, Part 1: Setting Up the Control Problem. Videos and Webinars. Mathworks. https://www.mathworks.com/videos/drone-simulation-and-control-part-1-setting-up-the-control-problem-1539323440930.html7 [Online; accessed 02/12/20].

20. Parrot: Parrot Mambo Fly. https://www.parrot.com/us/drones/parrot-mambo-fly [Online; accessed 01/25/20].

21. Mathworks: Install RNDIS for Parrot Minidrones. Simulink Support package for Parrot Minidrones. https://it.mathworks.com/help/supportpkg/parrot/ug/intro.html [Online; accessed 02/06/20].

22. Karnavas, Y., L., and Chasiotis, I., D.: "PMDC coreless micro-motor parameters estimation through Grey Wolf Optimizer," 2016, XXII International Conference on Electrical Machines (ICEM), Lausanne, 2016, pp. 865-870, doi: 10.1109/ICELMACH.2016.7732627.

23. Parrot: Parrot Mambo Fly Quick Start Guide. https://www.parrot.com/files/s3fs- public/firmware/mambo_quick-start-guide_uk-fr-sp-de-it-nl-pt-ar1.pdf [Online; accessed 02/09/20].

CITED LITERATURE (continued)

24. Educational Innovations: Deluxe Safety Glasses. https://www.teachersource.com/product/deluxe-safety-glasses/lab-equipment-goggles [Online; accessed 02/10/20].

25. Banggod: Mini Wireless Dongle CSR 4.0 bluetooth Adapter V4.0 USB 2.0/3.0 For Win 7/8/10/XP For Vista 32/64. https://www.banggood.com/Mini-Wireless-Dongle-CSR-4_0-Bluetooth-Adapter-V4_0-USB-2_03_0-For-Win-7810XP-For-Vista-3264-p-1132661.html [Online; accessed 03/14/20].

26. CDW: C2G 2m USB Cable - USB 2.0 A to Mini USB B Cable (6.6ft) - USB Cable. https://m.cdw.com/product/c2g-2m-usb-cable-usb-2.0-a-to-mini-usb-b-cable-6.6ft-usb-cable/328953 [Online; accessed 25/03/20].

27. Amazon: Parrot Mambo – PCB + Viti. https://www.amazon.it/Parrot-PF070237-Mambo-PCB-viti/dp/B01MRL7UMA [Online; accessed 03/15/20].

28. Alibaba: MPU6050 IC 3 Axis Gyroscope. https://www.alibaba.com/product-detail/High-quality-MPU6050-IC-3Axis-gyroscope_60826768961.html [Online; accessed 02/05/20].

29. InvenSense: MPU-6000 and MPU-6050 Product Specification Revision 3.4. p. 21. https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf [Online; accessed 04/09/20].

30. Collins, D.: What are coreless DC motors? https://www.motioncontroltips.com/what-are-coreless-dc-motors/ [Online; accessed 03/02/20].

31. Mathworks: Quadcopter Project. Aerospace Blockset. https://www.mathworks.com/help/aeroblks/quadcopter-project.html?s_tid=srchtitle [Online; accessed 16/03/20].

32. Prouty, R.,W.: Helicopter performance, stability, and control. Malabar, Fla: Krieger Pub, 2005, ISBN 1575242095 9781575242095.

33. Pounds, P., Mahony, R. and Corke, P.: Modelling and control of a large quad-rotor robot. In: Control Engineering practice, pp. 691–699, Elsevier Ltd., 02 2010, ISSN: 0967-0661.

34. Douglas, B.: Drone Simulation and Control, Part 2: How Do You Get a Drone to Hover? Videos and Webinars. Mathworks. https://www.mathworks.com/videos/drone-simulation-and-control-part-2-how-do-you-get-a-drone-to-hover--1539323448303.html [Online; accessed 02/15/20].

35. Douglas, B.: Drone Simulation and Control, Part 5: Tuning the PID Controller. Videos and Webinars. Mathworks. https://www.mathworks.com/videos/drone-simulation-and-control-part-5-tuning-the-pid-controller-1540450868204.html [Online; accessed 02/21/20].

CITED LITERATURE (continued)

36. Douglas, B.: Drone Simulation and Control, Part 3: How to Build the Flight Code. Videos and Webinars. Mathworks. https://www.mathworks.com/videos/drone-simulation-and-control-part-3-how-to-build-the-flight-code-1539323453258.html [Online; accessed 03/12/20].

37. Van de Maele, P., J.: Reading an IMU Without Kalman: The Complementary Filter. Pieter-Jan, 04 2013. http://www.pieter-jan.com/ [Online; accessed 05/09/20].

38. Jouybari, A., Ardalan, A., A., and Rezvani, M.-H.: experimental comparison between mahoney and complementary sensor fusion algorithm for attitude determination by raw sensor data of xsens imu on buoy, Int. Arch. Photogramm. Remote Sens. Spatial Inf. Sci., XLII-4/W4, 497–502, 09 2017. https://doi.org/10.5194/isprs-archives-XLII-4-W4-497-2017.

39. Zhang, J. and Chen, G. X.: Research on the Geometric Distortion Auto-Correction Algorithm for Image Scanned. In: Applied Mechanics and Materials pp.644–650. 09 2014. 4477–8.1 https://doi.org/10.4028/www.scientific.net/amm.644-650.4477.

40. Mathworks: Optical Flow with Parrot Minidrones. Simulink Support package for Parrot Minidrones. Help Center. https://www.mathworks.com/help/supportpkg/parrot/ug/optical-flow-with-parrot-minidrones.html [Online; accessed 05/22/20].

41. Douglas, B.: Drone Simulation and Control, Part 4: How to Build a Model for Simulation. Videos and Webinars. Mathworks. https://www.mathworks.com/videos/drone-simulation-and control-part-4-how-to-build-a-model-for-simulation-1539585112546.html [Online; accessed 02/24/20].

42. Amazon: Issuntex 10X16 ft Gray Background Muslin Backdrop,Photo Studio, Collapsible High Density Screen for Video Photography and Television. https://www.amazon.com/gp/product/B07NQ7VFJT/ref=ppx_yo_dt_b_asin_title_o04_s00?ie=UTF8&psc=1 [Online; accessed 04/07/20].

43. Amazon: VATIN 4" Wide Double Faced Polyester White Satin Ribbon- 5 Yard/Spool, Perfect for Chair Sash, Making Bow, Sewing and Wedding Bouquet. https://www.amazon.com/gp/product/B079K8GN9B/ref=ppx_yo_dt_b_asin_title_o02_s00?ie=UTF8&psc=1 [Online; accessed 03/12/20].

44. Mathworks: Connect a Parrot Mambo Minidrone to a Windows System Using Bluetooth. Simulink Support package for Parrot Minidrones. https://www.mathworks.com/help/supportpkg/parrot/ug/connect-parrot-mambo-minidrone-to-computer-using-bluetooth.html [Online; accessed 02/18/20].

45. Mathworks: Hough. Image processing Toolbox. Help Center. https://www.mathworks.com/help/images/ref/hough.html [Online; accessed 04/18/20].

VITA

NAME: Paolo Ceppi

EDUCATION: Undergraduate exchange program ("SEC-U"), B. S. in Mechanical Engineering, Vanderbilt University, Nashville, TN, USA, 2017

B.S. in Mechanical Engineering, Politecnico di Torino, Turin, Italy, 2018.

M.S. in Mechatronic Engineering, Politecnico di Torino, Turin, Italy, 2020.

M.S. in Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, IL, USA, 2020.

HONORS: "SEC-U" Program Scholarship, Politecnico di Torino and Vanderbilt University of Nashville

"TOP-UIC" Program Scholarship, Politecnico di Torino and the University of Illinois at Chicago